Formalizing and Implementing a Reflexive Tactic for Automated Deduction in Coq

Stephane Lescuyer

ORSAY
N° d'ordre:

UNIVERSITÉ DE PARIS-SUD 11
CENTRE D'ORSAY

THÈSE

présentée
pour obtenir

le grade de docteur en sciences DE L'UNIVERSITÉ PARIS XI
Discipline : Informatique

PAR

Stéphane LESCUYER

SUJET:

Formalizing and Implementing a Reflexive Tactic for Automated Deduction in Coq

Formalisation et Développement d'une Tactique Réflexive pour la Démonstration Automatique en Coq

soutenue le 4 janvier 2011 devant la commission d'examen composée de

Sylvain Conchon	(Encadrant)
Évelyne Contejean	(Encadrante)
Pierre Crégut	(Rapporteur)
Hugo Herbelin	(Examinateur)
Sava Krstic	(Examinateur)
Shankar Natarajan	(Rapporteur)
Burkhart Wolff	(Examinateur)

Acknowledgements

This work would not have been possible without the help and assistance of many people, and even though a PhD thesis ultimately is an individual work, a lot of credit should go to those who supported me one way or another during the last three to four years.

First and foremost, my thanks go to my advisors, Évelyne Contejean and Sylvain Conchon. They granted me an immensely interesting subject, which ambitiously aimed at bridging the gap between interactive and automated provers, killing two birds with one stone: bringing automation to Coq on one side, and formal soundness to our automated prover on the other side. The numerous sleepless nights I spent working on details that were at times nasty, at times nifty, are as good a measure as any of an original and interesting PhD topic. Even though I did not achieve as much as I would have liked – but do we ever? – I learned many things about formal methods and certainly hope this work can be put to good use, and for all that, I am very thankful to Évelyne and Sylvain. I especially thank Sylvain for how available he was every time I needed advice or wished to share my latest progress, and for the many friendly discussions we had along the way.

I sincerely thank Pierre Crégut and Shankar Natarajan for accepting to review the earlier version of this document, and for their numerous corrections and constructive comments. I also thank my other examiners Hugo Herbelin, Sava Krstic, and Burkhart Wolff, for accepting to be members of my defense committee and for their insightful questions and remarks on that day.

I was fortunate enough to be part of the Proval team while preparing this work, and I thank everyone in the team for the kindness and good spirits they showed week after week. Be it cake receipes, soccer games or challenging puzzles, there was always something to share and chat about in the coffee room besides everyone's ongoing work. I am very grateful to Jean-Christophe Filliâtre for how much I learned about data structures and algorithms through him and the passion he shows for such things, and TAOCP will always have a front row seat on my bookshelves. I also want to express my gratitude to Matthieu Sozeau and Guillaume Melquiond for passing their experience with Coq to me, and the discussions I had with them helped me greatly on some aspects of this work. Большое спасибо to Andrei Paskevich and his amazing ability to give relevant and insightful remarks on just about every topic.

To my coworkers and friends, you made the last 3+ years go way too quickly! Johannes, thank you for bearing with me when I kept disturbing you in the office, you know I'm always in for a beer on a CL evening. Florence, Louis, I'm glad I could help you explore the amazing world of Coq-iness, and Florence, I'm afraid you aren't done with me yet! Yannick, your cheerful attitude makes everyone forget how old you really are :-) Kalyan, fare thee well, too bad I have no one to talk cricket with anymore. To our younger members in the team, François, Cédric, Asma, Paolo, Mohamed, and Claire, I wish you all the very best for your PhDs.

I want to thank Assia Mahboubi, Pierre Letouzey, Pierre Castéran, Thomas Braibant, Benjamin Grégoire, and Aaron Stump, for helping me one way or another, by giving advice when I was stuck on technical issues or simply by being supportive of my work. I also want to thank the people who, in retrospect, gave me the will and motivation to engage in a PhD: Gilles Dowek, Benjamin Pierce and Dale Miller. Years ago, through your teaching or tutoring, you raised my interest in computer science and in logic and formal methods in particular, and I am sure you kept and keep inspiring many students to follow in your tracks.

Un grand merci enfin à mes chers parents, pour m'avoir soutenu patiemment dans ces études qui n'en finissaient pas. J'espère vous avoir rendus fiers, puissiez-vous voir en ce travail un témoignage de ma gratitude.

À Christine.

Contents

Introduction

> ...et remarquant que cette vérité, je pense, donc je suis, était si ferme et si assurée, que toutes les plus extravagantes suppositions des sceptiques n'étaient pas capables de l'ébranler, je jugeai que je pouvais la recevoir sans scrupule pour le premier principe de la philosophie que je cherchais.
>
> René Descartes, *Discours de la méthode*

Contents

A Short History of Formal Logic

When René Descartes asserted the famous "I think, therefore I am" in his *Discourse on Method*, his justification for this statement was that it "was so firm and so assured that all the most extravagant suppositions of the sceptics were unable to shake it". This informal kind of reasoning, based mainly on an intuitive notion of *truth*, on common sense and dialectics, had been for centuries the foundation for argumentations in every field of what was then called philosophy, a concept which included both natural and human sciences. In particular, advances in algebra, analysis and mathematics in general had been relying on an intuitive and well-accepted notion of proof.

As a matter of fact, Descartes was an accomplished mathematician himself and published, as an appendix to the *Discourse on Method*, his breakthrough approach to analytic geometry which fostered the rise of cartesian coordinate systems and calculus.

Over time, as mathematicians were working towards more and more complex results, the issue was raised of whether the intuitive approach was sufficient or whether a more formal language was required to describe mathematics and logical reasonings. As early as the end of the 17th century, Leibniz wished for a *calculus ratiocinator*, a formal logical and algorithmic language, which, in regard to modern computer science and proof theory, was an incredibly insightful and pioneering concept. It was not before the end of the 19th century that this idea started becoming reality, with the publication of Gottlob Frege's *Begriffsschrift* in 1879, and the later *Grundsgesetze der Arithmetik* in 1903. His work provided the first formal presentation of first-order logic and even if it was proved inconsistent by Russell's paradox, his system was the basis of many a work on the foundations of mathematics around the turn of the 20th century.

As the search for a novel foundation of mathematics led to the Zermelo-Fraenkel theory, an ambitious program launched by David Hilbert aimed at finding a consistent formal theory relying on a small number of well-understood axioms and on the basis of which all mathematics could be assembled. Kurt Gödel soon brought a negative answer to this ambition: his first incompleteness theorem shows that there does not exist a consistent system where all true properties are provable, as soon as a system embeds non-trivial arithmetic reasoning. Nevertheless, Gödel's discovery did not completely put a stop to Hilbert's program and later research focused on finding consistent logical systems which were expressive enough to formalize interesting fragments of mathematics.

In 1934, Gehrard Gentzen introduced the notion of sequent and proposed the two sequent calculi LJ and LK, respectively for intuitionistic and classical first-order logic. These calculi are expressed in terms of deduction rules between sequents, for instance the following rule of LJ:

$$\frac{\Gamma, A \vdash C \qquad \Sigma, B \vdash C}{\Gamma, \Sigma, A \vee B \vdash C} \,(\vee L)$$

means that if one can prove C from A and the assertions in Γ, and also from B and the assertions in Σ, then C can be proved from $A \vee B$ and the assertions in Γ and Σ. When read bottom-up, Gentzen's rules can be seen as instructions on how to construct a proof of the bottom statement. This analogy is fundamental since it means the rules describe a way to systematically search for a proof of a given statement, as long as there is only a finite way of applying them for any statement. In the absence of quantifiers, this condition is guaranteed by the fact that Gentzen's calculi satisfy the *cut-elimination* property, *i.e.* that the following rule:

$$\frac{\Gamma \vdash A \qquad \Sigma, A \vdash B}{\Gamma, \Sigma \vdash B}\ (Cut)$$

also known as *modus ponens*, can be removed from the system without reducing its expressiveness. In this regard, Gentzen's sequent calculi represented an important breakthrough and has had an important impact on the development of proof theory and automated deduction.

Towards Mechanized Reasoning

Automated Theorem Proving

With the development of computing systems, the second half of the 20th century made it possible to finally put into practice deduction systems such as Gentzen's sequent calculi which had been studied in the first half of the century. Although Church and Turing had independenly proved in the 1930s that first-order logic was not decidable, it remained to be seen whether computers could nonetheless automatically prove interesting formulae.

The first major works in automated deduction were Newell, Simon and Shaw's Logic Theory machine in 1956 [NSS57] and Wang's work [Wan60]. Both aimed at automatically proving a variety of first-order tautologies found in Russell and Whitehead's *Principia Mathematica*, but using quite different approaches. The Logic Theory machine attempted to prove a statement by following heuristics to perform a mix a backward and forward reasoning, thus becoming one of the first achievements in the field of *artificial intelligence*. On the other hand, Wang followed an algorithmic approach and based his procedure on sequent calculus, systematically exploring the possible proofs of a statement. Wang's approach fared better than the Logic Theory machine and gave the tone to later automated theorem provers (ATP).

The 1960s saw the development of the DPLL procedure [DP60, DLL62] to efficiently decide validity in propositional logic, and a major breakthrough was initiated by John A. Robinson's *resolution* rule [Rob65]. Resolution was very popular, in particular for its ability to deal with first-order logic, and led to the development of the logical programming language Prolog. Resolution is still in use in many modern ATPs. In order to become more versatile, automated deduction systems needed to go beyong propositional reasoning and deal for instance with the frequently used equality predicate. To that end, the paramodulation [RW69] rule was designed in order to achieve better equational reasoning.

As interest in ATP systems grew, so did the number of potential applications and the variety of formulae to discharge. In particular, many applications (notably software verification) required proving the validity of formulae in logics more constrained than first-order predicate logic with equality: integer arithmetic often became essential, and other theories such as arrays or bitvectors as well. To deal with these theories, an axiomatic approach in a standard ATP is not satisfactory and specific decision procedures were developed instead. The last decade has seen a very active development in the field of Satisfiability Modulo Theory (SMT) solvers, an alternative category of automated deduction systems which started around 1980. These SMT solvers decide the satisfiability of formulae by combining a propositional solver with decision procedures dedicated to background theories such as linear arithmetic. SMT solvers will be at the heart of our dissertation and we present them in more detail in Chapter 1.

Interactive Theorem Proving

In parallel to the development of automated theorem proving, others started using deductive systems in order to verify the validity of existing proofs. This task was particularly amenable to mechanization since it was both tedious and decidable. There were also some systems which were neither automated theorem provers nor proof checkers, but somewhere in the middle. This was the case of the Boyer-Moore prover, which was based on resolution but allowed the user to give directives at different points during a proof. We can consider that such a system is a proof checker since the "proof" consists in the sequence of directives, but how complicated can proof steps be if we are to qualify a system as a proof checker? A qualitative answer to this question was given by de Bruijn's criterion: the correctness of the proof checker as a whole shall only depend on a very small, well-understood, kernel. The Boyer-Moore prover, or any other automated theorem prover for that matter, hardly satisfies this criterion, and systems which verify this criterion have not been developed on top of techniques like resolution, but on *type theory*.

Type theory was introduced by Russell and Whitehead in their *Principia Mathematica* in order to avoid the inconsistency of Frege's approach as revealed by Russell's paradox. Zermelo-Fraenkel's set theory remained (and still remains) the preferred logical foundation for mathematics, but the interest in type theory was renewed by Church's invention of λ-calculus after it was discovered that there exists a strong correspondence between the deduction rules in type theory and a typing system for λ-calculus. This correspondance is known as the *Curry-Howard isomorphism* and allows one to identifies programs to proofs, and types to propositions: if there exists a ground λ-term t of type τ, then τ is a tautology and t is a proof of that tautology. The characterization of proofs as programs denotes the constructive nature of this formalism and it is not surprising that it is only describing intuitionistic logic. A proof checker for such a system is therefore simply a type-checker for λ-terms; in particular, it satisfies the de Bruijn criterion because it is quite reduced and is entirely described by a small set of typing rules.

A limitation of type theory is that only formulae which correspond to types of terms can be expressed in this framework, and simply-typed λ-calculus is not very expressive in that regard. In order to express richer properties, Martin-Löf proposed an intuitionistic type theory [ML75] richer than Russell and Whitehead's, insofar as it is possible to quantify over objects and types using a dependent product operator. By using dependent types, it is possible to express properties quantified by objects and which depend on the value of these objects, which makes it much more expressive than simple type theory. Another important change is that since terms are part of types, they can be reduced and therefore there is a natural notion of computation in the logic. The Calculus of Constructions, due to Coquand and Huet [CH88], can be seen as a higher-order extension of Martin-Löf's type theory.

The first proof checker based on type theory was Automath [dB94]: it was developed in 1968 by de Bruijn and would take a full proof term and verify it. Later came LCF, which relied on a proof language which had a big impact in the field of programming languages since it is at the basis of languages of the ML family. LCF had a revolutionary architecture which is now common to all so-called LCF-style provers, like HOL [hol], and which consists of a dedicated language of commands called *tactics* based on a small set of elementary rules. LCF used abstract types to prevent theorems to be built from other means than this reduced kernel. Because these systems allow one to iteratively build a verified proof, they are called *interactive provers* in contrast to automated provers.

Modern interactive provers based on type theory can be classified in two different families. Like LCF, the first family uses type theory as a meta-logic to justify basic inferences steps allowed by the prover. This family includes provers such as Isabelle [Isa] or TweLF [PS99]. The other class of interactive provers rely on a type theory and simply implement a typechecker for terms in this theory. Among these systems, NuPrl [NuP] and Agda [BDN09] are based on Martin-Löf's type theory, while Lego [Leg], Matita [ACTZ07] and Coq [Coq] are based on a variant of the Calculus of Constructions. Coq is our interactive prover of choice in this thesis and we discuss its logic and its architecture in much more detail in Chapter 4.

Combining Interactive and Automated Approaches

Modern interactive provers use very expressive logics based on type theory and therefore allows for an intuitive formalization of mathematical concepts. They can thus be used to formalize complex concepts and achieve complex proofs, which are way beyond the capabilities of automated theorem provers. Unfortunately, they can be very tedious to work with because proofs must be justified by small basic steps and therefore require much more detail than even the most detailed pencil-and-paper proof. Moreover, in very big proofs, it is often the case that there are just a few key arguments requiring human thinking and the remaining of the proof is then simple enough to be discharged by an automated prover.

This is therefore a natural idea to try and combine the interactive and automated approaches by using an automated prover to discharge easy enough goals during an interactive proof. Unfortunately, automated provers, as we explained, are complex systems which do not meet de Bruijn's criterion and therefore they cannot be embedded as such in an interactive prover without compromising its kernel. There is actually concern over the correctness of ATPs and SMT solvers considering the complexity of these systems and the fact that they are being used for critical software or hardware verification.

There exists a category of systems which take a less sceptical stance than the interactive provers cited above, and which dilutes the de Bruijn criterion. Such systems include ACL2 [ACL] (the descendant of the Boyer-Moore theorem prover), the PVS specification and verification system [PVS], or the Atelier B based on the B-Method [Abr96] (which has the particularity of relying on set theory). These verification systems provide an expressive logical language to formalize programs or mathematics and to write precise specifications about these formalizations. They also provide an interactive way of proving these properties in a manner similar to proof assistants, but with the help of automated decision procedures. These tools are very popular because they allow one to write formal specifications while the proving phase is assisted by automated provers and is therefore less tedious than typical interactive provers.

For those systems which still want to keep a small trusted kernel and not rely on automated provers directly, the integration of automated methods is a real challenge. In order to be trusted by the interactive prover, the automated prover must not only find a proof, it must explain its proof in terms of the basic steps accepted by the proof checker. This explanation is called a *proof trace* and since the steps accepted by the interactive prover are so basic, instrumenting an automated prover to return proof traces suitable for the interactive prover is a complex task. It is usually done in two steps, with the solver returning an intermediate proof trace which is further transformed into an object suitable for the proof checker (that second phase is called *proof reconstruction*).

Another way to proceed is to use the ability of the logic to embed computations, and more generally programs. Along with the ability of higher-order logic to reflect itself [Har95, BM90], this feature makes it possible to use a technique of proof by *reflection*. This consists in implementing a decision procedure directly as a program in the logic, and using the correctness of this implementation, prove formulae by a simple computation of the procedure. We will make use of this method in this thesis and it will be explained in detail in Chapter 4

Contributions

We now present the contributions of this dissertation. We have seen that interactive provers allow complex formalizations at the price of tedious proof developments, while automated theorem provers do not require human intervention but raise soundness issues. We are interested in the soundness of the SMT solver Alt-Ergo and use the Coq proof assistant to formally verify Alt-Ergo's core components. This leads to the two following contributions.

A Formally Verified SMT Solver Kernel

Our first contribution in this work is to have formalized Alt-Ergo's kernel components and formally established the correctness of this formalization in the Coq proof assistant. This kernel consists in a propositional solver based on the DPLL procedure, extended with standard optimizations, along with an original decision procedure combining the theory of equality on uninterpreted functions with an arbitrary theory under certain conditions. Because this procedure, called CC(X), is novel, it is all the more important that it is proved sound and complete in a formal setting.

This formalization and verification of Alt-Ergo's kernel dramatically increases the trust that we can have in Alt-Ergo; in particular developing the proof has helped us better understand some of the details of the algorithm and make sure of the conditions where it could be applied. This is particularly interesting because Alt-Ergo is used to discharge proof obligations coming from software verification systems, and must therefore be reliable.

A Reflexive Tactic for Automated Deduction

Our second contribution is to extend our Coq verification of Alt-Ergo's kernel in such a way that it is possible to use the underlying decision procedure as a Coq tactic. We do not extend Coq's trusted code base or perform proof reconstruction from Alt-Ergo; instead, we formalize the kernel's components by writing an effective implementation in the Coq proof assistant. This approach raises some issues since it amounts to reimplementing the solver's kernel in the context of the pure programming language contained in Coq's logic, and do it in such a way that it can be computed reasonably efficiently. In order to be used to prove Coq's formulae, we use the principle of proof by reflection and therefore we have to define semantics of the concrete objects manipulated by our algorithm which can be lifted to Coq's own notion of validity. Another critical point is the reification phase: the translation of Coq's formulae in concrete objects which represent them and on which the algorithm can be applied.

By following this approach, we develop a reflexive tactic which effectively combines three useful theories: propositional logic, equality with uninterpreted functions, and linear integer arithmetic. These three theories are ubiquitous in usual Coq developments and such a tactic is the first which can handle their combination. Indeed, many evolved tactics exist in Coq to deal with some logical fragment but it is generally impossible to combine them. Consequently, these existing tactics only work for formulae which are, for instance, purely arithmetic, purely propositional, or purely equational. Providing a tactic which actually combines these three fragments represents a real contribution towards more automation in Coq.

Throughout this development, we also implement components which are highly reusable and are not specific to our particular goal. For instance, we provide a library for ordered types and generic data structures commonly used in programming language. Such extensions are valuable to the Coq community since existing reusable components help develop faster programs. This is even more significant than in a standard programming language since components developed in Coq must also come with specifications and proofs, and thus are particularly time-consuming to reimplement.

Outline

This thesis is organized in two parts.

The first part is devoted to the mathematical formalization of Alt-Ergo's quantifer-free kernel. Chapter 1 presents the origin of SMT solvers and the architecture of Alt-Ergo. In Chapter 2, we present a formalization of the propositional solver at the heart of our SMT solver. This propositional solver is based on a standard DPLL procedure, which we formalize as an inference system. We also show how to extend this system to commonly used optimizations such as conflict-driven clause learning, and also discuss adaptations required for use in an SMT solver. Chapter 3 details Alt-Ergo's original combination scheme CC(X) used to perform congruence closure modulo a theory X. We also show how we extend this system in order to deal with disequations.

The second part is devoted to the implementation of a Coq reflexive tactic based on the formalization presented in the first part. Chapter 4 presents the Coq proof assistant, its logic, its specificities, and the approach of proof by reflection as well as other approaches for automating deduction in Coq. Chapter 5 presents a Coq library of first-class containers which provides common structures such as ordered types, finite sets and finite dictionaries, and which are fundamental to implementations in later chapters. Chapter 6 presents the Coq formalization of Alt-Ergo's propositional solver and how it can be instrumented into a reflexive tactic to automatically discharge propositional tautologies. We address the issue of conversion to conjunctive normal form in Chapter 7, where we present how to adapt the propositional solver in order to use a lazy conversion scheme. Chapter 8 presents the modifications which must be done in order to extend the propositional solver to an SMT solver and in order to extend the tactic's reification process to equalities between terms on an arbitrary signature. We then formalize and implement the combination scheme $\mathsf{CC(X)}$ in Chapter 9 and show how it can be plugged in the propositional solver to extend the tactics to propositional logic modulo equality. Chapter 10 finally presents the implementation of the theory of linear arithmetic and how it can be used in our framework.

We conclude in Chapter 11 with a presentation of the whole system implemented in Coq and its capabilities. We also address the various limitations and possible extensions which we envision.

Part I

Formalization of an SMT Solver's Kernel

CHAPTER 1

Solvers for Satisfiability Modulo Theories

> Ce n'est pas quand il a découvert l'Amérique,
> mais quand il a été sur le point de la découvrir,
> que Colomb a été heureux.
>
> Fiodor M. Dostoïevski, *L'Idiot*

Contents

This first chapter introduces and presents the Alt-Ergo tool, which is at the basis of the formalizations we present in this document. Alt-Ergo belongs to a family of tools called *SMT solvers*, where SMT stands for *Satisfiability Modulo Theories*. Section 1.1 is devoted to an informal presentation of the SMT decision problem and the field of SMT in general. In Section 1.2, we then present Alt-Ergo and show how it is dedicated to a certain class of problems that arise in program verification.

1.1 Satisfiability Modulo Theories

In the field of automated deduction systems, the two most popular subfields are SAT solvers on one side, and general first-order automated theorem provers (ATP) on the other side. Users of such deduction systems often want to know the satisfiability, or equivalently the validity, of formulas in a logic which is more expressive than propositional logic, but more restrained than first-order logic. Typically, these users are interested in the satisfiability of first-order formulae where some predicate or function symbols have a predetermined interpretation. For instance, the following formula:

$$x = 0 \implies f(2 + x) = f(2)$$

is not valid in general because 0, 2, + and even = can have nonstandard interpretations, but these nonstandard models are of no interest and this formula is indeed valid if the equality and arithmetic symbols have their standard meaning. The interpretation of the predetermined symbols is often called the *background theory*, and the problem of deciding the satisfiability of a formula with respect to such a background theory is called satisfiability modulo theory.

In order to deal with background theories in traditional automated deduction systems, one must somehow be able to impose the theory constraints to the prover. This can be done in different ways whether one is considering a generic ATP or a SAT solver.

The only way to force first-order automated theorem provers to only consider models which are consistent with the background theory is to add axioms to the formula which describe the theory. This is only possible when the theory is axiomatizable, or more precisely *finitely axiomatizable*, *i.e.* when there exists a finite set of first-order formulae which exactly describe the theory. For instance, considering the fact that almost all ATPs deal with equality adequately, the formula above can be proved valid by such ATP simply by adding the following two axioms:

(i) $\forall xyz,\ x + (y + z) = (x + y) + z$

(ii) $\forall x, x + 0 = x = 0 + x$

which describe + as a monoid operation whose neutral element is 0. The performance of dealing with interesting theories through such axiomatization is often unacceptable, but more importantly, a great number of interesting theories are not finitely axiomatizable. For instance, Tarski's axiomatization of real numbers [Tar46] cannot be expressed with a finite number of axioms, neither can Presburger arithmetic [Pre29]. All the theories of inductive datatypes with a finite number of constructors (such as finite trees [BRVs95] for instance) are not finitely axiomatizable either, because second-order logic is required to express the induction principle.

We have seen that some theories cannot be axiomatized in an ATP; however, for many such theories, as those cited above, there exists decision procedures for the satisfiability of quantifier-free formulae. Such decision procedures have been actively studied in the last two decades and there is a growing list of decision procedures for theories with practical applications. The research on SMT has been concerned with the problem of integrating these decision procedures in SAT solvers in order to solve the SMT problem for the corresponding theories. Early research on the problematic of incorporating decision procedures in formal provers was performed more than thirty years ago by the likes of Shostak [Sho78, Sho79, Sho84], Nelson and Oppen [NO79, NO80], and later by Boyer and Moore [BM88, BM90] in their Boyer-Moore prover. The interest in SMT research rose again at the end of the 1990s and has since been very active, both on theoretical and practical aspects. SMT solvers have been developed in academia as well as in the industry; an annual workshop brings together users and developers of the SMT community; a common pool of benchmarks has been established [BST10] in order to measure the progress of the systems and a competition [SMT] is organized in order to compare their relative strengths and weaknesses. Techniques and systems from the SMT community are now used in a variety of domains such as static checkers or verification systems (this is the case for Alt-Ergo, see Section 1.2), model checkers (BLAST), interactive theorem provers (HOL, PVS), etc.

There are two main approaches when designing an SMT solver, which are known as the *eager* and the *lazy* approach. Alt-Ergo, like most other systems, follows the lazy approach and we will present this architecture in detail in the next section. Whereas lazy SMT solvers rely on the dynamic combination of a SAT solver and a decision procedure for the theory literals, eager SMT solvers try to express all the possible useful theory constraints related to a formula and translate this formula in order to add all these constraints and retain equisatisfiability. The translated formulae are then passed on to a standard SAT solver. A survey with many details on modern SMT techniques in both lazy and eager SMT solvers is available in [BSST09].

1.2 Alt-Ergo: an SMT Solver Dedicated to Program Verification

We now present Alt-Ergo, an SMT solver dedicated to program verification. Before we detail its architecture, we look into the context of program verification.

1.2.1 Program Analysis and Software Verification

There exists a broad range of techniques which aim at ensuring certain properties (or, equivalently, avoiding certain run-time errors) in computing systems. The main characteristics that allow one to classify these techniques are whether they are automatic or human-driven, and whether they happen at run-time (dynamic) or are performed statically. For instance, research on programming languages leads to type systems which statically ensure that all well-typed programs will verify some properties (basically the absence of crash due to typing errors, but also the absence of null dereferencing in languages like OCaml, C# or Haskell) while other languages (typically scripting languages like Python, PHP or JavaScript) only provide dynamic type-checking.

In order to statically verify more complex properties of programs, for instance detecting divisions by zero, out-of-bounds accesses, overflows and other typical dangerous situations a program can encounter, techniques like model-checking, abstract interpretation or static analysis can be used. These techniques can be fully automated or simply semi-automated, but in any case require typically much less manual effort than full formal verification using proof assistants such as HOL, Isabelle or Coq. The amount of manual work required usually depends on the complexity of the properties that one wants to establish. Examples of these systems, called extended static checkers, include Spec# [BRS05], ESC/Java [FLL$^+$02] or SPARK. The Whyplatform [Fil03, FM07] is a multi-language, multi-prover platform for program verification, whose architecture is shown in Figure 1.1.

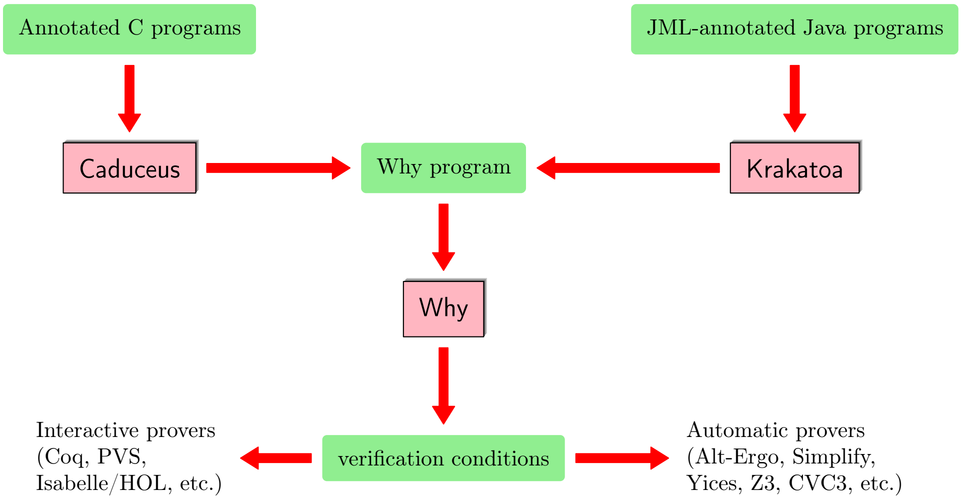


Figure 1.1: Architecture of the Whyplatform

The platform revolves around Why, a verification condition generator (VCG) which takes an annotated Whyprogram as input, analyzes it and returns a set of logical formulae, called *verification conditions* or *proof obligations* (PO). The annotations in the input program express logical properties on the program's behaviour and the tool guarantees that it is sufficient to verify that all the PO are valid in order to check that the logical properties in the program are verified. The Whyplatform can then translate these verification conditions and dispatch them to a variety of provers, interactive or automatic. Whyis used as an intermediate annotated language for verifying programs in mainstream languages, namely C and Java, through separate tools called Caduceus and Krakatoa. These tools perform language-specific analysis, in particular they need to model their respective language's features into the intermediate language. For example, let us consider the following annotated C program:

```
/*@ ensures
  @   \result >= x && \result >= y &&
  @   (\result == x || \result == y)
  @*/
int max(int x, int y) {
  if (x > y) return x; else return y;
}
```

It defines a function `max` which computes the maximum of two integer arguments. The special comments preceding the function are the annotations that describe its behaviour: it states that the result of the function should be greater or equal than both arguments and should be one of the two arguments. Processing this program through the Whyplatform will yield proof obligations corresponding to two branches of the conditional in the function:

$$\begin{array}{lll} \forall xy : int, & x > y \implies & x \geq x \wedge x \geq y \wedge (x = x \vee x = y) \\ \forall xy : int, & x \not> y \implies & y \geq x \wedge y \geq y \wedge (y = x \vee y = y) \end{array}$$

which are trivially true and can be discharged by any automated prover knowledgable about linear arithmetic. This is a very easy example, but such program analysis often yields a great number of proof obligations, many of which are quite easy. Therefore it is very important to be able to discharge these obligations automatically as much as possible. The few very complex obligations, if any, can be inspected by hand or in an interactive prover.

An automated theorem prover used at the back-end of such a program verification plaform needs to be able to deal with quantifiers and with background theories corresponding to the various built-in datatypes of the source languages, typically arithmetic, arrays, tuples, etc. This is why SMT solvers like Z3 [dMB08], Yices [Yic] or CVC [BT07], *i.e.* those which can deal with first-order logic in general, are tools of choice for such a task, and Alt-Ergo was developed specifically for that purpose.

1.2.2 Alt-Ergo

In the context of program verification, we have seen that goals to be proved are formulae of typed first-order logic with quantifiers and interpreted built-in symbols for equalities, integer and/or floating point arithmetic, etc. Sorts naturally arise from the usual datatypes of programming languages (as integers in our example above) and also from the user specifications. Annotations in Why, for instance, are very expressive since they allow user-defined types, symbols, functions and predicates. Whyalso has the particularity of using polymorphic types [Pie02]: polymorphism is very convenient to define and reason about generic data structures like arrays or lists, and also as a means to ensure separation in the memory model used by Caduceus [HM07, TKN07].

Unfortunately, there are only a few SMT solvers under active development which deal with quantifiers, but none of them can handle polymorphic first-order logic natively. In order to use these provers, which are either unsorted or multisorted, the available solutions are to ignore types, trying to guess the monomorphic instances which are needed for a given formula, or using encodings, and all these solutions are quite unsatisfactory [CL07]. Alt-Ergo fully supports polymorphic first-order logic and is therefore particularly well-suited for the Whyplatform.

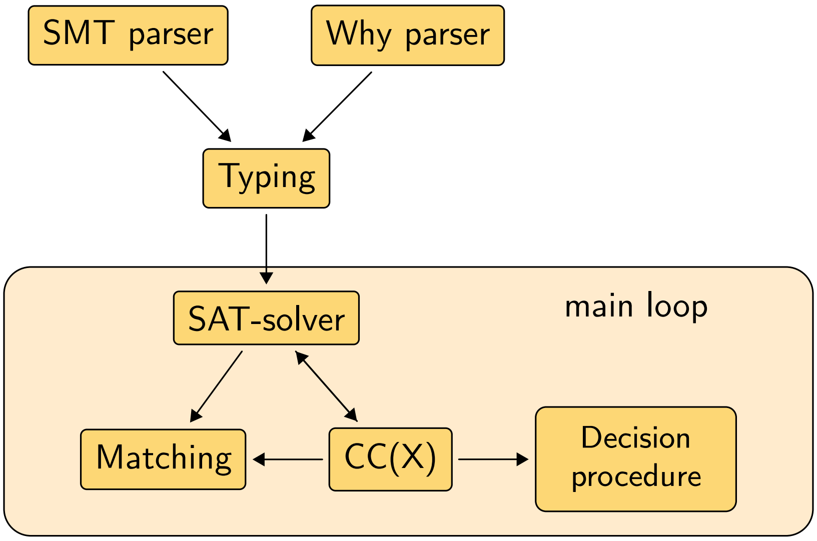


Figure 1.2: Architecture of Alt-Ergo

Alt-Ergo's architecture is shown in Figure 1.2; it is highly modular and this figure schematizes the relation between the different modules. On the front end, Alt-Ergo accepts two different syntaxes: the standard SMT format defined in the SMT-LIB [BST10], and Why's native format. For both formats, an abstract syntax tree in the same internal datatype is produced and then type-checked in polymorphic first-order logic. The formulae then enter the main loop of the prover, which performs the proof search:

SAT-solver. The main part is a home-made SAT-solver with backjumping which deals with the propositional part of the formulae. It also keeps track of the lemmas (*i.e.* universally quantified hypotheses) of the input problem and those that are generated during the execution.

Matching. The matching module is used to find terms that can be used to instantiate the lemmas contained in the SAT solver; it proceeds modulo the equivalence classes in CC(X) and allows the SAT-solver to derive ground sentences from the available lemmas.

CC(X). The CC(X) module handles the ground atoms assumed by the SAT-solver: the SAT-solver sends atoms to this box, which in turn informs the SAT-solver of what atoms are true or false. It combines the theory of equality (*i.e.* uninterpreted symbols) with a theory X via a congruence closure algorithm modulo X.

Decision Procedure. The decision procedure implements the reasoning relative to the background theory X and is used by CC(X) in order to construct equivalence classes modulo X.

Alt-Ergo is implemented in OCaml [Obj] and uses almost exclusively functional data structures, except for the technique of hash-consing, which is used extensively in order to ensure maximal sharing in the data structures and to avoid the blow-up in size due to the conversion to conjunctive normal form [FC06]. Its development was started in 2006 and its main loop is about 5000 lines of code, which is really small for an SMT prover. The small size and modular architecture of Alt-Ergo make it easier to establish that the prover is correct, and this last point has been a motivation (and a concern) from the beginning.

In order to ensure its correctness, we present formalizations of the algorithms at the heart of the most critical modules in Alt-Ergo. Chapter 2 deals with the SAT-solver module and formalizes the DPLL algorithm on which Alt-Ergo's SAT-solver is based, as well as various optimizations. Chapter 3 is devoted to the CC(X) module and describes Alt-Ergo's original congruence closure algorithm modulo a background theory. The requirements that the corresponding decision procedure must verify are also dealt with in Chapter 3. We do not give any formalization for the matching module: this module is indeed not critical for two reasons. First and foremost, the matching mechanism cannot really be incorrect in the sense that any possible lemma instantiations are correct, the matching mechanism is supposed to efficiently determine *useful* instances, and useful instances only, but too many instances can only cause inefficiencies. Second, first-order SMT solvers cannot be complete in general on non-ground formulae, therefore even if the matching mechanism misses all instances, the prover may just be "more" incomplete than ideal, but again it is not a critical error. Now, matching efficiently can be a difficult challenge and advances techniques exist (see [MB07] for instance). Alt-Ergo uses a rather naïve approach but some subtleties arise due to the polymorphic logic, as explained and detailed in [BCCL08].

CHAPTER 2

Formalization of the Propositional Solver

Contents

In this chapter, we present the formalization of the propositional solver at the heart of Alt-Ergo. As explained in the previous chapter, this part of the system is fundamental to any SMT solver and we want to guarantee its correctness. Alt-Ergo's propositional solver is a SAT solver based on the traditional Davis-Putnam-Logemann-Loveland (DPLL) procedure and we start in Section 2.1 by presenting this original DPLL procedure. We also give our own formalization of this algorithm through a set of inference rules and prove the correctness of our inference system. In Section 2.2, we extend this system by successively adding non-chronological backtracking and a mechanism for learning new clauses from conflicts. We then go on to discuss other typical optimizations of state-of-the-art SAT solvers which we have not integrated into our system. In Section 2.3, we show how the SAT solving procedure we have presented can be easily adapted in order to be integrated to an SMT architecture.

2.1 DPLL: A SAT-Solving Procedure

2.1.1 The Satisfiability Problem

The *satisfiability problem* SAT is the problem of deciding whether the variables of a propositional (or boolean) formula can be assigned values in such a way as to make the formula true. A formula for which such an assignment exists is said to be *satisfiable* whereas a formula for which no suitable assignment exists is said to be *unsatisfiable*. Of course, the unsatisfiability problem is dual to the satisfiability one and both are equally difficult. It is a well-known result, and one of the first historical results in complexity theory, that the satisfiability problem is NP-complete [Coo71].

More formally, the formulae of propositional logic are defined as follows. We assume a set $\mathcal{L}$ of propositional variables, also called *atoms*, and a formula is any sentence which can be built using the usual logical connectives and the atoms x in $\mathcal{L}$:

$$F \;:=\; x \mid \neg F \mid F \vee F \mid F \wedge F \mid F \to F \mid F \leftrightarrow F.$$

The SAT problem is traditionally presented with solely the conjunction $\wedge$, disjunction $\vee$ and negation $\neg$ operators, but any functionally complete set of boolean operators can be used without changing the nature of the problem, and we choose here to add the implication and equivalence connectives. A formula reduced to an atom is said to be *atomic*. A *literal* is a variable or the negation of a variable; it is called respectively a *positive* or a *negative* literal. We will write the negation of literals in a slightly different manner than the negation of formulae, namely $\bar{l}$ will denote the negation of literal l. A *clause* is a disjunction of literals and a formula is in *conjunctive normal form* (CNF) if it is a conjunction of clauses, *i.e.* a conjunction of disjunction of literals.

There are several ways to decide the satisfiability or unsatisfiability of a boolean formula. The most naive way is to enumerate all possible assignments and check for each one if the formula becomes true or not; for n variables in the formula, there are 2^n assignments to try. Much better ways have been developed over the years in order to avoid as much as possible the exploration of this exponential search space. Some techniques such as Binary Decision Diagrams [Bry92] can decide satisfiability for any boolean formula, but the majority of modern SAT solvers are variants of the DPLL procedure and only operate on formulae in CNF. Before we deal in detail with the DPLL procedure and some of its variants, let us recall that any propositional formula can be converted into an equivalent formula in CNF, using the well-known De Morgan rules. Therefore requiring that the formulae be in CNF is not a restriction *per se*, and in the remainder of this chapter we shall assume that formulae are in CNF. We will discuss the issue of CNF conversion in great detail later in Chapter 7.

To conclude this introduction, here are several examples:

- the formula $(x_1 \vee (x_3 \wedge x_1)) \leftrightarrow \neg(x_2 \vee x_3)$ is satisfiable, take for instance x_1 false, x_2 true and x_3 false;
- the formula $(x_1 \vee \bar{x}_2) \wedge x_2 \wedge \bar{x}_1$ is in CNF and is unsatisfiable;
- for any positive integer $n \in \mathbb{N}^*$, the formula

$$H_n = \bigwedge_{p=1}^{n} \bigvee_{i=1}^{n-1} x_{pi} \wedge \bigwedge_{i=1}^{n-1} \bigwedge_{p=1}^{n} \bigwedge_{q=1}^{p-1} (\bar{x}_{pi} \vee \bar{x}_{qi})$$

  is unsatisfiable. It expresses the pigeon-hole principle, *i.e.* the fact that n pigeons cannot be put in $n-1$ holes without two pigeons sharing the same hole. The variable x_{pi} stands for "pigeon p is in the hole i", the first part of the conjunct expresses the fact that all pigeons are sheltered, while the second part prevents each hole from containing two pigeons. Note that the formula is in conjunctive normal form. Generic formulae like this one are very useful to benchmark or test a procedure since the parameter can be changed at will; the unsatisfiability of the pigeon-hole formula is notoriously difficult when n grows.

2.1.2 The DPLL Procedure

The Davis-Putnam-Logemann-Loveland procedure was proposed in two seminal papers in the early 1960s in order to solve the satisfiability problem for propositional formulae. In [DP60], Davis and Putnam first proposed a semi-decision procedure for first-order logic which proceeded by enumerating all propositional ground instances of a formula and checking the satisfiability of each of these instances. The satisfiability check was performed by a resolution-based procedure, *i.e.* the instance was simplified repeatedly by using the following rule:

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

which resolves two clauses in a single clause by eliminating a literal appearing positively and negatively. This method led to a worst-case exponential blow-up in the size of the formula and in order to avoid this, Davis, Logemann and Loveland then refined the satisfiability procedure in [DLL62], and what is now known as DPLL.

The DPLL algorithm works on a CNF formula and runs by guessing truth values for literals and the way in which it improves on a naive exhaustive backtracking search is the eager use of the following rules:

Boolean constraints propagation. Once a truth value has been assigned to a literal, the formula can be simplified accordingly: false literals can be deleted from the clauses where they appear, and clauses that contain true literals can be removed from the formula.

Unit propagation. A *unit clause* is a clause which only contains one literal. It is obvious that such a clause can only be satisfied by assigning the adequate value to make that literal true. Such deterministic choices of a truth value for a variable cuts out a large part of the exponential search space and is thus very important for efficiency.

Pure literal elimination. A literal is *pure* if it only appears with the same polarity in the whole formula. A pure literal can be assigned such that all clauses that contain it are true, in other words, it is not constraining the proof search and they can be eliminated systematically. Note that this heuristics is not used anymore because the cost of detecting pure literals exceeds the benefit of eliminating them in modern SAT solvers, therefore we will not include this rule in our presentation.

In this fashion, the algorithm proceeds by successively assigning values to the variables in the formula until one of the following occurs:

- the simplified formula is reduced to the empty conjunction $\emptyset$, which means that the current assignment satisfies the formula; in other words, the formula is satisfiable and the algorithm stops;

- one of the clauses in the problem is empty (also called a *conflict* clause) and cannot be satisfied with the current assignment; in that case the search backtracks and tries another assignment to some variable. If this is not possible, the formula is unsatisfiable.

2.1.3 DPLL as an Inference System

We now present the DPLL procedure formally as a system of inference rules. We use the following conventions for denoting formulas in CNF:

- the order in which literals are presented in a clause is irrelevant, as well as the order of clauses in a CNF formula;

- we write $l \vee C$ for a clause containing the literal l, and we use set-theoretic notation $\{l_1, l_2, l_3\}$ to denote the clause $l_1 \vee l_2 \vee l_3$;

- a formula in CNF is written $C_1, \ldots, C_n$ where the C_i are the different clauses of the formula, we use Δ to range over such conjunctions of clauses.

$$\textsc{Red}\ \frac{\Gamma, l \vdash \Delta, C}{\Gamma, l \vdash \Delta, \bar{l} \vee C} \qquad \textsc{Elim}\ \frac{\Gamma, l \vdash \Delta}{\Gamma, l \vdash \Delta, l \vee C} \qquad \textsc{Assume}\ \frac{\Gamma, l \vdash \Delta}{\Gamma \vdash \Delta, \{l\}}$$

$$\textsc{Conflict}\ \frac{}{\Gamma \vdash \Delta, \emptyset} \qquad \textsc{Split}\ \frac{\Gamma, l \vdash \Delta \qquad \Gamma, \bar{l} \vdash \Delta}{\Gamma \vdash \Delta}$$

Figure 2.1: An abstract presentation of DPLL

Our DPLL formalization is given in Figure 2.1 through five inference rules. The state of the algorithm is described as a *sequent* $\Gamma \vdash \Delta$, where Γ is the set of literals assumed to be true, and Δ is the current formula. These rules must be read bottom-up: the state under the bar is the state *before* the application of the inference rule.

The first two rules perform the boolean constraints propagation as described above. If a literal is supposed to be false (its negation belongs to Γ), it can be eliminated from all clauses (Red); if a clause contains a true literal, the entire clause can be removed (Elim). Assume implements the unit propagation by assuming a literal in a unit clause. Split represents the variable assignment and is the only branching rule: a literal is assumed to be true on the left branch and false on the right branch. Finally, the Conflict rule detects empty clauses and has no premises: it is the only rule that ends the different branches of the proof search.

Starting with some sequent $\Gamma \vdash \Delta$, building a complete derivation with these rules requires each branch to end with an application of the Conflict rule. In other words, if there exists a derivation starting with $\Gamma \vdash \Delta$, there is no satisfying assignment of the variables in Δ such that all the variables in Γ are true (we will say that such an assignment *extends* Γ). Reciprocally, if there is no derivation for $\Gamma \vdash \Delta$, it means that there is a branch that reduces to the empty set of clauses, *i.e.* that there is a way to extend Γ while satisfying Δ. Now, given a formula in CNF Δ, the unsatisfiability of Δ is equivalent to the existence of a derivation for the sequent $\emptyset \vdash \Delta$, *i.e.* starting with an empty partial assignment. We will prove these properties in the next section.

Derivation system vs. Algorithm. The DPLL algorithm and its modern variants are traditionally presented in a procedural manner [DLL62, MMZ$^+$01], that is as deterministic algorithms (for instance as abstracted real code or pseudo-code). We instead chose to present the algorithm as an abstract set of inference rules, in particular we do not specify how and when rules should be applied.

This kind of presentation is more similar to Tinelli's DPLL($\mathcal{T}$) presentation [Tin02]. In our opinion, the main advantage of this approach is that we can manipulate the system without taking the details of a particular implementation into account. Typically, we can prove the correctness of our system regardless of a particular *strategy* of how rules should be applied, and the proofs will apply to any implementation based on the given rules. It would have been possible to add more "constraints" to the system, restricting which strategies are acceptable and which aren't, by using side conditions for some inference rules. For instance, the use of the splitting rule Split could be modified like this:

$$\textsc{Split'}\ \frac{\Gamma, l \vdash \Delta \qquad \Gamma, \bar{l} \vdash \Delta}{\Gamma \vdash \Delta}\ \begin{array}{l} l, \bar{l} \notin \Gamma \\ \exists C \in \Delta, l \in C \end{array}$$

in order to constrain the rule to only be applied to an unassigned literal that actually appears in the problem. There is not much benefit in doing that: these side conditions are not used in the soundness proof of the system, and they just constrain the completeness proof by forbidding some applications of the rules. On the other hand, if one finds a very efficient strategy which, for some reason, occasionally performs a useless split on an already assigned literal, one could not use the system to justify the strategy. Also, if we add some strategy to the rules, how much should we add exactly? It is reasonable to think that the Conflict rule should be used as soon as possible, and that boolean constraint and unit propagation should be performed eagerly otherwise, with Split used as a last resort. This specific strategy could be summarized in regular expression style as:

$$(\textsc{Conflict}?.(\textsc{Red}|\textsc{Elim}|\textsc{Assume})^*.\ \textsc{Split'})^*$$

but it is very restrictive and other reasonable alternatives or refinements exist, such as:

$$(\textsc{Conflict}?.\textsc{Assume}^*.\textsc{Red}^*.\textsc{Split'})^*$$

Because there is no reason to favour one particular strategy, we chose to not add any unnecessary constraint to our system in order to keep it as general as possible. Some strategies may be complete, some may be incomplete[1], but all strategies will be correct as long as the system is sound.

In the second part of this document, when we will provide a formal proof of this system in the Coq proof assistant and then derive some Coq implementations, this approach will be of the utmost importance. It will

[1] When considering one particular strategy, its completeness should always be investigated; the completeness of the system itself is just that there exists at least one complete strategy, as we can see in the proofs page 27.

allow us to prove the abstract system once and for all, and then prove the correctness of the different strategies we will implement with respect to the original system; in particular, this is a very useful way to factorize proofs.

2.1.4 Correctness Proofs for DPLL

We claimed in the previous section that the existence of a derivation of $\emptyset \vdash \Delta$ in the system presented in Figure 2.1 is equivalent to the unsatisfiability of the formula Δ. We will now prove this claim. There are actually two separate parts to prove: the *soundness* of the system is the fact that only unsatisfiable formulas have a derivation, whereas its *completeness* is the fact that a derivation can be found for every unsatisfiable formula[2].

We will actually prove slightly more general results, for any sequent $\Gamma \vdash \Delta$, and the case with an empty assignment Γ will only be a particular instance. We start with the definition of the semantic notion of model.

Definition 2.1.1 (Models). *Given a set of atoms $\mathcal{L}$, an $\mathcal{L}$-*model *$\mathcal{M}$ is a function $\mathcal{L} \mapsto \{\top, \bot\}$ which assigns a truth value (true $\top$, or false $\bot$) to every atom. We write $\mathcal{M}(x)$ for the truth value of atom x in the model $\mathcal{M}$.*

This notion of model is general and we will use it in the next chapter as well. We will write model instead of $\mathcal{L}$-model because the set of atoms is clear from the context. For example, in the remainder of this chapter, $\mathcal{L}$ is the set of propositional variables defined earlier.

We extend the $\mathcal{M}(x)$ notation to literals in a natural way: we write $\mathcal{M}(l)$ for the truth value of the literal l, namely $\mathcal{M}(x)$ if l is a positive literal x, and the negation of $\mathcal{M}(x)$ if l is a negative literal $\bar{x}$.

Definition 2.1.2 (Satisfiability). *A set of clauses Δ is* satisfiable *if and only if there exists a model $\mathcal{M}$ such that for every clause C in Δ, there exists a literal $l \in C$ such that $\mathcal{M}(l) = \top$. In that case, we write $\mathcal{M} \models \Delta$. If there exists no such model $\mathcal{M}$, Δ is said to be* unsatisfiable.

Because we will be dealing with models that are compatible with a partial assignment Γ, we need a more general notion of satisfiability with respect to a partial assignment, which we call compatibility.

Definition 2.1.3 (Submodel). *A set of literals Γ is a* submodel *of a model $\mathcal{M}$, denoted $\Gamma \subseteq \mathcal{M}$, if every literal $l \in \Gamma$ is true in $\mathcal{M}$. We also say that $\mathcal{M}$* completes *Γ.*

Definition 2.1.4 (Compatibility). *A set of literals Γ and a set of clauses Δ are* compatible *if and only if there exists a model $\mathcal{M}$ completing Γ such that $\mathcal{M} \models \Delta$. If there exists no such model, we say that Γ and Δ are* incompatible.

[2] In our choice for naming the two implications soundness and completeness, we are focusing on the unsatisfiability of a formula: if we were taking the dual point of view of satisfiability instead, the soundness and completeness properties would be swapped.

We can now prove the soundness of our DPLL derivation system.

Theorem 2.1.5 (Soundness of DPLL)**.** *Let* Γ *be a set of literals and* Δ *a set of clauses such that the sequent* $\Gamma \vdash \Delta$ *is derivable, then* Γ *and* Δ *are incompatible.*

Proof. We proceed by structural induction on the derivation of $\Gamma \vdash \Delta$ and by case analysis on the first rule applied:

(Conflict) The set of clauses Δ contains the empty clause $\emptyset$, therefore there cannot be a model $\mathcal{M}$ satisfying Δ and Γ and Δ are incompatible.

(Red) By induction hypothesis, there is no model $\mathcal{M}$ such that $\Gamma, l \subseteq \mathcal{M}$ and $\mathcal{M} \models \Delta, C$. Suppose now that there is a model $\mathcal{M}$ completing Γ, l and such that $\mathcal{M} \models \Delta, \bar{l} \vee C$. In particular, $\mathcal{M} \models \Delta$ and there exists a literal k in $\bar{l} \vee C$ such that $\mathcal{M}(k) = \top$. Because $\mathcal{M}$ completes Γ, l, $\mathcal{M}(l) = \top$ and therefore $k \neq l$ and $k \in C$. Thus, $\mathcal{M} \models C$ and $\mathcal{M} \models \Delta, C$, which contradicts the induction hypothesis.

(Elim) Assume there is a model $\mathcal{M}$ completing Γ, l such that $\mathcal{M} \models \Delta, l \vee C$. In particular, $\mathcal{M} \models \Delta$ and therefore Γ, l and Δ are compatible, which contradicts the induction hypothesis.

(Assume) Assume there is a model $\mathcal{M}$ completing Γ such that $\mathcal{M} \models \Delta, \{l\}$. By definition, it must be the case that $\mathcal{M}(l) = \top$. Thus, Γ, l is a submodel of $\mathcal{M}$, and since $\mathcal{M} \models \Delta$, then Γ, l and Δ are compatible, which contradicts the induction hypothesis.

(Split) Assume there is a model $\mathcal{M}$ completing Γ such that $\mathcal{M} \models \Delta$. Depending on whether $\mathcal{M}(l)$ is $\top$ or $\bot$, $\mathcal{M}$ completes Γ, l or $\Gamma, \bar{l}$. In either case, this contradicts the induction hypothesis for one of the two branches. □

Corollary 2.1.6. *Let* Δ *be a formula in conjunctive normal form. If* $\emptyset \vdash \Delta$ *is derivable,* Δ *is unsatisfiable.*

Proof. By Theorem 2.1.5, Δ and the empty assignment are incompatible. Since the empty assignment is a submodel of every model, this means that there are no models of Δ, in other words Δ is unsatsfiable. □

We now turn our attention to establishing the completeness of the derivation system, *i.e.* proving that a derivation can be found for any sequent $\Gamma \vdash \Delta$ as soon as Γ and Δ are incompatible. Such a proof actually contains a strategy: it explicitly shows how to build a derivation for a given incompatible sequent[3]. More precisely, any complete proof search strategy using the rules in Figure 2.1 can be used as a skeleton for a completeness

[3] This claim only holds if the proof is *constructive* of course, which will be the case here and for all our formal proofs in the Coq proof assistant later in Part 2. Our point here is really to stress that there is a strong link between an actual proof search strategy and the completeness proof.

proof, and there are at least as many proofs as strategies. Easier strategies make for easier proofs, therefore we will follow a very naive strategy for constructing our proof.

Definition 2.1.7 (Well-formed assignments)**.** *A set of literals* Γ *is* well-formed *if it does not contain both a literal* l *and its negation* $\bar{l}$*.*

Until now, we had not imposed any restriction on the partial assignment Γ in a sequent. In order to prove completeness of the system however, we need this notion of well-formedness. To see why, notice that according to the definition of a submodel, only a well-formed Γ can be a submodel of some $\mathcal{M}$. Therefore, an ill-formed Γ is incompatible with any sets of clauses Δ, but we cannot expect to be able to build a derivation for such sequents: consider $x_1, \bar{x}_1 \vdash \{x_2\}$ for instance. We will thus only prove completeness for incompatible sequents with a well-formed assignment.

Lemma 2.1.8. *Let* Γ *a well-formed set of literals and* Δ *a set of clauses incompatible with* Γ*, such that all literals appearing in* Δ *are present either positively or negatively in* Γ*. Then, there is a derivation of the sequent* $\Gamma \vdash \Delta$*.*

Proof. Let $\mathcal{M}$ be a model completing Γ. There exists such a model because Γ is well-formed, and it suffices to arbitrarily complete Γ to all variables in $\mathcal{L}$ not appearing in Γ. Now, because Γ is incompatible with Δ, there exists a clause C in Δ such that all literals in C are false in $\mathcal{M}$. Since all variables in Δ are assigned positively or negatively in Γ, this means that for all literal $l \in C$, $\bar{l} \in \Gamma$.

Therefore, we can apply Red as many times as there are literals in the clause C, and we are left with a sequent containing the empty clause, to which point we apply Conflict. We have built a derivation for $\Gamma \vdash \Delta$:

$$\dfrac{\dfrac{\dfrac{\overline{\Gamma \vdash \Delta, \emptyset}\ \textsc{Conflict}}{\vdots}}{\Gamma \vdash \Delta, \{l_2, \dots, l_n\}}\ \textsc{Red}}{\Gamma \vdash \Delta, \{l_1, l_2, \dots, l_n\}}\ \textsc{Red}$$

□

Theorem 2.1.9 (Completeness of DPLL)**.** *Let* Γ *a well-formed set of literals and* Δ *a set of clauses incompatible with* Γ*, then the sequent* $\Gamma \vdash \Delta$ *is derivable.*

Proof. Let $\mathcal{L}'$ be the set of variables appearing in Δ which are not assigned (neither positively nor negatively) in Γ. Let us call these variables $x_1, \dots, x_n$. Starting with $\Gamma \vdash \Delta$, we apply the Split rule as many times as necessary on all the x_i in sequence, until we obtain 2^n branches of the form $\Gamma' \vdash \Delta$ where Γ' ranges from $\Gamma, x_1, \dots, x_n$ to $\Gamma, \bar{x}_1, \dots, \bar{x}_n$.

$$\dfrac{\dfrac{\dfrac{\Gamma, x_1, \dots, x_n \vdash \Delta \qquad \dots}{\vdots}\,\textsc{Split} \quad \dots}{\Gamma, x_1 \vdash \Delta}\,\textsc{Split} \qquad \dfrac{\dfrac{\Gamma, \bar{x}_1, \dots, \bar{x}_n \vdash \Delta \qquad \dots}{\vdots}\,\textsc{Split}}{\Gamma, \bar{x}_1 \vdash \Delta}\,\textsc{Split}}{\Gamma \vdash \Delta}\,\textsc{Split}$$

Let us consider one of the top sequent of the form $\Gamma' \vdash \Delta$. Since Γ' is a superset of Γ and Γ and Δ are incompatible, Γ' and Δ are incompatible. By construction, since Γ is well-formed, so is Γ' since we only split on each variable once. Finally, all the variables that appear in Δ are assigned in Γ', therefore we can apply Lemma 2.1.8 to the sequent $\Gamma' \vdash \Delta$ and find a derivation for this sequent.

By applying the lemma for each sequent at the top, we have built a full derivation for the sequent $\Gamma \vdash \Delta$. □

Corollary 2.1.10. *Let Δ be an unsatisfiable formula in conjunctive normal form. The sequent $\emptyset \vdash \Delta$ is derivable.*

Proof. The empty set of literals $\emptyset$ is well-formed. Therefore, we can apply Theorem 2.1.9 and $\emptyset \vdash \Delta$ is derivable. □

Final remarks. We have established the equivalence between the unsatisfiability of a formula and the existence of a derivation in our system from Figure 2.1. Note that since we based the completeness proof on a very naive strategy, it does not even use the Elim or Assume rule. Indeed, the system formed by the rules Red, Conflict and Split is a correct and complete inference system for the unsatisfiability of formulae in CNF. We added the Elim rule because it may be desirable and it cannot be implemented with the three basic rules; typically, most imperative implementations will not perform elimination of true clauses explicitely during the proof search, but some functional implementations may, in order to simplify the problem during the proof search[4]. The Assume rule can actually be implemented using the other rules:

$$\dfrac{\dfrac{\vdots}{\Gamma, l \vdash \Delta}}{\Gamma \vdash \Delta, \{l\}}\,\textsc{Assume} \quad \Longleftrightarrow \quad \dfrac{\dfrac{\dfrac{\vdots}{\Gamma, l \vdash \Delta}}{\Gamma, l \vdash \Delta, \{l\}}\,\textsc{Elim} \qquad \dfrac{\dfrac{}{\Gamma, \bar{l} \vdash \Delta, \emptyset}\,\textsc{Conflict}}{\Gamma, \bar{l} \vdash \Delta, \{l\}}\,\textsc{Red}}{\Gamma \vdash \Delta, \{l\}}\,\textsc{Split}$$

[4] This will of course be the case for our implementation of this system in Coq, but it is also the case in Alt-Ergo, therefore we need to include this rule to adequately describe Alt-Ergo's SAT solver.

but we add it specifically because of its historical and practical importance.

2.2 Standard DPLL Optimizations

The system described in the previous section remains very naive, and modern SAT solvers, though based on this original procedure, achieve much better results thanks to numerous optimizations [ZM02, Fre95]. Some of these optimizations have a heuristic nature, as they try to pick the most "relevant" decision literals when applying the Split rule for instance. Others, on the contrary, are purely algorithmic and aim at pruning parts of the proof derivation in order to avoid repeating similar reasonings several times.

In this section, we will only focus on the latter kind of enhancements (namely non-chronological backtracking and conflict clause learning), while the others will be briefly addressed at the end of the chapter. In particular, we will show how slight modifications of the system presented so far can lead to sharp improvements.

2.2.1 Non-Chronological Backtracking

Principle. Non-chronological backtracking [SS96], also called backjumping, consists in checking whether a literal introduced in the application of Split was "useful" to the derivation of a conflict in the left branch of this rule. In the case where l wasn't used to establish the conflict, the system can avoid checking the right branch of the rule since the same conflict could be derived in that branch anyway. To illustrate this method, Figure 2.2 shows a run of DPLL on a particular example where variables are encoded as integers:

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\bar{4} \vdash \{\}}}{3 \vdash \{\bar{4}\}, \{4\}}\,\textsc{Assume} \qquad \dfrac{\dfrac{}{\bar{5} \vdash \{\}}}{\bar{3} \vdash \{\bar{5}\}, \{5\}}\,\textsc{Assume}}{2 \vdash \{\bar{3},\bar{4}\}, \{\bar{3},4\}, \{3,5\}, \{3,\bar{5}\}}\,\textsc{Split} \qquad \dfrac{\vdots}{\bar{2} \vdash \ldots}}{1 \vdash \{\bar{3},\bar{4}\}, \{\bar{3},4\}, \{2,3,5\}, \{3,5\}, \{3,\bar{5}\}}\,\textsc{Split} \quad \ldots}{0 \vdash \{\bar{3},\bar{4}\}, \{\bar{1},\bar{3},4\}, \{2,3,5\}, \{3,5\}, \{3,\bar{5}\}}\,\textsc{Split} \quad \ldots}{\emptyset \vdash \{\bar{0},\bar{3},\bar{4}\}, \{\bar{1},\bar{3},4\}, \{2,3,5\}, \{3,5\}, \{3,\bar{5}\}}\,\textsc{Split}
$$

Figure 2.2: An example run of DPLL

Only the rules Assume and Split are actually represented, as we assume that every possible boolean constraint propagation has been realized between each application of these rules. Also, due to space constraints, only the last added literal is shown in Γ. One can notice that in the branch where 2 has been assumed, conflicts arise from the interaction of the literals 3, 4 and 5. The same derivation certainly exists in the right branch where $\bar{2}$ was supposed instead of 2, and the proof search in this branch is therefore done uselessly by DPLL.

Whereas some optimizations are based on heuristics and try to pick the best candidates to split on in order to avoid cases like the one above as much as possible, non-chronological backtracking permits to detect these cases during the proof-search and recover from an earlier unfortunate literal choice.

Changing the rules. In order to take this phenomenon into account, the system has to be able to calculate which literals are responsible for the conflicts in a given branch of a proof derivation. We do this by adding dependency information to literals and clauses in a sequent. To that purpose, we modify our DPLL system from Figure 2.1 in the following manner:

- the context Γ now contains *annotated* literals, *i.e.* pairs $l[\mathcal{A}]$ where l is the literal added to the context and $\mathcal{A}$ is a set of literals (called its *dependencies*) representing those literals who led to the introduction of l in the context;

- each clause in Δ is now also annotated by a set containing the literals that played a role in its reduction;

- finally, sequents are now of the form $\Gamma \vdash \Delta : \mathcal{A}$ where the new element $\mathcal{A}$ is the set of literals used to establish the incompatibility of Γ and Δ. One can also view these sequents as an algorithm taking as input Γ and Δ, and returning a set of literals $\mathcal{A}$. We call $\mathcal{A}$ the *conflict set* of the sequent $\Gamma \vdash \Delta : \mathcal{A}$.

$$\textsc{Red}\ \frac{\Gamma, l[\mathcal{B}] \vdash \Delta, C[\mathcal{B} \cup \mathcal{C}] : \mathcal{A}}{\Gamma, l[\mathcal{B}] \vdash \Delta, \bar{l} \vee C[\mathcal{C}] : \mathcal{A}} \qquad \textsc{Elim}\ \frac{\Gamma, l[\mathcal{B}] \vdash \Delta : \mathcal{A}}{\Gamma, l[\mathcal{B}] \vdash \Delta, l \vee C[\mathcal{C}] : \mathcal{A}}$$

$$\textsc{Conflict}\ \frac{}{\Gamma \vdash \Delta, \emptyset[\mathcal{A}] : \mathcal{A}} \qquad \textsc{Assume}\ \frac{\Gamma, l[\mathcal{B}] \vdash \Delta : \mathcal{A}}{\Gamma \vdash \Delta, l[\mathcal{B}] : \mathcal{A}}$$

$$\textsc{Split}\ \frac{\Gamma, l[l] \vdash \Delta : \mathcal{A} \qquad \Gamma, \bar{l}[\mathcal{A} \setminus l] \vdash \Delta : \mathcal{B}}{\Gamma \vdash \Delta : \mathcal{B}}\ l \in \mathcal{A}$$

$$\mathrm{BJ}\ \frac{\Gamma, l[l] \vdash \Delta : \mathcal{A}}{\Gamma \vdash \Delta : \mathcal{A}}\ l \notin \mathcal{A}$$

Figure 2.3: Inference rules for DPLL with backjumping

The rules corresponding to this mechanism are detailed in Figure 2.3. The five original rules are adapted from the first system, and a new one BJ performs the backjumping. In the rules Red, Elim and Assume, annotations are naturally passed over to clauses and literals: the dependencies of a reduced clause are the dependencies of the literal used to reduce it plus those of the original clause; the dependencies of a unit clause are propagated to the corresponding literal; other dependencies do not change, including the conflict sets. The conflict sets are actually assigned exclusively by the Conflict rule, which now returns, in the right-hand part of the sequent, the set of literals that led to the empty clause. The Split rule is the one which introduces new literals in the mix, and therefore introduces new dependencies: a literal l assumed in a split only depends on itself. The right branch is more involved: the negation $\bar{l}$ depends on the conflict set of the left branch, *i.e.* it is implied by the fact that no satisfying assignment was found in the left branch, with l assumed. The conflict set of the whole split is the conflict set returned by the second branch. Finally, the information brought by the conflict set is used in the BJ rule in order to implement the *backjumping* mechanism, by discarding the right branch of the split when the conflict set does not contain the chosen literal l.

Now, if we take another look at the example of Figure 2.2, the derivation where Split was applied with the literal 2 will now be an application of the new BJ rule. This is represented in Figure 2.4, where $\mathcal{A}$ stands for the set of literals $\{0, 1, 3\}$ et $\mathcal{B} = \mathcal{A} \setminus 3 = \{0, 1\}$. Since $\mathcal{A}$ decorates the left branch and does not contain 2, the right branch will not be explored.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\bar{4}[0,3] \vdash \{\}[\mathcal{A}] : \mathcal{A}}\;\textsc{Conflict}}{3[3] \vdash \{\bar{4}\}[0,3], \{4\}[1,3] : \mathcal{A}}\;\textsc{Assume} \qquad \dfrac{\dfrac{}{\bar{5}[\mathcal{B}] \vdash \{\}[\mathcal{B}] : \mathcal{B}}\;\textsc{Conflict}}{\bar{3}[\mathcal{A}] \vdash \{\bar{5}\}[\mathcal{A}], \{5\}[\mathcal{A}] : \mathcal{B}}\;\textsc{Assume}}{2[2] \vdash \{\bar{3}, \bar{4}\}[0], \{\bar{3}, 4\}[1], \{3, 5\}[], \{3, \bar{5}\}[] : \mathcal{B}}\;\textsc{Split}}{1[1] \vdash \{\bar{3}, \bar{4}\}[0], \{\bar{3}, 4\}[1], \{2, 3, 5\}[], \{3, 5\}[], \{3, \bar{5}\}[] : \mathcal{B}}\;\text{BJ}$$

Figure 2.4: An example run of DPLL with *backjumping*

As a side remark about the inference system, notice that this time we added some side conditions to the rules: the one for BJ is required for the rule to be correct, but the one for Split could be removed safely. There is just no reason to use Split where BJ could be used, therefore we added this second side condition in order to make the two rules mutually exclusive.

2.2.2 Correctness of the Backjumping Mechanism

In order to prove correctness of the inference system with non-chronological backtracking presented in the previous section, we will simulate derivations in this system with derivations in the system without backtracking. This is one advantage of using a very generic presentation in Section 2.1: we can prove further systems as refinements of the first one, ensuring some factorization of the proofs. We start by showing a weakening property for the derivation system without backjumping.

Lemma 2.2.1 (Weakening)**.** *Let Γ, Γ' be two sets of literals such that $\Gamma \subseteq \Gamma'$, and Δ, Δ' two sets of clauses such that $\Delta \subseteq \Delta'$. Then, if $\Gamma \vdash \Delta$ is derivable, so is $\Gamma' \vdash \Delta'$.*

Proof. The proof is really straightforward and proceeds by induction on the derivation of $\Gamma \vdash \Delta$. By analyzing each possible rule, it is easy to check that adding new clauses and literals does not change the applicability of the rules. Note that it is a very natural property if we take the point of view of unsatisfiability instead of derivability: if Δ is incompatible with Γ, then surely adding more clauses to Δ will not help, and neither will adding more constraints to Γ. □

Definition 2.2.2 (Cutting dependencies)**.** *If Γ is a set of annotated literals and $\mathcal{A}$ a set of literals, we write $\Gamma_{|\mathcal{A}}$ for the set of literals which only depend on literals in $\mathcal{A}$:*

$$\Gamma_{|\mathcal{A}} = \{l \mid l[\mathcal{B}] \in \Gamma, \mathcal{B} \subseteq \mathcal{A}\}.$$

Similarly, if Δ is a set of annotated clauses, we write $\Delta_{|\mathcal{A}}$ for the set of clauses only depending on literals in $\mathcal{A}$:

$$\Delta_{|\mathcal{A}} = \{C \mid C[\mathcal{B}] \in \Delta, \mathcal{B} \subseteq \mathcal{A}\}.$$

This cutting operation provides us with a translation from sequents with dependencies to sequents without dependencies. We also write $\Gamma_{|*}$ and $\Delta_{|*}$ for respectively the sets of literals in Γ and clauses in Δ, *i.e.* this is a special case of cutting which just removes all annotations. Our proof is based on a *stability* property: if $\Gamma \vdash \Delta : \mathcal{A}$ is derivable, then $\Gamma_{|\mathcal{A}} \vdash \Delta_{|\mathcal{A}}$ is derivable, which gives a relation between derivations with backjumping and derivations in the original DPLL system. In order to prove the stability, we need an invariant on the annotations in Γ and Δ. To see why, consider the sequent $\emptyset \vdash \Delta, \emptyset[x_1] : \{x_1\}$ where Δ is some set of clauses, it is trivially derivable; if we cut this sequent with the set $\{x_1\}$, the resulting sequent is $\emptyset \vdash \Delta$ and is of course not derivable in general. To avoid such cases, we define well-annotated sequents:

Definition 2.2.3 (Well-annotated)**.** *Let Γ be a set of annotated literals, Δ a set of annotated clauses and $\mathcal{A}$ a set of literals. The sequent $\Gamma \vdash \Delta : \mathcal{A}$ is* well-annotated *if the following holds:*

(i) $\forall k[\mathcal{B}] \in \Gamma,\ \forall l \in \mathcal{B},\ l[l] \in \Gamma$

(ii) $\forall C[\mathcal{B}] \in \Delta,\ \forall l \in \mathcal{B},\ l[l] \in \Gamma$

In other words, all literals l appearing in dependencies in Γ and Δ must be such that $l[l]$ belongs to Γ. We call such literals decision literals.

Note that the definition of well-annotated sequents does not say anything about the conflict set $\mathcal{A}$ and one may wonder if the literals in $\mathcal{A}$ should also be decision literals or not. This is indeed a consequence of the derivability of a well-annotated sequent.

Lemma 2.2.4. *If $\Gamma \vdash \Delta : \mathcal{A}$ is a derivable, well-annotated, sequent, then for all literal $l \in \mathcal{A}$, $l[l]$ belongs to Γ.*

Proof. We proceed by induction on the derivation of $\Gamma \vdash \Delta : \mathcal{A}$ and case analysis on the first rule applied.

(Conflict) When Conflict is used, $\emptyset[\mathcal{A}]$ belongs to Δ, and because the sequent is well-annotated, all literals in $\mathcal{A}$ are decision literals.

(Red) If Red is used, the start of the derivation looks like this:

$$\dfrac{\Gamma, l[\mathcal{B}] \vdash \Delta, C[\mathcal{B} \cup \mathcal{C}] : \mathcal{A}}{\Gamma, l[\mathcal{B}] \vdash \Delta, \bar{l} \vee C[\mathcal{C}] : \mathcal{A}}\ \textsc{Red}$$

It is straightforward to check that the sequent $\Gamma, l[\mathcal{B}] \vdash \Delta, C[\mathcal{B} \cup \mathcal{C}] : \mathcal{A}$ is well-annotated, and therefore we get the result by induction hypothesis.

(Elim) If Elim is used, the start of the derivation looks like this:

$$\dfrac{\Gamma, l[\mathcal{B}] \vdash \Delta : \mathcal{A}}{\Gamma, l[\mathcal{B}] \vdash \Delta, l \vee C[\mathcal{C}] : \mathcal{A}}\ \textsc{Elim}$$

We can apply the induction hypothesis because the premise sequent is well-annotated and we obtain that all literals in $\mathcal{A}$ are decision literals.

(Assume) If Assume is used, the start of the derivation looks like this:

$$\dfrac{\Gamma, l[\mathcal{B}] \vdash \Delta : \mathcal{A}}{\Gamma \vdash \Delta, l[\mathcal{B}] : \mathcal{A}}\ \textsc{Assume}$$

Noting that $\Gamma, l[\mathcal{B}] \vdash \Delta : \mathcal{A}$ is well-annotated, we get by induction hypothesis that any literal k in $\mathcal{A}$ is such that $k[k]$ belongs to $\Gamma, l[\mathcal{B}]$. Because l cannot be in $\mathcal{B}$, this means that $k[k]$ belongs to Γ and we have the needed result.

(BJ) If BJ is used first, the start of the derivation looks like this:

$$\dfrac{\Gamma, l[l] \vdash \Delta : \mathcal{A}}{\Gamma \vdash \Delta : \mathcal{A}}\ \text{BJ}$$

and we have the additional hypothesis that $l \notin \mathcal{A}$. Let $k \in \mathcal{A}$, by induction hypothesis we know that $k[k] \in \Gamma, l[l]$. Since $k \neq l$, we know that $k[k] \in \Gamma$.

(Split) If Split is used first, the start of the derivation looks like this:

$$\dfrac{\Gamma, l[l] \vdash \Delta : \mathcal{B} \qquad \Gamma, \bar{l}[\mathcal{B} \setminus l] \vdash \Delta : \mathcal{A}}{\Gamma \vdash \Delta : \mathcal{A}}\ \textsc{Split}$$

with the additional hypothesis that $l \in \mathcal{B}$. We can apply the induction hypothesis on the left branch, and we obtain that all literals k in $\mathcal{B}$ are such that $k[k] \in \Gamma, l[l]$. Therefore, we know that all literals k in $\mathcal{B} \setminus l$ are such that $k[k]$ belongs to Γ, and thus that the sequent $\Gamma, \bar{l}[\mathcal{B} \setminus l] \vdash \Delta : \mathcal{A}$ is well-annotated. Hence, we can apply the induction hypothesis to this sequent and we get that all literals in $\mathcal{A}$ are decision literals. □

We now have enough to express the stability theorem.

Theorem 2.2.5 (Stability). *Let* Γ *be a set of annotated literals,* Δ *a set of annotated clauses and* $\mathcal{A}$ *a set of literals such that* $\Gamma \vdash \Delta : \mathcal{A}$ *is a derivable, well-annotated, sequent. Then, there exists a derivation of* $\Gamma_{|\mathcal{A}} \vdash \Delta_{|\mathcal{A}}$.

Proof. First, note that the statement mixes two different kind of derivations. Because the syntactic nature of the sequent usually suffices to distinguish between derivations in DPLL with and without backjumping, we do not explicitely state which system we are using unless it is absolutely necessary.

The proof proceeds by a structural induction on the derivation of $\Gamma \vdash \Delta : \mathcal{A}$ and by case analysis on the first rule applied. Note that when applying the induction hypothesis, we will not explicitely prove that the premise sequents are well-annotated, the arguments are exactly the same as in the above lemma.

(Conflict) When Conflict is used, the empty set belongs to Δ and is annotated with the conflict set $\mathcal{A}$. Therefore, it also belongs to $\Delta_{|\mathcal{A}}$ and we can apply Conflict to find a derivation of $\Gamma_{|\mathcal{A}} \vdash \Delta_{|\mathcal{A}}$.

(Red) If Red is used, the start of the derivation looks like this:

$$\dfrac{\Gamma, l[\mathcal{B}] \vdash \Delta, C[\mathcal{B} \cup \mathcal{C}] : \mathcal{A}}{\Gamma, l[\mathcal{B}] \vdash \Delta, \bar{l} \vee C[\mathcal{C}] : \mathcal{A}}\ \textsc{Red}$$

There are two cases to consider:

- if $\mathcal{B} \cup \mathcal{C} \subseteq \mathcal{A}$, then both $\mathcal{B}$ and $\mathcal{C}$ are subsets of $\mathcal{A}$, and thus l, C and $l \vee C$ are not removed when cutting the sequent. The induction hypothesis gives us a derivation of $\Gamma_{|\mathcal{A}}, l \vdash \Delta_{|\mathcal{A}}, C$ and by applying Red we obtain a suitable derivation :

$$\dfrac{\Gamma_{|\mathcal{A}}, l \vdash \Delta_{|\mathcal{A}}, C}{\Gamma_{|\mathcal{A}}, l \vdash \Delta_{|\mathcal{A}}, \bar{l} \vee C}\ \textsc{Red}$$

- if $\mathcal{B} \cup \mathcal{C} \not\subseteq \mathcal{A}$, then the reduced clause C is cut from the top sequent, and the induction hypothesis gives us a derivation of $(\Gamma, l[\mathcal{B}])_{|\mathcal{A}} \vdash \Delta_{|\mathcal{A}}$. Since $\Delta_{|\mathcal{A}}$ is included in $(\Delta, C[\mathcal{C}])_{|\mathcal{A}}$, applying the weakening lemma to the induction hypothesis gives us a derivation for $(\Gamma, l[\mathcal{B}])_{|\mathcal{A}} \vdash (\Delta, C[\mathcal{C}])_{|\mathcal{A}}$.

(Elim) If Elim is used, the start of the derivation looks like this:

$$\frac{\Gamma, l[\mathcal{B}] \vdash \Delta : \mathcal{A}}{\Gamma, l[\mathcal{B}] \vdash \Delta, l \vee C[\mathcal{C}] : \mathcal{A}}\ \textsc{Elim}$$

The induction hypothesis gives us a derivation of $(\Gamma, l[\mathcal{B}])_{|\mathcal{A}} \vdash \Delta_{|\mathcal{A}}$. By weakening, we have a derivation of $(\Gamma, l[\mathcal{B}])_{|\mathcal{A}} \vdash (\Delta, l \vee C[\mathcal{C}])_{|\mathcal{A}}$.

(Assume) If Assume is used, the start of the derivation looks like this:

$$\frac{\Gamma, l[\mathcal{B}] \vdash \Delta : \mathcal{A}}{\Gamma \vdash \Delta, l[\mathcal{B}] : \mathcal{A}}\ \textsc{Assume}$$

The unit clause and the literal l have the same dependencies $\mathcal{B}$ and therefore they are both cut or both kept. In the first case, we need a derivation of $\Gamma_{|\mathcal{A}} \vdash \Delta_{|\mathcal{A}}$ and it is simply the induction hypothesis; in the latter case, we can apply Assume to the cut sequent to retrieve the induction hypothesis:

$$\frac{\Gamma_{|\mathcal{A}}, l \vdash \Delta_{|\mathcal{A}}}{\Gamma_{|\mathcal{A}} \vdash \Delta_{|\mathcal{A}}, \{l\}}\ \textsc{Assume}$$

(BJ) If BJ is used first, the start of the derivation looks like this:

$$\frac{\Gamma, l[l] \vdash \Delta : \mathcal{A}}{\Gamma \vdash \Delta : \mathcal{A}}\ \mathrm{BJ}$$

and we have the additional hypothesis that $l \notin \mathcal{A}$. After cutting, the top and bottom sequents are the same and therefore we just need to apply the induction hypothesis.

(Split) If Split is used first, the start of the derivation looks like this:

$$\frac{\Gamma, l[l] \vdash \Delta : \mathcal{B} \qquad \Gamma, \bar{l}[\mathcal{B} \setminus l] \vdash \Delta : \mathcal{A}}{\Gamma \vdash \Delta : \mathcal{A}}\ \textsc{Split}$$

with the additional hypothesis that $l \in \mathcal{B}$. The induction hypothesis on the left branch gives us a derivation for the sequent $\Gamma_{|\mathcal{B}}, l \vdash \Delta_{|\mathcal{B}}$. There are two cases to consider depending on what happens on the right branch:

- if $\mathcal{B} \setminus l \not\subseteq \mathcal{A}$, the induction hypothesis on the right branch gives us a derivation of $\Gamma_{|\mathcal{A}} \vdash \Delta_{|\mathcal{A}}$, which is exactly what we want;

- if $\mathcal{B} \setminus l \subseteq \mathcal{A}$, the induction hypothesis on the right branch gives us a derivation of $\Gamma_{|\mathcal{A}}, \bar{l} \vdash \Delta_{|\mathcal{A}}$. We would like to apply the SPLIT rule, in other words we would like to establish that $\Gamma_{|\mathcal{A}}, l \vdash \Delta_{|\mathcal{A}}$ is derivable. Unfortunately, the induction hypothesis on the left branch gives a slightly different derivation, namely $\Gamma_{|\mathcal{B}}, l \vdash \Delta_{|\mathcal{B}}$. We prove $\Gamma_{|\mathcal{A}}, l \vdash \Delta_{|\mathcal{A}}$ from $\Gamma_{|\mathcal{B}}, l \vdash \Delta_{|\mathcal{B}}$ by using the weakening property, *i.e.* we prove that $\Gamma_{|\mathcal{B}} \subseteq \Gamma_{|\mathcal{A}}$ and $\Delta_{|\mathcal{B}} \subseteq \Delta_{|\mathcal{A}}$. Let $k \in \Gamma_{|\mathcal{B}}$, there is $k[\mathcal{C}] \in \Gamma$ such that $\mathcal{C} \subseteq \mathcal{B}$, we want to prove that $k \in \Gamma_{|\mathcal{A}}$, *i.e.* that $\mathcal{C} \subseteq \mathcal{A}$. Since $\mathcal{B} \setminus l \subseteq \mathcal{A}$, it is equivalent with the fact that $l \notin \mathcal{C}$. Because the sequent on the right branch is well annotated, we know that all literals in $\mathcal{C}$ are decision literals in $\Gamma, \bar{l}[\mathcal{B} \setminus l]$, and therefore that l does not belong to $\mathcal{C}$. This proves that $\Gamma_{|\mathcal{B}} \subseteq \Gamma_{|\mathcal{A}}$ and by the same argument, we can prove that $\Delta_{|\mathcal{B}} \subseteq \Delta_{|\mathcal{A}}$. Therefore we have a derivation of $\Gamma_{|\mathcal{A}}, l \vdash \Delta_{|\mathcal{A}}$ and by using the rule SPLIT, we can build the derivation we want:

$$\frac{\Gamma_{|\mathcal{A}}, l \vdash \Delta_{|\mathcal{A}} \qquad \Gamma_{|\mathcal{A}}, \bar{l} \vdash \Delta_{|\mathcal{A}}}{\Gamma_{|\mathcal{A}} \vdash \Delta_{|\mathcal{A}}} \textsc{Split}$$

□

Theorem 2.2.6 (Soundness)**.** *Let Γ be a set of annotated literals, Δ a set of annotated clauses and $\mathcal{A}$ a conflict set such that $\Gamma \vdash \Delta : \mathcal{A}$ is well-annotated and derivable. Then, $\Gamma_{|*}$ and $\Delta_{|*}$ are incompatible.*

Proof. By the stability lemma, the sequent $\Gamma_{|\mathcal{A}} \vdash \Delta_{|\mathcal{A}}$ is derivable, and by weakening, this means that the sequent $\Gamma_{|*} \vdash \Delta_{|*}$ is derivable as well. We simply conclude by applying theorem 2.1.5, *i.e.* the soundness of the derivation system without backjumping. □

We finish these proofs by stating the particular case of soundness for an empty assignment, which is the starting point of a procedure based on these rules:

Corollary 2.2.7. *Let Δ be a formula in CNF. Let us annotate all clauses in Δ with an empty set of dependencies. Then, if $\emptyset \vdash \Delta : \mathcal{A}$ is derivable for some $\mathcal{A}$, Δ is unsatisfiable.*

Proof. By Theorem 2.2.6. □

The completeness of the system with backjumping can be easily obtained by showing that any derivation of a sequent $\Gamma_{|*} \vdash \Delta_{|*}$ also leads to a derivation of $\Gamma \vdash \Delta : \mathcal{A}$ for some $\mathcal{A}$.

Lemma 2.2.8. *Let Γ be a set of annotated literals and Δ a set of annotated clauses. If $\Gamma_{|*} \vdash \Delta_{|*}$ is derivable, then there exists some conflict set $\mathcal{A}$ such that $\Gamma \vdash \Delta : \mathcal{A}$ is derivable.*

Proof. The proof proceeds by structural induction on the derivation of $\Gamma_{|*} \vdash \Delta_{|*}$ and by case analysis on the first rule used. Intuitively, each rule can be mimied by the corresponding rule in the system with backjumping, simply by adding the dependencies and the conflict sets. For instance, if the rule used was Conflict, it has the following form:

$$\frac{}{\Gamma_{|*} \vdash \Delta'_{|*}, \emptyset}\ \textsc{Conflict}$$

where $\Delta = \Delta', \emptyset[\mathcal{A}]$ for some set of dependencies $\mathcal{A}$. Thus, the empty clause appears in Δ annotated with $\mathcal{A}$ and therefore the following derivation is possible:

$$\frac{}{\Gamma \vdash \Delta', \emptyset[\mathcal{A}] : \mathcal{A}}\ \textsc{Conflict}$$

If instead the rule used was Red, the derivation has the following form:

$$\frac{\Gamma'_{|*}, l \vdash \Delta'_{|*}, C}{\Gamma'_{|*}, l \vdash \Delta'_{|*}, \bar{l} \vee C}\ \textsc{Red}$$

where $\Gamma = \Gamma', l[\mathcal{B}]$ and $\Delta = \Delta', \bar{l} \vee C[\mathcal{C}]$ for some sets of dependencies $\mathcal{B}$ and $\mathcal{C}$. By applying the induction hypothesis to the sets Γ and $\Delta', C[\mathcal{B} \cup \mathcal{C}]$, we know that there exists $\mathcal{A}$ such that $\Gamma, l[\mathcal{B}] \vdash \Delta, C[\mathcal{B} \cup \mathcal{C}] : \mathcal{A}$ is derivable. Hence, we can build the following derivation:

$$\frac{\Gamma', l[\mathcal{B}] \vdash \Delta', C[\mathcal{B} \cup \mathcal{C}] : \mathcal{A}}{\Gamma', l[\mathcal{B}] \vdash \Delta', \bar{l} \vee C[\mathcal{C}] : \mathcal{A}}\ \textsc{Red}$$

i.e. a derivation of $\Gamma \vdash \Delta : \mathcal{A}$. The rules Elim and Assume can be treated similarly without any difficulty. The only interesting rule is the Split rule. Suppose the derivation of $\Gamma_{|*} \vdash \Delta_{|*}$ starts with the Split rule:

$$\frac{\Gamma_{|*}, l \vdash \Delta_{|*} \qquad \Gamma_{|*}, \bar{l} \vdash \Delta_{|*}}{\Gamma_{|*} \vdash \Delta_{|*}}\ \textsc{Split}$$

We can apply the induction hypothesis to $\Gamma, l[l]$, the clauses Δ and the derivation in the first branch and we get a derivation of $\Gamma, l[l] \vdash \Delta : \mathcal{A}$ for some conflict set $\mathcal{A}$. Now if $l \in \mathcal{A}$, we apply the induction hypothesis to $\Gamma, \bar{l}[\mathcal{A} \setminus l]$, the clauses Δ and the second branch of the above derivation: we get a derivation of $\Gamma, \bar{l}[\mathcal{A} \setminus l] \vdash \Delta : \mathcal{B}$ for some set $\mathcal{B}$, and we apply the split rule in order to get a derivation of $\Gamma \vdash \Delta : \mathcal{B}$.

$$\frac{\Gamma, l[l] \vdash \Delta : \mathcal{A} \qquad \Gamma, \bar{l}[\mathcal{A} \setminus l] \vdash \Delta : \mathcal{B}}{\Gamma \vdash \Delta : \mathcal{B}}\ \textsc{Split}$$

If on the contrary l does not belong to $\mathcal{A}$, we can simply apply the BJ rule and take advantage of the backjumping mechanism:

$$\frac{\Gamma, l[l] \vdash \Delta : \mathcal{A}}{\Gamma \vdash \Delta : \mathcal{A}}\ \text{BJ}$$

□

Using this lemma and the completeness of the DPLL system, we get the completeness of the system with backjumping.

Theorem 2.2.9 (Completeness)**.** *Let Δ be a formula in CNF with all clauses annotated with empty dependencies. Then, if $\Delta_{|*}$ is unsatisfiable, there exists $\mathcal{A}$ such that $\emptyset \vdash \Delta : \mathcal{A}$ is derivable.*

Proof. By the completeness of the derivation system without backjumping (Corollary 2.1.10), there exists a derivation of $\emptyset \vdash \Delta_{|*}$. We conclude by Lemma 2.2.8. □

2.2.3 Conflict-Driven Learning

Principle. Adding non-chronological backtracking has allowed our system to avoid exploring some parts of the tree by analyzing the way earlier conflicts were found, but it still does not take advantage of all the information that is available. To realize this issue, consider the situation schematized in Figure 2.5.

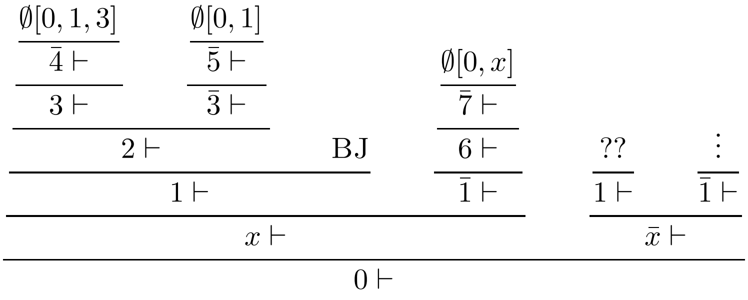


Figure 2.5: Example showing the insufficiency of *backjumping*

This figure shows the skeleton of a proof derivation (in the system of Figure 2.3) which is somehow similar to the one shown in Figure 2.2. Only the decision literals and the conflict sets at the leaves of the tree are represented. The difference between the derivation of Figure 2.4 and this one is that, in the latter, a new literal x has been introduced by Split between the introductions of 0 and 1. Now, 0 and 1 were precisely the two literals which were leading to the conflicts, for after backjumping on 2, the dependencies associated to the sequent were $\mathcal{B} = \{0, 1\}$.

In particular, this means that assuming both 0 and 1 will also lead to a conflict in the branch marked with a question mark. Nevertheless, non-chronological backtracking cannot help pruning this part of the tree since the dependency information $\{0, 1\}$ is lost as soon as the algorithm returns "below" a node where one of these literals was introduced. In our case, when returning from the branch where 1 was assumed, the new set of dependencies is $\{0, x\}$ and cannot anyway mention the literal 1: when backtracking to the point where x was introduced, we lost the information that 0 and 1 do not go along so well, and we can't exploit it in the remaining part of the proof search.

Changing the rules, again. In order to solve this problem, a possible solution is to keep, along with the current set of dependencies, a set of clauses called *conflict clauses* representing all the clauses that have already been "learnt" during the proof search. On our example, we have learnt that 0 and 1 imply the empty clause, since $\emptyset$ is annotated with $[0, 1]$. This is the information we keep in the conflict set on the right-hand side of the sequent. More generally, every time we have a clause C annotated with literals $l_1, \ldots, l_n$, this means that $l_1 \wedge \ldots \wedge l_n$ implies C. The only such relation that the system with backjumping remembers is the one that is stored in the conflict set. When the solver returns to the branch on x, it will lose this information so we want to make sure that it remembers that $0 \wedge 1$ implies a conflict. Because 1 does not appear in the assignment anymore, it cannot appear in the dependencies; in other words, when removing 1 from the context, we want to change $\emptyset[0, 1]$ to $\{\bar{1}\}[0]$. More generally, we will consider that conflict clauses are annotated clauses and define an operation called "shifting", noted Shift_l, used to remove a literal l from a clause's annotations and move it to the clause itself. Shift_l is a function applied to a set of annotated clauses:

$$\begin{aligned}
\mathit{Shift}_l\,(\emptyset) &= \emptyset \\
\mathit{Shift}_l\,(\{C[\mathcal{A}, l]\} \cup \mathbb{A}) &= \{\bar{l} \vee C[\mathcal{A}]\} \cup \mathit{Shift}_l\,(\mathbb{A}) \\
\mathit{Shift}_l\,(\{C[\mathcal{A}]\} \cup \mathbb{A}) &= \{C[\mathcal{A}]\} \cup \mathit{Shift}_l\,(\mathbb{A}) \text{ if } l \notin \mathcal{A}
\end{aligned}$$

Sequents are now of the form $\Gamma \vdash \Delta : \mathcal{A}, \mathbb{A}$ where the new element $\mathbb{A}$ is the set of conflict clauses. The rules are very similar to the one in Figure 2.3 and only add the treatment of conflict clauses; they are presented in Figure 2.6. Conflict clauses originate from the dependencies found in Conflict, and Split takes care of adding $\bar{l}[\mathcal{A}\backslash l]$ to the set of conflict clauses when the set of dependencies $\mathcal{A}$ contains l. The clauses are maintained by all other rules, with the exception of Split and BJ, which apply Shift_l to all conflict clauses found in the left branch, as suggested in the discussion above. Finally, these clauses are used in the right branch of the Split rule in order to accelerate the search of a conflict in this branch. Actually, among the clauses in $\mathit{Shift}_l(\mathbb{A})$, those who contain $\bar{l}$ will be eliminated by Bcp, but the other ones will possibly help in quickly establishing a conflict.

$$\textsc{Red}\ \frac{\Gamma, l[\mathcal{B}] \vdash \Delta, C[\mathcal{B} \cup \mathcal{C}] : \mathcal{A}, \mathbb{A}}{\Gamma, l[\mathcal{B}] \vdash \Delta, \bar{l} \vee C[\mathcal{C}] : \mathcal{A}, \mathbb{A}} \qquad \textsc{Elim}\ \frac{\Gamma, l[\mathcal{B}] \vdash \Delta : \mathcal{A}, \mathbb{A}}{\Gamma, l[\mathcal{B}] \vdash \Delta, l \vee C[\mathcal{C}] : \mathcal{A}, \mathbb{A}}$$

$$\textsc{Conflict}\ \frac{}{\Gamma \vdash \Delta, \emptyset[\mathcal{A}] : \mathcal{A}, \emptyset} \qquad \textsc{Assume}\ \frac{\Gamma, l[\mathcal{B}] \vdash \Delta : \mathcal{A}, \mathbb{A}}{\Gamma \vdash \Delta, l[\mathcal{B}] : \mathcal{A}, \mathbb{A}}$$

$$\textsc{Split}\ \frac{\Gamma, l[l] \vdash \Delta : \mathcal{A}, \mathbb{A} \qquad \Gamma, \bar{l}[\mathcal{A} \setminus l] \vdash \Delta, \mathit{Shift}_l\,(\mathbb{A}) : \mathcal{B}, \mathbb{B}}{\Gamma \vdash \Delta : \mathcal{B}, \mathit{Shift}_l\,(\mathbb{A}) \cup \{\bar{l}[\mathcal{A} \setminus l]\} \cup \mathbb{B}}\ l \in \mathcal{A}$$

$$\text{BJ}\ \frac{\Gamma, l[l] \vdash \Delta : \mathcal{A}, \mathbb{A}}{\Gamma \vdash \Delta : \mathcal{A}, \mathit{Shift}_l\,(\mathbb{A})}\ l \notin \mathcal{A}$$

Figure 2.6: Inference rules for DPLL with conflict clause learning

Correctness proofs. Unlike the previous derivation system presented in Section 2.2.1, where we were able to derive the soundness and completeness proofs of the backjumping mechanism from the proofs of the basic DPLL system, this is not easily feasible for the system with conflict-driven clause learning. The intuition behind this is that the first two systems had the same proof derivations, with some parts being cut off by the backjumping rule. With learning, clauses in a part of the tree can come from a conflict obtained in a totally different part of the tree. Moreover, they cannot be justified "locally" in the proof derivation, but are justified by the initial problem at the root of the tree. Note that the completeness property can still be established exactly like our first two systems, by ignoring the learnt clauses and just building the naive derivation similar similar to what we did in Section 2.2.2. The soundness proof is quite long and is given in Appendix A. The soundness theorem is stated as follows:

Theorem 2.2.10 (Soundness)**.** *Let Δ be a formula in CNF, with all clauses annotated with empty dependencies. Then, if there exists a conflict set $\mathcal{A}$ and some set of conflict clauses $\mathbb{A}$ such that $\emptyset \vdash \Delta : \mathcal{A}, \mathbb{A}$ is derivable, Δ is unsatisfiable.*

Proof. See Appendix A. □

2.2.4 Backjumping vs. Learning

We have just presented two different mechanisms for optimizing the DPLL procedure: backjumping and conflict-driven clause learning. They are traditionally presented together as a single mechanism because the clause learning mechanism supersedes the backjumping mechanism: as we explained above, a conflict set $\mathcal{A}$ is indeed just a special case of conflict clause $\emptyset[\mathcal{A}]$. Nevertheless, these two mechanisms are fundamentally different and it is one of the specificities of our approach to present them separately.

To understand the important difference between backjumping and learning, we can look at the impact of each of these optimizations in comparison to the basic DPLL. Backjumping enhances the proof search by trimming the search tree and each use of of the backjumping rule strictly simplifies the search. In constrast, conflict-driven clause learning proceeds by adding new clauses to the problem which hopefully allow the system to derive conflicts faster and accelerates the search. The cost of adding backjumping is simply the cost of adding dependency analysis and is easily compensated by the gain in efficiency due to the use of the BJ rule. On the contrary, the cost of adding backtracking encompasses both dependency analysis and the fact that the number of clauses in the problem can augment dramatically (up to 2^n clauses where n is the number of variables in the problem). In practice, there is no guarantee that learning will actually improve the efficiency of the system on a given problem, it might well slow down the prover: this has a lot to do with how well the implementation can cope with a great number of clauses.

Therefore, the decision of whether or not clause learning should be used in a given system depends on the context in which it is implemented and used. In the context of software verification of programs annotated by humans, as explained in Section 1.2.1, the propositional complexity of proof obligations derives mainly from the propositional complexity of annotations and from the annotated functions' structure, and is therefore quite limited. Such formulae do not require state-of-the-art optimizations and Alt-Ergo's SAT-solver relies on the DPLL procedure with backjumping (because it is always profitable) but without clause learning, because its effect is too unpredictable and having too many useless clauses can be very detrimental to the solver[5].

2.3 From SAT to SMT

So far in this chapter, we have described a system to decide the unsatisfiability of propositional formulae, but as explained in Chapter 1, when it is

[5] For instance, as explained in Section 1.2.2, the matching mechanism relies on the terms available in the current clauses to derives new instances, therefore having too many clauses can yield too many instances.

used at the heart of an SMT solver like Alt-Ergo, the propositional atoms are not variables but are typically terms with some interpreted function symbols. This means that not all assignments are acceptable and we discuss in this section how the rules seen so far can be easily adapted to account for satisfiability modulo theories.

Definition 2.3.1. *A* theory *is a set of models. If $\mathcal{T}$ is a theory, we call its elements $\mathcal{T}$-models. We say that a formula F is $\mathcal{T}$-satisfiable (resp. $\mathcal{T}$-unsatisfiable) if there exists (resp. there does not exist) a $\mathcal{T}$-model satisfying the formula F.*

As with models, the definition of a theory is quite general and we will reuse it in the next chapter. Let us look at an example first. Let $\mathcal{S}$ be a set of symbols, and assume the set of propositional atoms $\mathcal{L}$ is the set of equations between elements of $\mathcal{S}$, *i.e.* $\mathcal{L} = \mathcal{S} \times \mathcal{S}$. The sequent $\emptyset \vdash \{s_1 = s_2\}, \{s_2 = s_3\}, \{s_3 \neq s_1\}$ is not derivable and therefore the set of clauses is satisfiable, but any satisfying assignment maps $s_1 = s_2$ to $\top$, $s_2 = s_3$ to $\top$ and $s_3 = s_1$ to $\bot$, which does not respect the “meaning”of equality. We are actually only interested in the models which verify the following properties:

(i) $\forall x \in \mathcal{S}, \mathcal{M} \models x = x$

(ii) $\forall xy \in \mathcal{S}, \mathcal{M} \models x = y \rightarrow \mathcal{M} \models y = x$

(iii) $\forall xyz \in \mathcal{S}, \mathcal{M} \models x = y \rightarrow \mathcal{M} \models y = z \rightarrow \mathcal{M} \models x = z$

and the set of models which verify these properties is an example of a theory[6] (which can be seen as the theory of equality on $\mathcal{S}$). If we only consider the models in this theory, the set of clauses above is unsatisfiable. To account for this, we change the nature of partial assignments from a set of literals to an abstract structure of *environment*.

Definition 2.3.2. *An* environment Γ *is a structure which supports the two following operations:*

(i) the assumption *of a literal l, which is a partial operation; we write Γ, l when assuming l in Γ succeeds;*

(ii) querying *whether a literal l is true in the environment or not; we write $\Gamma \downarrow l$ to denote that the literal l is true in Γ.*

These two operations correspond to the two manners in which we use the partial assignment in the different systems from Figures 2.1, 2.3 and 2.6. We

[6] In traditional model theory, where theories are defined as sets of formulae (or *axioms*), these properties (i), (ii), (ii) could be seen as the axioms defining this theory. The presentation as sets of models is equivalent and can be more natural when dealing with SMT: the SMT solver does not know about the axioms of a theory $\mathcal{T}$, but tries to construct a $\mathcal{T}$-model for an input formula.

assume literals, *i.e.* add them to the environment, when we assign a value to some literal, and we *query* the partial assignment for the state of a literal, *i.e.* check whether a literal or its negation has already been assigned a value. The assumption of a literal is a partial operation because the assumed literal can be inconsistent with the current environment. It is then straightforward to rewrite the rules with an environment in the left-hand side of the sequent instead of a set of literals, for instance Figure 2.7 show how we adapt the basic DPLL.

$$\textsc{Red}\ \frac{\Gamma \vdash \Delta, C}{\Gamma \vdash \Delta, \bar{l} \vee C}\ \Gamma \downarrow l \qquad\qquad \textsc{Elim}\ \frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, l \vee C}\ \Gamma \downarrow l$$

$$\textsc{Assume}\ \frac{\Gamma, l \vdash \Delta}{\Gamma \vdash \Delta, \{l\}} \qquad\qquad \textsc{Conflict}\ \frac{}{\Gamma \vdash \Delta, \emptyset}$$

$$\textsc{Split}\ \frac{\Gamma, l \vdash \Delta \qquad \Gamma, \bar{l} \vdash \Delta}{\Gamma \vdash \Delta}$$

Figure 2.7: DPLL with an environment

The Red and Elim rules now have a side condition to express that the query in the environment must return true, and other rules do not change syntactically. Note that because assumption must succeed, the rules Assume and Split, although they do not change syntactically, are slightly more constrained than in the original presentation: in particular, it is now impossible to build a derivation where Γ is not well-formed in the sense of Definition 2.1.7 page 29, because a new literal cannot be assumed if it contradicts a formerly assumed literal.

In an environment for some theory $\mathcal{T}$, a literal can be true even if it (or its negation) has not been assigned explicitely, because it can be a consequence in $\mathcal{T}$ of the literals explicitely assumed in the environment. Conversely, a literal can be false if it is inconsistent with the literals already assumed in the environment. We write $|\Gamma|$ for the set of literals explicitely assumed in environment Γ. For instance, an environment for the theory of equality above will typically perform the equivalence closure of the equalities assumed and the query of $x_3 = x_1$ in the environment $x_1 = x_2, x_2 = x_3$ will return true. More generally, in order to be suitable to decide satisfiability in some theory $\mathcal{T}$, an environment will have to verify some properties:

- for the system to be sound, the environment must be sound with respect to the theory $\mathcal{T}$, *i.e.* that if $\Gamma \downarrow l$, l must be a consequence of all the assumed literals:

$$\forall l, \Gamma \downarrow l \quad \rightarrow \quad \forall \mathcal{M} \in \mathcal{T}, \mathcal{M} \models |\Gamma| \rightarrow \mathcal{M}(l) = \top$$

- for the system to be complete, the environment must be complete with respect to the theory $\mathcal{T}$, in symbols:

$$\forall l, \forall \mathcal{M} \in \mathcal{T}, \mathcal{M} \models |\Gamma| \rightarrow \mathcal{M}(l) = \top \quad \rightarrow \quad \Gamma \downarrow l$$

With such invariants, the correctness proofs are straightforward to adapt and we can prove that the derivability of the sequent $\emptyset \vdash \Delta$ is equivalent to the $\mathcal{T}$-satisfiability of the formula Δ. We will not detail how to precisely adapt the correctness proofs of our DPLL derivation system here, the soundness proof will be detailed formally later in Chapter 8.

An equivalent characterization of the existence of an environment structure suitable for a theory $\mathcal{T}$ is the existence of a decision procedure P for the $\mathcal{T}$-satisfiability of conjunctions of literals. Indeed, if such a procedure P exists, the following operations define a suitable environment:

- an environment is simply a set of literals;
- the adding operation Γ, l simply adds l to the set Γ and uses P to check that the new set of literals is not unsatisfiable; if it is, the assumption does not succeed;
- to perform a query of l in Γ, use the procedure P to test the satisfiability of the set of literals $\Gamma, \neg l$: if it is unsatisfiable, then l is a consequence of the literals of Γ and $\Gamma \downarrow l$ holds; otherwise it does not hold.

The latter characterization is slightly more convenient. For instance, this method can be applied to the trivial theory of all models in order to retrieve the DPLL procedure for pure propositional logic: the procedure P simply checks whether both a literal and its negation are present in the conjunction.

SMT with dependencies. A natural question is whether it is also possible to adapt the backjumping and clause learning mechanisms to this SMT architecture. In order to do so, environments must be able to deal with annotations:

- the assumption of a literal should also take its dependencies as input: we write $\Gamma, l[\mathcal{B}]$ for the assumption of l with dependencies $\mathcal{B}$ in Γ;
- when a query for a literal l succeeds, the environment should also return a set of dependencies which justify that l is indeed true, which we write $\Gamma \downarrow l[\mathcal{B}]$.

The adaptations of the rules is then straightforward, and the rules with backjumping are given in Figure 2.8 for instance. In practice, adding dependency analysis to an environment based on a satisfiability procedure for some

$$\textsc{Red}\ \frac{\Gamma \vdash \Delta, C[\mathcal{B} \cup \mathcal{C}] : \mathcal{A}}{\Gamma \vdash \Delta, \bar{l} \vee C[\mathcal{C}] : \mathcal{A}}\ \Gamma \downarrow l[\mathcal{B}] \qquad \textsc{Elim}\ \frac{\Gamma \vdash \Delta : \mathcal{A}}{\Gamma \vdash \Delta, l \vee C[\mathcal{C}] : \mathcal{A}}\ \Gamma \downarrow l[\mathcal{B}]$$

$$\textsc{Conflict}\ \frac{}{\Gamma \vdash \Delta, \emptyset[\mathcal{A}] : \mathcal{A}} \qquad \textsc{Assume}\ \frac{\Gamma, l[\mathcal{B}] \vdash \Delta : \mathcal{A}}{\Gamma \vdash \Delta, l[\mathcal{B}] : \mathcal{A}}$$

$$\textsc{Split}\ \frac{\Gamma, l[l] \vdash \Delta : \mathcal{A} \qquad \Gamma, \bar{l}[\mathcal{A} \setminus l] \vdash \Delta : \mathcal{B}}{\Gamma \vdash \Delta : \mathcal{B}}\ l \in \mathcal{A}$$

$$\text{BJ}\ \frac{\Gamma, l[l] \vdash \Delta : \mathcal{A}}{\Gamma \vdash \Delta : \mathcal{A}}\ l \notin \mathcal{A}$$

Figure 2.8: DPLL with backjumping and an environment

theory can be very challenging since the decision procedure must be instrumented in order to find the (possibly smallest) sets of literals which justify its results. Examples of interesting results in this area of proof-producing decision procedures are [NO05, dMRS05, RRT07]. Alt-Ergo implements a coarse but effective dependency analysis in order to use backjumping, but we have not implemented a proof producing procedure in Coq, and consequently our Coq implementation does not use backjumping but stays with the basic DPLL procedure (see Chapter 6).

2.4 Discussion

In this chapter, we have described the propositional solver at the heart of Alt-Ergo as a system of inference rules. This algorithm is based on the DPLL SAT solving procedure and we showed how to enhance the basic system with a non-chronological backtracking mechanism, as well as conflict-driven clause learning. These two mechanisms are ubiquitous in modern implementations of DPLL-based SAT solvers.

2.4.1 State-of-the-Art SAT Solvers

Even with the backjumping and learning mechanisms, our DPLL system does not qualify as a modern, state-of-the-art, SAT solver. Such SAT solvers typically include a great number of different optimizations and heuristics and can deal efficiently with industrial problems containing hundreds of thousands of propositional variables (cf. [sat]).

We do not claim to achieve the sheer performance of these systems or to be able to simulate their behaviour with our rule-based systems. Instead, our motivation is to apply this formalization to Alt-Ergo's SAT solver in order to accurately describe it, and Alt-Ergo uses a relatively basic SAT solving procedure. In fact, Alt-Ergo is based on the system with backjumping but does not use clause learning. Therefore, the rules we have presented so far are sufficient to describe Alt-Ergo's kernel. More generally speaking, they are also a solid foundation on which to implement a SAT solver, and this is what motivated us into adding conflict-driven clause learning. We now take a quick look at other typical optimizations that are present in modern SAT solvers, and discuss what kind of challenge they would represent.

Variable assignment. When applying the splitting rule, *i.e.* when arbitrarily trying to assign a variable either boolean value, some variable must be chosen. As we emphasized at the start of Section 2.2.1, the performance of the SAT solver is very sensitive to that particular choice. Different strategies have been designed in order to pick variables in a sensible way: some choose randomly, some try to maximize some measure (e.g. the number of times a variable appears in a problem), some are much more involved and perform very well in a great variety of problems, like the Variable State Independent Decaying Sum (VSIDS) decision heuristic used in Chaff and presented in [MMZ$^+$01], which is used in conjunction with conflict-based clause learning. The important thing about variable assignment choices is that any strategy is correct and therefore there is almost nothing to prove about it: soundness is granted, and completeness is guaranteed as long as the strategy tries every variable sooner or later. This is why there is no reason to mention such a strategy in our formalization; on the contrary, our rules gives full freedom as far as the choice of a literal is concerned.

Two-watched literals. A SAT solver spends most of its time performing boolean constraint propagation and trying to apply the unit rule. Modern optimizations often employ a variant of a technique called *two-watched literals* [MMZ$^+$01, Zha97], which consists in keeping a handle on two non-false literals per clause at all time and only performing simplifications on these literals, until it is not possible to find two such literals, which means the corresponding clause is unitary or empty. Such a technique is very important in practice but in our opinion, it is not a feature that requires a formal description and proof, but rather it is a matter of implementation.

Restarts. Modern SAT solvers also rely on some way of *restarting* the proof search at regular intervals, in order to explore the search space more efficiently. A typical restart strategy for our system with clause learning would be to stop search at some point and restart with an empty assignment, but retaining some of the clauses learnt so far. That way, the search starts in a “fresh” state, but with more information than the first time, hopefully avoiding bad variable choices in the future. Restarts cannot be simulated with our rules, because this would require the initial state (or formula) to be stored in the sequent, but once again the critical point about restarts is whether the learnt clauses are correct, not the restart mechanism itself and we decided not to adapt our rules to include restarts. Incidentally, there exists a broad range of restart strategies, see [Hua07] for instance.

Conflict Analysis. In our inference rules, we described the conflicts found during the proof search thanks to the literals in annotations. These literals were what is known as *decision literals*, *i.e.* literals which were added through a Split (or BJ) rule. There exists other ways to describe a conflict, and conflict analyses have been thoroughly studied because their effect on the performance of a SAT solver is very significant (see [SS96, ZMM01] for instance). In particular, [ZMM01] describes conflicts using an implication graph between assigned literals and their empirical results show that literals which have some property in this graph (known as UIP, for Unique Implication Point) lead to better conflict clauses than decision literals for instance. Our system could be adapted to any conflict analysis by keeping an implication graph instead of the simple annotations we have, but we did not formalize that modification. In particular, such analyses are only useful to improve the effect of conflict-driven clause learning, in the sense that it generates conflict clauses which are maybe more pertinent, but it does not improve on backjumping since a system with backjumping always backtracks to the lowest possible literal in the proof tree. Note also that unlike the preceding optimizations, the conflict analysis is critical and requires an accurate formalization, since unsound clauses could be derived by an inappropriate strategy.

2.4.2 Conclusion

The work closest to this approach originated with [Tin02] and is Nieuwenhuis, Oliveras and Tinelli's formalization of DPLL [NOT04]. Their system is based on transition rules and describes a version of DPLL where side conditions are expressed in an abstract manner. This allows them to encompass at once a broad range of common optimizations and to easily reason about the correctness of such techniques. In particular, unlike ours, their presentation does not differentiate backjumping from clause learning, and we explained above why we think that it is important to separate these two mechanisms. The main downside of their approach is that its abstraction makes it harder to derive a trustworthy implementation from the formalization. On the contrary, the gap between our system and the actual implementation is really small: in particular, our rules describe exactly how to calculate dependencies and conflict clauses.

This is also a downside, of course, since our system is much less expressive than the one in [NOT04]. Nevertheless, as we emphasized several times in this chapter, we tried to remain as generic as possible. We do not have any strategy to select decision literals, but adding heuristics to pick literals in the Split rule would not impact our correctness proof. In our Coq implementation in Chapter 7, we will demonstrate how our system is independent of the actual representation of formulas, and how to take advantage of this to use techniques of efficient CNF conversion, such as maximal sharing of sub-formulas using *hash-consing*.

CHAPTER 3

CC(X): Congruence Closure Modulo Solvable Theories

Contents

In Chapter 2, we presented how to handle propositional logic with the DPLL procedure and its modern variants. We also hinted at the fact that the same procedure could be used to deal with formulae where literals have some interpretation, *i.e.* to decide the satisfiability of a formula modulo some *theory*, as long as one is able to provide an environment which decides entailment in this theory. This chapter is devoted to show how to build such an environment for a certain class of theories. More precisely, we will show how to build an environment for the combination of the theory of equality and any theory X which verifies certains properties, among which the existence of a particular function called a *solver*. This algorithm is parameterized by this theory X and will be called $\mathsf{CC(X)}$. In Section 3.1, we will describe the problem of solving the theory of equality modulo another theory and present the two main existing methods: the Nelson-Oppen combination method on one hand, and Shostak's algorithm on the other. In Section 3.2, we present our algorithm $\mathsf{CC(X)}$ for the congruence closure modulo a theory X and show how it differs and improves on the two existing methods. We then prove that the algorithm is sound and complete for suitable theories. Finally, we extend $\mathsf{CC(X)}$ in Section 3.4 in order to deal with disequations instead of just equalities.

3.1 Combining Equality and Other Theories

3.1.1 Preliminaries

In order to define the theories we are interested in and to build their literals, we need a term algebra. In the following, we assume a large, fixed, set Σ of symbols and we suppose that each symbol comes with a non-negative integer called its *arity*. We define the set of (ground) terms $\mathcal{T}$ inductively as the smallest set which is closed for the following operation: if $f \in \Sigma$ is a symbol of arity n and $t_1, \ldots, t_n$ are some terms in $\mathcal{T}$, then $f(t_1, \ldots, t_n)$ belongs to $\mathcal{T}$. In particular, our terms are untyped since we do not consider any typing constraint for the construction of terms. The set of propositional atoms that we are interested in in the remaining of this chapter is the set $\mathcal{L}$ of all equalities $u = v$ for some $u, v \in \mathcal{T}$.

Definition 3.1.1. *The* theory of equality*, written $\mathcal{E}$, is defined by the fact that $=$ is a congruence relation,* i.e. *by the following axioms:*

(Reflexivity) $\forall t \in \mathcal{T}, t = t$

(Symmetry) $\forall t, u \in \mathcal{T}, t = u \implies u = t$

(Transitivity) $\forall t, u, v \in \mathcal{T}, t = u \implies u = v \implies t = v$

(Congruence) $\forall f \in \Sigma, \forall t_1, u_1 \ldots, t_n, u_n \in \mathcal{T}$,
$(\forall i, t_i = u_i) \implies f(t_1, \ldots, t_n) = f(u_1, \ldots, u_n)$

The theory $\mathcal{E}$ (in the sense of Definition 2.3.1 page 44) is the set of models for which these axioms hold.

The theory $\mathcal{E}$ is often called EUF, for Equality on Uninterpreted Functions, and is obviously essential to deduction and verification systems. For instance, problem divisions in the SMT competition [BST10] include a category devoted to this theory (QF_UF) and other categories deal with the combination of EUF and other theories such as bitvectors (QF_AUFBV), difference logic (QF_UFIDL), arrays (QF_AUFLIA), etc.

Given a set of equalities E, the set of all equalities implied by the combination of E and the theory of equality is the *congruence closure* of E. If we consider E as a relation over terms, its congruence closure is also a relation over terms and we write it $=_E$. Formally, this means that given two terms u and v:

$$u =_E v \quad \iff \quad \forall M \in \mathcal{E}, M \models E \implies M \models u = v.$$

For example, if f and a are some symbols in Σ, and E is the set of equations $\{a = f(f(f(a))), a = f(f(f(f(f(a)))))\}$, then $a =_E f(a)$.

The task of computing the congruence closure of a finite set of equations has been addressed separately by Downey, Sethi and Tarjan [DST80], Nelson and Oppen [NO80] and Shostak [Sho78] thirty years ago. Their procedures all achieved worst-case complexity of $\mathcal{O}(n\,log(n))$ and are formulated on relations over vertices of a graph representing the terms of the problem.

In a solver like Alt-Ergo, we are not only dealing with uninterpreted functions, but some symbols have a standard interpretation which should be accounted for. The meaning of these symbols is given by one or several theories. For instance, the following formula[1]:

$$k = 0 \implies s - a = a \implies f(s + k, 2 + 3) = f(a + a, 5) \tag{3.1}$$

is valid in the union of $\mathcal{E}$ and the theory of linear arithmetic on rationals but not in $\mathcal{E}$ alone. To decide the satisfiability of such formulae, the previous algorithms for computing a congruence closure are not sufficient and one needs a procedure for congruence closure modulo a theory.

3.1.2 The Nelson-Oppen Combination Method

The most widely used method to combine the theory of equality and other theories was proposed by Nelson and Oppen [NO79]. Their method is actually more general in that it gives an algorithm to combine decision procedures for different theories into a decision procedure for the union of these theories.

Let $\mathcal{T}_1, \ldots, \mathcal{T}_n$ be n theories such that there exists satisfiability procedures $P_1, \ldots, P_n$ for each of these theories. Among other things, the Nelson-Oppen method requires that theories use disjoint sets of interpreted symbols, say $\Sigma_1, \ldots, \Sigma_n$. The algorithm proceeds by splitting a formula Φ into n subformulae $\Phi_1, \ldots, \Phi_n$ where Φ_i only uses *abstraction variables*[2] and symbols in Σ_i. It then dispatches each subformula Φ_i to the corresponding decision procedure P_i. The different decision procedure only cooperate indirectly by exchanging informations about the variables of the problem through the dispatcher. This architecture is summarized in Figure 3.1.

The procedure can be summarized by the following steps:

[1] as is usually done, we write binary arithmetic symbols in infix notation.

[2] These abstraction variables are not strictly speaking variables but can also be considered as fresh constants. They are traditionally called variables in the literature about Nelson-Oppen combination.

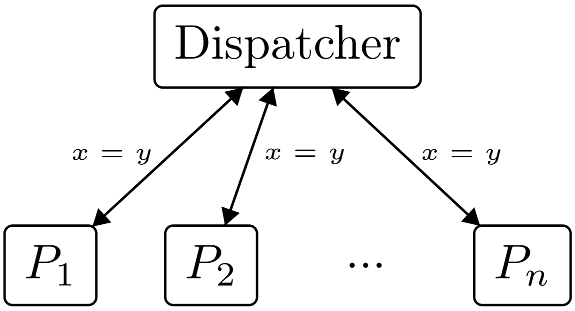


Figure 3.1: Architecture of the Nelson-Oppen combination

1. (Variable abstraction) Split the formula Φ in a conjunction of pure formulae $\Phi_1, \ldots, \Phi_n$ which only share abstraction variables.
2. (Dispatching) Send each formula Φ_i to the corresponding procedure P_i. If any returns *unsatisfiable* then the whole formula is *unsatisfiable.*
3. (Equality propagation) Gather all the equalities between variables which have been found by the P_i during the previous step, and propagate them to all theories. Return to step 2.
4. (End) When no contradiction has been found by any decision procedure, and no more equalities between variables are found, Φ is *satisfiable.*

One can see that a key point in the method originally presented by Nelson and Oppen is that the P_i must return the equalities between variables they find when they are run. Although critical for efficiency, this requirement is not theoretically mandatory. In a later presentation of this algorithm [TH96], Tinelli and Harandi proposed a non-deterministic version of the algorithm where the correct partition between the variables (what they call an *arrangement* of the variables) is simply guessed. Since there are a finite number of arrangements, an algorithm could proceed by trying all of them.

It is clear that, provided that the unsatisfiability procedures $P_1, \ldots, P_n$ are correct, the formula is truly unsatisfiable when the procedure says so. The converse however is not true in general: when all subproblems are satisfiable in their respective theories, the conjunction is not necessarily satisfiable in the union of theories. To be sound and complete, the Nelson-Oppen procedure thus requires strong properties on the theories:

- The theories must be *convex*: this means that a conjunction of literals should not entail a disjunction of equalities without entailing at least one of the disjuncts. This restriction ensures that there is no need for “splits” since the combination scheme cannot dispatch disjunctions of equalities. Although many theories of interest are indeed convex, the convexity requirement is the biggest obstacle in practice (for instance, the theories of arrays or linear arithmetic with inequalities are non-convex).

- The theories must be *stably infinite*. This condition was formalized in [TH96] and not in the original paper, and it expresses the fact that all satisfiable formulae admit models with infinite cardinality. In particular, this excludes theories that specify finite types, e.g. booleans.

This general combination scheme has been applied to the issue of combining congruence closure and other theories. For instance we can use this scheme with the theory $\mathcal{E}$ and linear rational arithmetic to solve our example formula 3.1. The variable abstraction yields the following conjunction of literals:

$$\begin{array}{lcl} \Phi_1 & : & f(z_1, z_2) \neq f(z_3, z_4) \\ \Phi_2 & : & k = 0 \wedge s - a = a \wedge s + k = z_1 \wedge 2 + 3 = z_2 \wedge a + a = z_3 \wedge 5 = z_4 \end{array}$$

Φ_1 and Φ_2 are both satisfiable in their theory, but when analyzing Φ_2 the decision procedure for linear arithmetic reports that $s = z_1 = z_3$ and $z_2 = z_4$. After propagation in Φ_1, the congruence closure algorithm reports that Φ_1 is unsatisfiable, and so is the original formula.

The Nelson-Oppen architecture or variants thereof are used in deduction systems such as the Stanford Pascal Verifier [LGvH$^+$79], Yices [Yic], Simplify [DNS05], CVC3 [BT07] and Z3 [dMB08]. It is widely used because of its generic nature and because it applies to many theories of interests.

3.1.3 The Shostak Combination Method

The Nelson-Oppen combination method is not devoted to the combination of equality and another theory, but it is more generic than that. One consequence is that $\mathcal{E}$ and the other theories play a totally symmetric role. In [Sho84] Shostak proposed an alternative which is specifically devoted to combining equality with another theory. Shostak's procedure only works on equational theories which have two special functions: a *canonizer* and a *solver*. The canonizer is used to transform a term into a normal form with respect to the theory, while the solver takes an equation and "solves" it into an equivalent *substitution*, *i.e.* a list of equalities of the form $x = t$ where x is a variable in the original equation. We call these theories *Shostak theories*.

Congruence closure algorithms in [DST80, NO80, Sho78] proceed by computing a canonical form for all terms, in particular using a *union-find* structure; Shostak's procedure does essentially the same thing but using the canonizer and the solver of the theory $\mathcal{T}$ in order to build a canonical form modulo $\mathcal{T}$. The canonizer is used to normalize terms modulo $\mathcal{T}$ and the solver is used to propagate all the consequences of an equation into the union-find structure. For instance, let us look at example 3.1 again. The theory of linear rational arithmetic is a Shostak theory: the normal form for this theory is a sum of ordered monomials with rational coefficients, and the solver can be implemented with standard Gauss elimination. Solving the first two equalities $k = 0$ and $s - a = a$ yields the substitutions $k \mapsto 0$ and $s \mapsto 2 * a$. After substitution, the last equality becomes $f(2*a+0, 2+3) = f(a+a, 5)$, and after canonization, it becomes $f(2*a, 5) = f(2*a, 5)$ which is obviously true.

The original presentation of Shostak's procedure suffered multiple flaws, in particular it is neither complete nor terminating. The procedure was revamped and corrected first partially in [CLS96] by Cyrluk, Lincoln and Shankar, and then completely in [RS01] by Rueß and Shankar. The formalization and the proofs are much more involved than in the original presentation, and Ford and Shankar later published [FS02] a formal proof of the presentation in [RS01], done in PVS [PVS]. Proofs about combinations of theories are notoriously difficult and error-prone, and such verified proofs are rare and valuable.

3.1.4 Motivations

The restriction imposed on Shostak theories, *i.e.* the properties that must hold for the canonizers and solvers, make them a smaller class than the class of theories suitable for Nelson-Oppen. However, when it applies, Shostak's combination scheme improves on Nelson-Oppen's architecture. Indeed, Nelson-Oppen does not treat $\mathcal{E}$ in a special way, and all decision procedures must perform their own equality propagation (typically using union-find) which is costly. Shostak's procedure regroups equality reasoning in a single congruence closure algorithm, and factors all theory reasoning in the canonizer and solver functions. We schematize this situation in Figure 3.2. Thanks to this better interaction with the traditional congruence reasoning, the Shostak procedure seems to perform better than the Nelson-Oppen procedure: comparing these two algorithms in practice is not easy because they are usually part of bigger systems, but an informal comparison reported in [CLS96] suggests a difference of about an order of magnitude. Shostak's algorithm is also simpler to implement than Nelson-Oppen because there is no exchange of equalities between the different procedures.

Although some of the disadvantages of the Nelson-Oppen scheme are avoided by Shostak, his procedure has its own shortcomings. In particular, the underlying decision procedures in Nelson-Oppen can be implemented in any possible way, whereas a Shostak theory revolves around the term data structure: it must be implemented with a term canonizer and a solver which returns term substitutions. Altogether, canonizing, solving and substituting are actions which require a lot of term manipulations and traversals. For

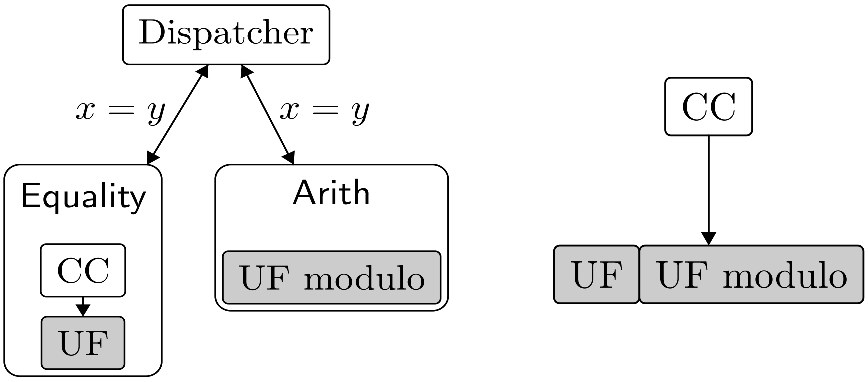


Figure 3.2: Schematic comparison of the Nelson-Oppen (left) and Shostak architecture (right).

most theories, this does not represent the way one would implement such functions, and more efficient representations of the terms could be more convenient. For instance, the term data structure is not adapted to linear arithmetic manipulation, and solving and substituting can be implemented much more efficiently with polynoms, *i.e.* an ad-hoc data structure. This is the motivation for the algorithm we present in the remainder of this chapter: a mechanism for congruence closure modulo a theory inspired by Shostak but where abstract data representation is possible and encouraged.

3.2 CC(X): Congruence Closure Modulo X

In this section, we present the algorithm CC(X) (for congruence closure modulo X) which combines the theory $\mathcal{E}$ with an arbitrary built-in theory X. This algorithm uses *abstract values* as representatives allowing efficient data structures for the implementation of solvers. We first define the class of theories which are amenable for our algorithm, which we call *solvable theories*, and then present CC(X) as a set of inference rules whose description is detailed enough to truly reflect the actual implementation of the combination mechanism in Alt-Ergo.

3.2.1 Solvable Theories

While solvers and canonizers of Shostak theories operate on terms directly, solvable theories work on a certain set $\mathcal{R}$, whose elements are called *semantic values*. The main particularity is that we don't know the exact structure of these values, only that they are somehow constructed from interpreted and uninterpreted (or foreign) parts. To compensate, we dispose of two functions $[\cdot]$ and *leaves* which are reminiscent of the variable abstraction mechanism found in the Nelson-Oppen method. The function $[\cdot]$, which we also call *make*, constructs a semantic value from a term; *leaves* extracts its uninterpreted parts in an abstract form.

Definition 3.2.1. *We call a* solvable theory X *a tuple* $(\Sigma_{\mathsf{X}}, \mathcal{R}, X)$*, where* $\Sigma_{\mathsf{X}} \subseteq \Sigma$ *is the set of function symbols interpreted by* X*,* $\mathcal{R}$ *is the set of semantic values and* X *is an equational theory. In particular,* X *is a relation over terms and therefore* $=_X \subseteq \mathcal{T} \times \mathcal{T}$ *denotes the congruence closure of the relation* X*. Additionally, a solvable theory* X *has the following properties:*

(i) There is a function $[\cdot] : T(\Sigma) \to \mathcal{R}$ *to construct a semantic value out of a term. For any set* E *of equations between terms we write* $[E]$ *for the set* $\{[x] = [y] \mid x = y \in E\}$ *and similary for sequences of equations.*

(ii) There is a function leaves $: \mathcal{R} \to \mathcal{P}^*_f(\mathcal{R})$*, where the elements of* $\mathcal{P}^*_f(\mathcal{R})$ *are finite non-empty sets of semantic values. Intuitively, its role is to return the set of maximal uninterpreted values a given semantic value consists of*[3]*. Its behaviour is left undefined for now, but is constrained by axioms given below.*

(iii) There is a special value $\mathbf{1} \in \mathcal{R}$ *which we will use to denote the leaves of pure terms' representatives.*

(iv) There is a function subst $: \mathcal{R} \times \mathcal{R} \times \mathcal{R} \to \mathcal{R}$*. Instead of subst*$(p, P, r)$ *we write* $r\,\{p \mapsto P\}$*. The pair* (p, P) *is called a* substitution *and subst*(p, P, r) *is the* application *of the substitution* (p, P) *to* r*.*

(v) There is a function solve $: \mathcal{R} \times \mathcal{R} \to (\mathcal{R} \times \mathcal{R})^{\top,\bot}$ *which takes an equation between semantic values and returns either* $\top$*,* $\bot$ *or an equation between semantic values (which must be seen as a substitution). When the result is* $\top$ *(resp.* $\bot$*), we say that the equation is solved (resp. unsolvable).*

In the remaining of this paper, we simply call *theory* a solvable theory. An example of such a theory is given in Section 3.2.3. We write $\equiv$ the equality in the set of semantic values, and it should not be confused with term equality $=$.

In the following, for any set S, we write S^* the set of finite sequences of elements of S. If $s \in S^*$ is such a sequence and a is an element of S, we write $a; s$ for the sequence obtained by prepending a to s. The empty sequence is denoted $\bullet$. We will use sequences instead of sets in many places in order to be able to describe the incrementality of our algorithm; we will however use sequences as sets implicitly in places where order does not matter. As we will often talk about successive substitutions, we define an auxiliary function that does just that:

[3] Therefore, the *leaves* correspond to what are called the *solvables* part of an interpreted term in [RS01].

Definition 3.2.2. *We define the partial function iter* : $(\mathcal{R} \times \mathcal{R})^* \times \mathcal{R} \to \mathcal{R}^\perp$ *that applies solve and subst successively in the following way:*

$$\begin{array}{rcll}
iter(\bullet, r) & = & r & \\
iter((r_1, r_2); S,\, r_3) & = & r_3' \,\{p \mapsto P\} & \text{where} \left\{ \begin{array}{rcl} r_i' & = & iter(S, r_i) \\ solve(r_1', r_2') & = & (p, P) \end{array} \right. \\
iter((r_1, r_2); S,\, r_3) & = & r_3' & \text{where} \left\{ \begin{array}{rcl} r_i' & = & iter(S, r_i) \\ solve(r_1', r_2') & = & \top \end{array} \right. \\
iter((r_1, r_2); S,\, r_3) & = & \perp & \text{where} \left\{ \begin{array}{rcl} r_i' & = & iter(S, r_i) \\ solve(r_1', r_2') & = & \perp \end{array} \right. \\
iter((r_1, r_2); S,\, r_3) & = & \perp & \text{otherwise.}
\end{array}$$

Thus, $iter(S, r)$ successively solves all equations in S, applying the resulting substitution (if any) to r and to the remaining equations along the way. It returns $\perp$ if and only if one of the equations was unsolvable. We now use this notion of iterated substitution to define entailment in the set $\mathcal{R}$ of semantic values.

Definition 3.2.3. *Let E be a sequence of equations between semantic values, and r_1, r_2 two semantic values. We write $E \models_X r_1 = r_2$ to denote that the sequence of equations E entail that $r_1 = r_2$, and we define it in the following way:*

$$E \models_X r_1 = r_2 \overset{def}{\iff} iter(E, r_1) \equiv iter(E, r_2).$$

In particular, if $iter(E, r_1)$ and $iter(E, r_2)$ are $\perp$, $E \models_X r_1 = r_2$ holds.

In addition to definition 3.2.1, a theory X must fulfill the following axioms:

Axiom 3.2.4. *For any $r_1, r_2, p, P \in \mathcal{R}$,*

(i) $solve(r_1, r_2) = (p, P) \Rightarrow r_1 \{p \mapsto P\} \equiv r_2 \{p \mapsto P\}$

(i') $solve(r_1, r_2) = (p, P) \Rightarrow p \notin leaves(P)$

(ii) $solve(r_1, r_2) = \top \iff r_1 \equiv r_2$

(iii) $solve(r_1, r_2) = \perp \iff \forall (p, P), r_1 \{p \mapsto P\} \not\equiv r_2 \{p \mapsto P\}.$

Axiom 3.2.5. *For any set of term equations E and pair of terms u, v,*

$$[E] \models_X [u] = [v] \Rightarrow u =_{E,X} v,$$

where $=_{E,X}$ is the congruence closure of the equational theory defined by $E \cup X$.

Axiom 3.2.6. *For any $r, p, P \in \mathcal{R}$ such that $r \not\equiv r\{p \mapsto P\}$,*

(i) $p \in leaves(r)$

(ii) $leaves(r\{p \mapsto P\}) = (leaves(r) \setminus \{p\}) \cup leaves(P)$.

Axiom 3.2.7. *For any pure term t,* i.e. *a term built exclusively from symbols in Σ_{X}, we have $leaves([t]) = \{\mathbf{1}\}$.*

Let us explain this a little bit. First of all, as we will see in section 3.2.2, the algorithm establishes and maintains equivalence classes over semantic values. Every equivalence class is labeled by an element of the set $\mathcal{R}$; a function $\Delta : \mathcal{R} \to \mathcal{R}$ is maintained which for each value returns its current label. Together with the [.] function, this function can be used to maintain equivalence classes over terms. The function *solve* is capable of solving an equation between two elements of $\mathcal{R}$, that is, it transforms an equation $r_1 = r_2$ for $r_1, r_2 \in \mathcal{R}$ into the substitution (p, P), with $p, P \in \mathcal{R}$, where the value p is now isolated. Axiom 3.2.4-(i) makes sure that such a substitution renders equal the two semantic values r_1 and r_2, which are at the *origin* of this substitution, and 3.2.4-(i') enforces that the left-hand side of a substitution cannot appear in the right-hand side[4]. The last two items in Axiom 3.2.4 are straightforward and cover the cases where the equation is either solved or unsolvable. We have equipped $\mathcal{R}$ with a notion of *implication* of equalities, the relation $\models_X$. Axiom 3.2.5 just states that, if some equations $[E]$ between semantic values imply an equation $[u] = [v]$, then $u =_{E,X} v$, that is, an equality on the theory side implies an equality between corresponding terms. Axiom 3.2.6 ensures that substituting p with P in a semantic value only has effect if p is a leaf of this value, and that the new leaves after the substitution are leaves coming from P. In this respect, leaves can be understood as the "variables" of a semantic value. Finally, the last axiom describes why we introduced a special value $\mathbf{1}$ in $\mathcal{R}$: representatives of pure terms do not have leaves *per se*, but it is convenient for the algorithm that the set $leaves(r)$ be non-empty for any semantic value r. To that purpose, we arbitrarily enforce that $leaves([t])$ is the singleton $\{\mathbf{1}\}$ for any pure term t.

As a last remark, we have given the interface of a theory X in a slightly less general fashion as was possible: depending on the theory, the function *solve* may as well return a *list* of pairs (p_i, P_i) with $p_i, P_i \in \mathcal{R}$. It becomes clear why we call this a substitution: the p_i can be seen as variables, that, during the application of a substitution, are replaced by a certain semantic value. However, for the example presented in the next section, *solve* always returns a single pair, if it succeeds at all. Thus, we will stick with the simpler forms of *solve* and *subst* in our presentation.

The following proposition is a simple, but useful, consequence of the axioms stated above. It will be used in the soundness proof. It simply states that, if semantic values constructed with $[\cdot]$ are equal, the original terms were already equal with respect to X.

[4] This is a standard way of ensuring that the substitution is idempotent and that applying it will remove all occurences of the left-hand side.

Proposition 3.2.8. *For any terms $u, v \in \mathcal{T}$, $[u] \equiv [v] \Rightarrow u =_X v$.*

Proof. This is simply axiom 3.2.5 with E the empty sequence. □

Another, less trivial, consequence of the axioms and definitions above is that if r' has been obtained from r by iterated substitution, then the equations at the origin of these substitutions imply the equality $r' \equiv r$.

Proposition 3.2.9. *For any $S \in (\mathcal{R} \times \mathcal{R})^*$ and any $r \in \mathcal{R}$, we have $S \models_X$ $iter(S, r) = r$ where S is seen as a set on the left-hand side of $\models_X$.*

Proof. By definition, we need to show that $iter(S, iter(S, r)) \equiv iter(S, r)$, which can be seen as the idempotency of the iterated substitution. This is of course a consequence of the idempotency of the substitutions returned by *solve* (see Axiom 3.2.4-(i')). We proceed by induction on the sequence of equations S. If S is the empty sequence $\bullet$, the goal becomes $r \equiv r$ which is trivially true.

Now, let us suppose that $S \models_X iter(S, r) = r$ and let r_1, r_2 be some semantic values. We want to prove that $(r_1, r_2); S \models_X iter((r_1, r_2); S, r) = r$. If $iter(S, r)$ is $\bot$, then the result is obviously true; otherwise, $iter(S, .)$ is defined for all values and let $r' = iter(S, r), r'_1 = iter(S, r_1)$ and $r'_2 = iter(S, r_2)$. We proceed by case analysis on the result of $solve(r'_1, r'_2)$:

$\bot$: $iter((r_1, r_2); S, r) = \bot$ hence the result holds.

$\top$: $iter((r_1, r_2); S, r) \equiv iter(S, r) \equiv r'$ and by induction hypothesis the result holds.

(p, P): by definition, $(r_1, r_2); S \models_X iter((r_1, r_2); S, r) = r$ is true if and only if $r' \{p \mapsto P\} \{p \mapsto P\} \equiv r' \{p \mapsto P\}$. By Axioms 3.2.4-(i') and 3.2.6, we know that p does not belong to $leaves(r' \{p \mapsto P\})$ and hence that substituting $\{p \mapsto P\}$ in $r'\{p \mapsto P\}$ does not have any effect, which proves the equality above.

□

In order to prove the completeness, we need to make a few more assumptions about the theory X, or rather about the interpretation of symbols in Σ_{X}.

Axiom 3.2.10. *For each interpreted symbol $f \in \Sigma_{\mathsf{X}}$ of arity n, we assume there exists a function f^{X} from $\mathcal{R}^n$ to $\mathcal{R}$ such that:*

$$\forall t_1, \ldots, t_n \in T(\Sigma), [f(t_1, \ldots, t_n)] \equiv f^{\mathsf{X}}([t_1], \ldots, [t_n])$$

Note, though, that these functions need not be implemented for the algorithm to work: only their existence matters to us, [.] could be computed in any other conceivable way and our algorithm $\mathsf{CC(X)}$ will never need to use one of these functions explicitly. The last axiom simply state that substitutions happen at the leaves level of semantic values.

Axiom 3.2.11. *For any interpreted symbol f, given values $r_1, \dots, r_n$ and two semantic values p and P,*

$$f^{\mathsf{X}}(r_1, \dots, r_n)\{p \mapsto P\} \equiv f^{\mathsf{X}}(r_1\{p \mapsto P\}, \dots, r_n\{p \mapsto P\})$$

Together with Axiom 3.2.10, this last axiom indeed implies that substitution "traverses" interpreted symbols.

3.2.2 The CC(X) Algorithm

The backtracking search underlying the architecture of a lazy SMT solver enforces an incremental treatment of the set of ground equations maintained by the solver. Indeed, for efficiency reasons, equations are given one by one by the SAT solver to the decision procedures which prevents them from realizing a global preliminary treatment, unless restarting the congruence closure from scratch. Therefore, $\mathsf{CC(X)}$ is designed to be incremental and deals with a sequence of equations $u = v$ and queries $u \overset{?}{=} v$ instead of a given set of ground equations.

The algorithm works on tuples (*configurations*) $\langle\, \Theta \mid \Gamma \mid \Delta \mid \Phi \,\rangle$, where:

- Θ is the set of terms already encountered by the algorithm;

- Γ is a mapping from semantic values to sets of terms which intuitively maps each semantic value to the terms that "use" it directly. This structure is reminiscent of Tarjan *et al.*'s algorithm [DST80] but differs in the sense that it traverses interpreted symbols (as expressed in Proposition 3.3.12 in Section 3.3). This information is used to efficiently retrieve the terms which have to be considered for congruence;

- Δ is a mapping from semantic values to semantic values maintaining the equivalence classes over $\mathcal{R}$ as suggested in Section 3.2.1: it is a structure that can tell us if two values are known to be equal (it can be seen as the $find$ function of a union-find data structure);

- Φ is a sequence of equations between terms that remain to be processed.

There is a special kind of configurations written $\langle\, \bot \mid \Phi \,\rangle$ to denote the cases where $\mathsf{CC(X)}$ has reached an inconsistent state, *i.e.* the case where some of the equations already treated are inconsistent with the theory.

Given a sequence E of equations and a query $a \overset{?}{=} b$ for which we want to solve the uniform word problem, $\mathsf{CC(X)}$ starts in an initial configuration $K_0 = \langle\, \emptyset \mid \Gamma_0 \mid \Delta_0 \mid E\ ;\ a \overset{?}{=} b \,\rangle$, where $\Gamma_0(r) = \emptyset$ and $\Delta_0(r) = r$ for all $r \in \mathcal{R}$. In other words, no terms have been treated yet by the algorithm, and the partition Δ_0 corresponds to the physical equality $\equiv$.

In Figure 3.3, we describe our algorithm $\mathsf{CC(X)}$ as six inference rules operating on configurations. The semantic value $\Delta(r)$, for $r \in \mathcal{R}$ is also

$$\textsc{Congr}\ \frac{\langle\ \Theta\ \mid\ \Gamma\ \mid\ \Delta\ \mid\ a = b\ ;\ \Phi\ \rangle}{\langle\ \Theta\ \mid\ \Gamma \uplus \Gamma'\ \mid\ \Delta'\ \mid\ \Phi'\ ;\ \Phi\ \rangle}\ a, b \in \Theta,\quad \Delta[a] \not\equiv \Delta[b]$$

where,

$$(p, P) = solve(\Delta[a], \Delta[b])$$

$$\Gamma' = \bigcup_{l \in leaves(P)} l \mapsto \Gamma(l) \cup \Gamma(p)$$

$$\forall r \in \mathcal{R}, \Delta'(r) := \Delta(r)\,\{p \mapsto P\}$$

$$\Phi' = \left\{ f(\vec{u}) = f(\vec{v}) \middle| \begin{array}{l} \Delta'[\vec{u}] \equiv \Delta'[\vec{v}],\quad f(\vec{u}) \in \Gamma(p) \\ f(\vec{v}) \in \Gamma(p) \cup \bigcup_{t \in \Theta | p \in leaves(\Delta[t])} \bigcap_{l \in leaves(\Delta'[t])} \Gamma(l) \end{array} \right\}$$

$$\textsc{Unsolv}\ \frac{\langle\ \Theta\ \mid\ \Gamma\ \mid\ \Delta\ \mid\ a = b\ ;\ \Phi\ \rangle}{\langle\ \bot\ \mid\ \Phi\ \rangle}\ a, b \in \Theta,\quad \Delta[a] \not\equiv \Delta[b]$$

where $\bot = solve(\Delta[a], \Delta[b])$

$$\textsc{Remove}\ \frac{\langle\ \Theta\ \mid\ \Gamma\ \mid\ \Delta\ \mid\ a = b\ ;\ \Phi\ \rangle}{\langle\ \Theta\ \mid\ \Gamma\ \mid\ \Delta\ \mid\ \Phi\ \rangle}\ a, b \in \Theta,\quad \Delta[a] \equiv \Delta[b]$$

$$\textsc{Add}\ \frac{\langle\ \Theta\ \mid\ \Gamma\ \mid\ \Delta\ \mid\ C[f(\vec{a})]\ ;\ \Phi\ \rangle}{\langle\ \Theta \cup \{f(\vec{a})\}\ \mid\ \Gamma \uplus \Gamma'\ \mid\ \Delta\ \mid\ \Phi'\ ;\ C[f(\vec{a})]\ ;\ \Phi\ \rangle}\ \left\{ \begin{array}{l} f(\vec{a}) \notin \Theta \\ \forall v \in \vec{a}, v \in \Theta \end{array} \right.$$

where $C[f(\vec{a})]$ denotes an equation or a query containing the term $f(\vec{a})$

$$\text{with} \left\{ \begin{array}{l} \Gamma' = \displaystyle\bigcup_{l \in \mathcal{L}_\Delta(\vec{a})} l \mapsto \Gamma(l) \cup \{f(\vec{a})\} \\ \Phi' = \left\{ f(\vec{a}) = f(\vec{b}) \middle| \Delta[\vec{a}] \equiv \Delta[\vec{b}],\quad f(\vec{b}) \in \displaystyle\bigcap_{l \in \mathcal{L}_\Delta(\vec{a})} \Gamma(l) \right\} \end{array} \right.$$

where $\mathcal{L}_\Delta(\vec{a}) = \bigcup_{v \in \vec{a}} leaves(\Delta[v])$

$$\textsc{Query}\ \frac{\langle\ \Theta\ \mid\ \Gamma\ \mid\ \Delta\ \mid\ a \overset{?}{=} b\ ;\ \Phi\ \rangle}{\langle\ \Theta\ \mid\ \Gamma\ \mid\ \Delta\ \mid\ \Phi\ \rangle}\ a, b \in \Theta,\quad \Delta[a] \equiv \Delta[b]$$

$$\textsc{Incons}\ \frac{\langle\ \bot\ \mid\ e\ ;\ \Phi\ \rangle}{\langle\ \bot\ \mid\ \Phi\ \rangle}\ e \text{ equation or query}$$

Figure 3.3: The rules of the congruence closure algorithm $\mathsf{CC(X)}$

called *representative* of r. When t is a term of $\mathcal{T}$, we write $\Delta[t]$ as an abbreviation for $\Delta([t])$, which we call the representative of t. Figure 3.3 also uses several other abbreviations: we write $\vec{u}$ for $u_1, \ldots, u_n$, where n is clear from the context; we also write $\Delta[\vec{u}] \equiv \Delta[\vec{v}]$ for the equivalences $\Delta[u_1] \equiv \Delta[v_1], \ldots, \Delta[u_n] \equiv \Delta[v_n]$. If $t \in \Gamma(r)$ for $t \in \mathcal{T}, r \in \mathcal{R}$, we also say r is *used by* t, or t *uses* r.

We now have all the necessary elements to understand the rules. There are actually only two of them, namely Congr and Add, which perform any interesting tasks. The others are much simpler: Remove just checks if the first equation in Φ is already known to be true (by the help of Δ), and, if so, discards it. Query is analogous to Remove but deals with a query[5]. The other two rules deal with inconsistent configurations: Unsolv takes an unsolvable equation from the sequence of pending equations and returns the inconsistent configuration; rule Incons expresses the fact that once a configuration is inconsistent, all new equations can be ignored, and all queries are true. Finally, note that the case where the first pending equation is already solved is dealt with by the Remove rule, because Axiom 3.2.4-(ii) ensures that $solve(\Delta[a], \Delta[b])$ returns $\top$ if and only if $\Delta[a] \equiv \Delta[b]$.

The rule Congr is much more complex. It deals with the first equation in Φ, but only when it is neither solved nor unsolvable. This equation $a = b$ with $a, b \in \Theta$ is transformed into an equation in $\mathcal{R}$, $\Delta[a] \equiv \Delta[b]$, and then solved in the theory X, which yields two semantic values p and P. The value p is then substituted by P in all representatives. The map Γ is updated according to this substitution: the terms that used p up to that point now also use all the values $l \in leaves(P)$. Finally, a set Φ' of new equations is computed, and appended to the sequence Φ of the equations to be treated (the order of the equations in Φ' is irrelevant). The set Φ' is computed in the following way: the left hand side of any equation in Φ' is a term that used p, and the right hand side is either a term that used p, or a term that used every $l \in leaves(\Delta'(r))$ for a value r such that $p \in leaves(\Delta(r))$. This rather complicated condition ensures that only relevant terms are considered for congruence. As the name implies, the Congr rule will only add equations of the form $f(t_1, \ldots, t_n) = f(t'_1, \ldots, t'_n)$, where the corresponding subterms are already known to be equal: $\Delta'[t_i] \equiv \Delta'[t'_i]$, $1 \leq i \leq n$.

The rule Add is used when the first equation of Φ contains at least a term $f(\vec{a})$ that has not yet been encountered by the algorithm ($f(\vec{a}) \notin \Theta$). Its side condition ensures that all proper subterms of this term have been added before; in other words, new terms are added recursively. The first task that this rule performs is of course to update the map Γ by adding the

[5] Our system does not "return" any truth value for a query *per se*: it passes queries that are true (using the Query rule) and is blocked at false queries.

information that $f(\vec{a})$ uses all the leaves of its direct subterms. However, this is not sufficient: we lose the completeness of the algorithm if no equation is added during the application of an ADD rule. Indeed, suppose for instance that Φ is the sequence $f(a) = t; a = b; f(b) = u$. Then, we would fail to prove that $t = u$ since the equality $a = b$ is processed too early. At this point, $f(b)$ has not been added yet to the structure Γ, thus preventing the congruence equation $f(a) = f(b)$ to be discovered in the CONGR rule. For this reason, the ADD rule also performs congruence closure by looking for equations involving the new term $f(\vec{a})$: this is the construction of the set Φ' of equations, where the restrictive side condition over $f(\vec{b})$ ensures that only relevant terms are considered.

Soundness and completeness proofs of $\mathsf{CC}(\mathsf{X})$ are given in Section 3.3. Since no new terms are generated during $\mathsf{CC}(\mathsf{X})$'s execution, it is easy to bound the number of times that the CONGR rule and the ADD rule can be used. Let k be the number of terms (and subterms) in the input problem: ADD can be called at most k times and CONGR at most $k(k-1)/2$ times. The number of steps in a $\mathsf{CC}(\mathsf{X})$ run is therefore quadratically bounded by the input problem size.

3.2.3 Example: Rational Linear Arithmetic

In this section, we present the theory $\mathcal{A}$ of linear arithmetic over the rationals $\mathbb{Q}$ as an interesting example of instantiation of $\mathsf{CC}(\mathsf{X})$. This theory consists of the following elements:

- The interpreted function symbols are $+, -, \times$ and all constants $q \in \mathbb{Q}$.
- The semantic values are polynomials of the form

  $$c_0 + \sum_{i=1}^{n} c_i \boxed{r_i}, \quad c_i \in \mathbb{Q}\,, \boxed{r_i} \in \mathcal{T}, c_i \neq 0.$$

  From an implementation point of view, these polynomials can be represented as pairs where the left component represents c_0 and the right component is a map from foreign values (terms not handled by linear arithmetic; these are surrounded by a box in this section, in order to distinguish them from interpreted terms) to rationals that represents the sum $\sum_{i=1}^{n} c_i \boxed{r_i}$. Note that in the semantic value above, $+$ is *not* the interpreted function symbol but just notation to separate the different components of the polynomial.
- $=_A$ is just the usual equality of linear arithmetic over rationals.

The functions needed by the algorithm are defined as follows:

- The function $[\cdot]$ interprets the above function symbols as usual and constructs polynomials accordingly.

- The function leaves just returns the set of all the foreign values in the polynomial:

$$leaves\left(c_0 + \sum_{i=1}^{n} c_i\boxed{r_i}\right) = \{\boxed{r_i} \mid 1 \leq i \leq n\}\,.$$

- For the value $\boxed{r}$ and the polynomials p_1, p_2, $subst(\boxed{r}, p_1, p_2)$ replaces the foreign value $\boxed{r}$ by the polynomial p_1 in p_2, if r occurs in p_2.

- For two polynomials $p_1, p_2 \in \mathcal{R}$, $solve(p_1, p_2)$ is simply the Gaussian elimination algorithm that solves the equation $p_1 = p_2$ for a certain foreign value occurring with different coefficients in p_1 and p_2.

If we admit the soundness of the $[\cdot]$ function and the Gauss algorithm used in *solve*, the axioms that need to hold are true and $\mathcal{A}$ is indeed a solvable theory.

We now want to show the execution of $\mathsf{CC(X)}$ by an example using this theory of arithmetic. Consider therefore the set of equations

$$E = \{g(x + k) = a, s = g(k), x = 0\}$$

and we want to find out if the equation $s = a$ follows from E. We will present the equations of E to the algorithm in the same sequence as above. The algorithm starts in the initial configuration $K_0 = \langle\ \emptyset \mid \Gamma_0 \mid \Delta_0 \mid E\ ;\ s \overset{?}{=} a\ \rangle$, as defined in section 3.2.2. In the following, components of the configuration with the subscript i denote the state of the component after complete treatment of the ith equation.

Before being able to treat the first equation $g(x + k) = a$ using the Congr rule, all the terms that appear in the equation have to be added by the Add rule. This means in particular that the components Γ and Θ are updated according to Fig. 3.3. No new equations are discovered, so Φ and Δ remain unchanged. Now we can apply the Congr rule to the first equation $g(x+k) = a$. This yields an update of Γ and Δ, but no congruence equations are discovered. Here is the configuration after the treatment of the first equation:

$$\begin{aligned}
\Gamma_1 &= \left\{\boxed{x} \mapsto \{x + k,\ g(x + k)\}\,,\ \boxed{k} \mapsto \{x + k,\ g(x + k)\}\ \right\}\ \cup\ \Gamma_0 \\
\Delta_1 &= \left\{\boxed{g(x + k)} \mapsto \boxed{a},\ \boxed{a} \mapsto \boxed{a}\right\}\ \cup\ \Delta_0
\end{aligned}$$

The second equation is treated similarly: the terms s and $g(k)$ are Added and the representative of $g(k)$ becomes $\boxed{s}$. These are the changes to the structures Γ and Δ:

$$\begin{aligned}
\Gamma_2 &= \left\{\boxed{k} \mapsto \{x + k,\ g(x + k),\ g(k)\}\ \right\}\ \cup\ \Gamma_1 \\
\Delta_2 &= \left\{\boxed{g(k)} \mapsto \boxed{s},\ \boxed{s} \mapsto \boxed{s}\right\}\ \cup\ \Delta_1
\end{aligned}$$

The most interesting part is the treatment of the third equation, $x = 0$, because we expect the equation $g(x + k) = g(k)$ to be discovered. Otherwise, the algorithm would be incomplete. Every term in the third equation has already been added, so we can directly apply the Congr rule. $solve(\Delta_2[x], \Delta_2[0])$ returns the substitution $(x, 0)$, which is applied to all representatives. The value 0 is a pure arithmetic term, so $leaves(0)$ returns $\{\mathbf{1}\}$. We obtain the following changes to Γ_3 and Δ_3:

$$\begin{array}{rcl} \Gamma_3 & = & \{\mathbf{1} \mapsto \{x + k,\ g(x + k)\}\} \ \cup\ \Gamma_2 \\ \Delta_3 & = & \left\{\boxed{x} \mapsto 0, \boxed{x} + \boxed{k} \mapsto \boxed{k}\right\} \ \cup\ \Delta_2 \end{array}$$

It is important to see that the representative of $x + k$ has changed, even if the term was not directly involved in the equation that was treated.

To discover new equations, the set Φ_3 has to be calculated. To calculate this set, we first collect the terms that use x:

$$\Gamma_2(\boxed{x}) = \{x + k, g(x + k)\}\,.$$

The elements of $\Gamma_2(\boxed{x})$ are potential left-hand sides of new equations. To calculate the set of potential right-hand sides, we first construct the set of values r corresponding to terms in Θ_2 such that the representative of r contains x:

$$\{r \mid x \in leaves(\Delta_2(r))\} = \left\{\boxed{x}, \boxed{x} + \boxed{k}\right\}$$

Now, for every value r in this set, we calculate $leaves(\Delta_3(r))$ and construct their intersection:

$$\begin{array}{rcl} \displaystyle\bigcap_{l \in leaves(0)} \Gamma_2(l) & = & \Gamma_2(\mathbf{1}) = \emptyset \\ \displaystyle\bigcap_{l \in leaves\left(\boxed{k}\right)} \Gamma_2(l) & = & \{x + k, g(x + k), g(k)\} \end{array}$$

The union of the two sets and the set $\Gamma_2(\boxed{x})$ is the set of potential right-hand sides $\{x + k, g(x + k), g(k)\}$. If we cross this set with the set $\Gamma_2(\boxed{x})$ and filter the equations that are not congruent, we obtain three new equalities

$$\Phi_3 = x + k = x + k\ ;\ g(x + k) = g(x + k)\ ;\ g(x + k) = g(k)\ ;\ s \overset{?}{=} a.$$

The first two equations get immediately removed by the Remove rule. The third one, by transitivity, delivers the desired equality which permits to discharge the query $s \overset{?}{=} a$.

3.3 Correctness Proofs

3.3.1 Soundness

We now proceed to prove the soundness of the algorithm. Let E be a set of equations between terms of $\mathcal{T}$ and X a solvable theory as defined page 58. For the proof, we need an additional information about the run of an algorithm, which is not contained in a configuration: the set O of equations that have already been treated in a Congr or Unsolv rule.

The first proposition shows that the equations that are already treated are never contradicted by Δ.

Proposition 3.3.1. *For any configuration* $\langle\ \Theta \mid \Gamma \mid \Delta \mid \Phi\ \rangle$ *and for all* $t_1, t_2 \in \mathcal{T}$ *we have:* $t_1 = t_2 \in O \Rightarrow \Delta[t_1] \equiv \Delta[t_2]$.

Proof. The property is true for the initial configuration K_0 since O is the empty set. We proceed by induction on the derivation that led to the configuration $\langle\ \Theta \mid \Gamma \mid \Delta \mid \Phi\ \rangle$ and by case analysis on the last rule used. The cases of Remove, Query and Add are trivial since they change neither O nor Δ. If the Congr rule is used, the new equation $a = b$ is added to O and Δ is updated with the substitution $(p, P) = \mathit{solve}(\Delta[a], \Delta[b])$. Old equations in O are equal in Δ by induction hypothesis, and as for $a = b$, by Axiom 3.2.4-(i), the new representatives of a and b are equal in the updated Δ. □

The next proposition shows that Δ coincides with the function *iter*, applied to the equations that have already been treated.

Proposition 3.3.2. *For any configuration* $\langle\ \Theta \mid \Gamma \mid \Delta \mid \Phi\ \rangle$ *and for all* $t \in \mathcal{T}$ *we have* $\Delta[t] = \mathit{iter}([O], [t])$.

Proof. It is straightforward to verify this property by induction on O and by definition of *iter*. □

Now that we have characterized the representative of a term t as the result of iterated substitution, we can prove the next proposition. It states that the evolution of the representative of a term is always justified by the equations that have been treated:

Proposition 3.3.3. *For any configuration* $\langle\ \Theta \mid \Gamma \mid \Delta \mid \Phi\ \rangle$ *and for all* $t \in \mathcal{T}$ *we have* $[O] \models_X \Delta_0[t] = \Delta[t]$.

Proof. We have $\Delta_0[t] = [t]$ and by Proposition 3.3.2, $\Delta[t] = \mathit{iter}([O], [t])$. Proposition 3.2.9 ensures that $[O] \models_X t = \mathit{iter}([O], [t])$, hence the result. □

We now turn to the main lemma: it basically states the soundness of Δ, crucial for the soundness of the whole algorithm.

Lemma 3.3.4. *For any configuration* $\langle\, \Theta \mid \Gamma \mid \Delta \mid \Phi \,\rangle$ *and for all* $t_1, t_2 \in \mathcal{T}$, *we have:*

$$\Delta[t_1] \equiv \Delta[t_2] \Rightarrow t_1 =_{X,O} t_2.$$

Proof. By applying Proposition 3.3.3 to t_1 and t_2, we get $[O] \models_X [t_1] = \Delta[t_1]$ and $[O] \models_X [t_2] = \Delta[t_2]$. By transitivity, if $\Delta[t_1] = \Delta[t_2]$, then $[O] \models_X [t_1] = [t_2]$. We now apply Axiom 3.2.5 and obtain $t_1 =_{X,O} t_2$. □

We are now ready to state the main soundness theorem: whenever two terms have the same representative, they are equal w.r.t. the equational theory defined by E and X, and every newly added equation is sound as well. For the soundness of the algorithm, we are only interested in the first statement, but we need the second to prove the first, and the statements have to be proved in parallel by induction.

Theorem 3.3.5. *For any configuration* $\langle\, \Theta \mid \Gamma \mid \Delta \mid \Phi \,\rangle$, *we have:*

$$\begin{array}{lccl} \forall t_1, t_2 \in \mathcal{T}: & \Delta[t_1] \equiv \Delta[t_2] & \Longrightarrow & t_1 =_{X,E} t_2 \\ \forall t_1, t_2 \in \mathcal{T}: & t_1 = t_2 \in \Phi & \Longrightarrow & t_1 =_{X,E} t_2. \end{array}$$

Proof. We prove the two claims simultaneously by induction on the derivation and we are only interested in the application of the rules Congr, Remove, Add and Query. First, we observe that both claims are true for the initial configuration K_0: the second claim is trivial as $\Phi = E$, and the first claim is true because of proposition 3.2.8.

In the induction step, consider the last rule applied to the configuration $\langle\, \Theta \mid \Gamma \mid \Delta \mid \Phi \,\rangle$, and show that the claims still hold in the configuration obtained by application of that rule. For the rules Remove and Query this is actually trivial, as Δ does not change and Φ does not get any new equalities added. For the rule Add, the first claim is trivial, as Δ remains unchanged. The second claim is established as follows. If $t_1 = t_2 \in \Phi$, we can conclude by induction hypothesis. If $t_1 = t_2 \in \Phi'$, then $t_1 \equiv f(\vec{a})$ and $t_2 \equiv f(\vec{b})$, for f with arity n. The conditions in Figure 3.3 guarantee that $\Delta[\vec{a}] \equiv \Delta[\vec{b}]$. By the first claim, we can state that $a_i =_{X,E} b_i$ $(1 \leq i \leq n)$ and by the congruence property of $=_{X,E}$ we have $f(\vec{a}) =_{X,E} f(\vec{b})$, which proves the second claim.

We finally assume that the last rule applied was a Congr rule. To prove the first claim, we assume $\Delta'[t_1] \equiv \Delta'[t_2]$. By lemma 3.3.4, we have $t_1 =_{X,O,a=b} t_2$. Now, $a = b$ is obviously an element of the set $\{a = b\} \cup \Phi$, so that, by induction hypothesis, $a =_{X,E} b$. By the induction hypothesis and proposition 3.3.1, for any $a_i = b_i \in O$ we have also $a_i =_{X,E} b_i$. As $=_{X,E}$ is a congruence relation, we can conclude $t_1 =_{X,E} t_2$. The second claim can be proved as in the case of the Add rule, by the aid of the first claim. □

Until now, we have only addressed the case of consistent configurations and indeed Theorem 3.3.5 establishes the soundness of the Δ map along a derivation as long as the configuration remains consistent. We now deal with inconsistent configurations: in order to be sound, we need to show that as soon as a configuration becomes inconsistent, it must be the case that the original set of equations E is inconsistent with X.

Theorem 3.3.6. *If an inconsistent derivation* $\langle \perp \mid \Phi \rangle$ *is derivable from* K_0*, then* E *and* X *are inconsistent. Consequently,* $a =_{X,E} b$ *for any terms* a *and* b*.*

Proof. When the configuration first becomes inconsistent, it must be by application of the Unsolv rule. Thus, there is a configuration $\langle \Theta \mid \Gamma \mid \Delta \mid a = b; \Phi \rangle$ derivable from K_0 such that $solve(\Delta[a], \Delta[b])$ returns $\perp$. Let O be the equalities treated up to that point. By the second part of Theorem 3.3.5, we know that $a =_{X,E} b$ and that for all $u = v \in O$, $u =_{X,E} v$.

Let t be any term, we want to show that $iter(a = b; O, [t]) = \perp$. By Proposition 3.3.2, $iter(O, [a]) = \Delta[a]$ and $iter(O, [b]) = \Delta[b]$. Thus by definition of $iter$ and since $solve(\Delta[a], \Delta[b])$ returns $\perp$, $iter(a = b; O, [t])$ is undefined. By applying this to any two terms t_1 and t_2, we can prove that $a = b; O \models_X t_1 = t_2$ and by Axiom 3.2.5, this means that $t_1 =_{X,O,a=b} t_2$.

Because this last equality is true for any terms t_1 and t_2 and because $a = b$ and the equations in O are consequences of X and E, X and E are inconsistent. □

3.3.2 Completeness

We finally proceed to the completeness of the algorithm. In opposition to the correctness proof, we are now interested in the fact that every possible equation on the terms of the problem can be deduced by the algorithm, and in particular we are interested in its termination. We will only consider consistent configurations since inconsistent configurations cannot be incomplete.

Termination and congruence closure of Δ

In the following, we assume a fixed problem Π consisting in the set of equations E and a query $a \stackrel{?}{=} b$; we denote the successive configurations by $\langle \Theta_n \mid \Gamma_n \mid \Delta_n \mid \Phi_n \rangle$ with $n = 0$ the initial configuration (as defined in Section 3.2.2). Let T_Π be the set of terms and subterms that appear in $E; a \stackrel{?}{=} b$, in particular, T_Π is closed by subterm. At any stage n in the algorithm, we write O_n for the set of equations that have been treated by the algorithm so far through the rule Congr or Remove.

The first property we are interested in is the fact that all the equations inferred, and thus all the terms added, are only using terms from T_Π.

Proposition 3.3.7. *For any* n*,* $Im(\Gamma_n) \subseteq T_\Pi$*,* $\Phi_n \subseteq T_\Pi \times T_\Pi$ *and* $\Theta_n \subseteq T_\Pi$*.*

Proof. Straightforward to verify by analyzing every rule. □

Theorem 3.3.8 (Termination). *The algorithm terminates on any input problem* Π.

Proof. To prove that this system terminates, it is sufficient to consider the measure defined as $(|T_\Pi \setminus \Theta_n|, |\Delta_n/\equiv|, |\Phi_n|)$, where the second component represents the number of equivalence classes over T_Π in Δ_n. To be precise, the measure is only defined for consistent configurations but inconsistent configurations can be considered as final (they just discard every equation and query pending).

It is immediate to check that, used lexicographically, this measure decreases for every rule of the system. The first element of this measure remains unchanged for all rules except ADD, where it *strictly* decreases: indeed a new term is added to Φ_n and by Proposition 3.3.7, this new term belongs to T_Π.

The second part measures the number of different equivalence classes in Δ_n with respect to $\equiv$. It is obvious that rules REMOVE and QUERY do not alter this quantity. As for CONGR, this quantity decreases strictly since two elements that were different in Δ_n are made equal in Δ_{n+1} by Axiom 3.2.4.

Finally, the third part of the measure is the number of equations and queries that remain to be treated, and it is clear that rules REMOVE, QUERY always remove one element from this set. To sum up, we have the following table :

	$\lvert T_\Pi \setminus \Theta_n\rvert$	$\lvert\Delta_n/\equiv\rvert$	$\lvert\Phi_n\rvert$
ADD	$<$	$\geq$	$\geq$
CONGR	$=$	$<$	$\geq$
REMOVE	$=$	$=$	$<$
QUERY	$=$	$=$	$<$

which proves the termination of the algorithm. □

Now, we know that there exists a final configuration, for $n = \omega$. At this stage, all the equations from the original problem have been treated, and every term in T_Π has been encountered :

Proposition 3.3.9. $O_\omega \supseteq E$.

Proof. Since $\Phi_0 = \Pi$ and all these equations have been treated at the end, it is obvious that O_ω contains at least the equations in Π, *i.e.* E. □

Corollary 3.3.10. *At the end of the algorithm,* $\Theta_\omega = T_\Pi$.

Proof. We already know by 3.3.7 that Θ_ω is included in T_Π. By 3.3.9 and 3.3.7, all the left and right-hand sides of the equations/queries in Π are in Θ_ω. Since Θ_ω is closed by subterm, it also contains T_Π, so it is equal to T_Π. □

Proposition 3.3.11. *The function $n \mapsto \Gamma_n$ is nondecreasing, i.e. $\Gamma_n(r) \subseteq \Gamma_{n+1}(r)$ for all r and n.*

Proof. It is easy to check this property by looking at all the rules. □

The following proposition gives the true "meaning" of the map Γ_n. It shows that a term in Θ_n uses all the leaves of the representatives of its direct subterms.

Proposition 3.3.12. *For any term $f(t_1, \ldots, t_m)$ in Θ_n, if there exists $i \leq m$ such that $p \in leaves(\Delta_n[t_i])$, then $f(t_1, \ldots, t_n) \in \Gamma_n(p)$.*

Proof. The proof proceeds by induction on n. The result holds trivially for the initial configuration since Θ_0 is empty. If the result holds after n steps, we proceed by case analysis on the rule used to get to the $n+1$-th step. The rules Remove, Query do not change Θ_n, Γ_n or Δ_n, so if one of these rules is used the result still holds at $n+1$. We detail both remaining rules :

Congr: Let $f(t_1, \ldots, t_m) \in \Theta_{n+1} = \Theta_n$, and i and p such that $p \in leaves(\Delta_{n+1}[t_i])$. If (v, R) is the substitution applied, by definition of Δ_{n+1}, $p \in leaves(\Delta_n[t_i]\{v \mapsto R\})$. Now, we distinguish two cases :

- if $p \in leaves(\Delta_n[t_i])$, then by induction hypothesis, we know that $f(t_1, \ldots, t_n) \in \Gamma_n(p)$, and thus $f(t_1, \ldots, t_n) \in \Gamma_{n+1}(p)$.
- if $p \notin leaves(\Delta_n[t_i])$, then $\Delta_n[t_i]$ has been changed by the substitution and the axiom 3.2.6 tells us that $v \in leaves(\Delta_n[t_i])$ and $p \in leaves(R)$. Therefore, by applying the induction hypothesis to v and the definition of Γ_{n+1}, we can conclude that :

$$f(t_1, \ldots, t_n) \in \Gamma_n(v) \subseteq \Gamma_n(p) \cup \Gamma_n(v) = \Gamma_{n+1}(p)$$

Add: If $f(t_1, \ldots, t_m)$ was already in Θ_n, then it is straightforward to check that for all $p \in leaves(\Delta_{n+1}([t_i]))$, p was already in $\Delta_n[t_i]$ and the induction hypothesis together with the monotonicity of Γ_n gives us the wanted result.

If $f(t_1, \ldots, t_m)$ is in fact the new term $f(\vec{a})$ added by the rule, then let $p \in leaves(\Delta_{n+1}[t_i])$. Again, p was already in $\Delta_n[t_i]$ and since t_i is a direct subterm of the new added term $f(\vec{a})$, we have by definition that $f(\vec{a}) \in \Gamma_{n+1}(p) = \Gamma_n(p) \cup \{f(\vec{a})\}$.

□

The next proposition is the central property ensuring the completeness of the algorithm, and states that Δ_ω indeed represents a congruence relation.

Proposition 3.3.13. *The restriction of Δ_ω to T_Π is congruence-closed, i.e.*

$$\forall f(\vec{a}), f(\vec{b}) \in T_\Pi,\ \Delta_\omega[\vec{a}] \equiv \Delta_\omega[\vec{b}] \ \Rightarrow\ \Delta_\omega[f(\vec{a})] \equiv \Delta_\omega[f(\vec{b})].$$

Proof. Let k the smallest integer such that both $f(\vec{a})$ and $f(\vec{b})$ belong to Θ_k. Because terms can only be added to Θ by the rule Add, we know the rule applied at the previous step was Add. We can safely assume the term added was $f(\vec{a})$, by switching $\vec{a}$ and $\vec{b}$ if necessary. If $f(\vec{a})$ and $f(\vec{b})$ are equal, the result is obvious. Otherwise, $f(\vec{a}) \neq f(\vec{b})$ and $f(\vec{b})$ had been added before and was in Θ_{k-1}. Now there are two cases, depending on whether $\Delta_{k-1}[\vec{a}] \equiv \Delta_{k-1}[\vec{b}]$ or not.

- if $\Delta_{k-1}[\vec{a}] \equiv \Delta_{k-1}[\vec{b}]$, we will prove that $f(\vec{a}) = f(\vec{b})$ has been added to Φ_k, that is to say we need to establish that :

  $$\forall i, \forall l \in \mathit{leaves}(\Delta_{k-1}[a_i]), f(\vec{b}) \in \Gamma_{k-1}(l).$$

  For any such i and l, we know that l is in $\mathit{leaves}(\Delta_{k-1}[a_i])$, and therefore in $\mathit{leaves}(\Delta_{k-1}[b_i])$. By Proposition 3.3.12, this means that $f(\vec{b}) \in \Gamma_{k-1}(l)$, which is exactly what we wanted.

- if on the contrary, $[\vec{a}]$ and $[\vec{b}]$ were not equal in Δ_{k-1}, then let $j \geq k$ be the smallest integer such that $\Delta_j[\vec{a}] \equiv \Delta_j[\vec{b}]$. The rule applied at the previous step must be Congr since only Congr changes Δ. Thus, a substitution $\{p \mapsto P\}$ has made $\Delta_{j-1}[\vec{a}]$ and $\Delta_{j-1}[\vec{b}]$ equal: there exists an i, such that

  $$\Delta_{j-1}[a_i] \not\equiv \Delta_{j-1}[b_i] \ \wedge\ \Delta_{j-1}[a_i]\{p \mapsto P\} \equiv \Delta_{j-1}[b_i]\{p \mapsto P\}.$$

  This means that at least one of these values, say $\Delta_{j-1}[a_i]$, has been changed by the substitution and by Axiom 3.2.6, that p belongs to $\mathit{leaves}(\Delta_{j-1}[a_i])$. Proposition 3.3.12 ensures that $f(\vec{a}) \in \Gamma_{j-1}(p)$.

  We still have to prove that $f(\vec{b})$ verifies the conditions in the rule Congr, namely that:

  $$f(\vec{b}) \in \Gamma_{j-1}(p) \cup \bigcup_{t | p \in \mathit{leaves}(\Delta_{j-1}(t))} \bigcap_{l \in \mathit{leaves}(\Delta_j(t))} \Gamma_{j-1}(l).$$

  Again, we distinguish two cases :

  - if $\Delta_{j-1}[b_i] \not\equiv \Delta_j[b_i]$, then by the same argument as above for $f(\vec{a})$, $f(\vec{b}) \in \Gamma_{j-1}(p)$ and $f(\vec{b})$ has the desired property.

- if $\Delta_{j-1}[b_i] \equiv \Delta_j[b_i]$, then $leaves(\Delta_j[a_i]) = leaves(\Delta_j[b_i]) = leaves(\Delta_{j-1}[b_i])$ and by applying Proposition 3.3.12 once again, we deduce that for every l in $leaves(\Delta_j[a_i])$, $f(\vec{b}) \in \Gamma_{j-1}(l)$. Since $p \in leaves(\Delta_{j-1}[a_i])$, this means indeed that:

$$f(\vec{b}) \in \bigcup_{t|p\in leaves(\Delta_{j-1}(t))} \bigcap_{l\in leaves(\Delta_j(t))} \Gamma_{j-1}(l).$$

So far, we have established that the equation $f(\vec{a}) = f(\vec{b})$ has been added when the rule Congr was applied at the step $j-1$, and thus that $f(\vec{a}) = f(\vec{b})$ belongs to Φ_j. At the end of the algorithm, this equation must have been treated. Thus, by 3.3.1, we know that the representatives of $f(\vec{a})$ and $(\vec{b})$ are equal in Δ_ω.

□

The axioms 3.2.10 and 3.2.11 introduced in Section 3.2.1 are used to prove that the Δ_ω component of the final configuration is coherent with the theory X, that is to say:

Proposition 3.3.14. *Let $f(t_1,\dots,t_n)$ a term in T_Π where f is an interpreted symbol. Then, $\Delta_\omega[f(t_1,\dots,t_n)] \equiv f^{\mathsf{X}}(\Delta_\omega[t_1],\dots,\Delta_\omega[t_n])$.*

Proof. We will prove this result by proving it (by simple induction) for Δ_n for every N between 0 and the final configuration.

First, we observe that the result is true for the initial configuration, *i.e.* $\Delta_0[f(t_1,\dots,t_m)] \equiv f^{\mathsf{X}}(\Delta_0[t_1],\dots,\Delta_0[t_m])$ because it directly follows from Axiom 3.2.10 and the definition of Δ_0.

Now, it is sufficient to show that if the equality holds for Δ_n, it still holds in Δ_{n+1}. Since the only rule that changes Δ_n is Congr, the result is obvious for any other rule. In the case of a Congr rule, let p, P be the substitution applied to Δ_n :

$$\begin{array}{rcl}\Delta_{n+1}[f(t_1,\dots,t_m)] & = & \Delta_n[f(t_1,\dots,t_m)]\{p \mapsto P\} \text{ by definition}\\ & \equiv & f^{\mathsf{X}}(\Delta_n[t_1],\dots,\Delta_n[t_m])\{p \mapsto P\} \text{ by induction}\\ & \equiv & f^{\mathsf{X}}(\Delta_n[t_1]\{p \mapsto P\},\dots,\Delta_n[t_m]\{p \mapsto P\}) \text{ by 3.2.11}\\ & \equiv & f^{\mathsf{X}}(\Delta_{n+1}[t_1],\dots,\Delta_{n+1}[t_m]) \text{ by definition}\end{array}$$

which proves the result. □

In other words, this property means that Δ actually represents a union-find structure modulo X, that is, it behaves correctly with respect to the interpreted symbols.

Models and Structures

We now recall some usual definitions about structures and models on a certain signature, which we will use to finish the completeness proof.

Definition 3.3.15. *A* Σ-structure $\mathcal{M}$ *is defined as a tuple* $(|\mathcal{M}|, (f^{\mathcal{M}})_{f\in\Sigma})$ *where:*

- $|\mathcal{M}|$ *is a set called the* domain *of* $\mathcal{M}$
- *for each function symbol* $f \in \Sigma$ *of arity* n*,* $f^{\mathcal{M}}$ *is a function from* $|\mathcal{M}|^n$ *to* $|\mathcal{M}|$ *called the* interpretation *of* f *in* $\mathcal{M}$

Definition 3.3.16. *Let* $\mathcal{M}$ *be a* Σ*-structure,* t *a term in* $\mathcal{T}$*. The* interpretation *of* t *in* $\mathcal{M}$*, noted* $\mathcal{M}(t)$*, is recursively defined*[6] *by:*

$$\forall f \in \Sigma, t_1, \ldots t_n \in \mathcal{T}, \mathcal{M}(f(t_1, \ldots, t_n)) \overset{def}{=} f^{\mathcal{M}}(\mathcal{M}(t_1), \ldots, \mathcal{M}(t_n))$$

Σ-structures can now be used as models for our atoms, in the sense of Definition 2.1.1 page 27. Recall that in this chapter, atoms are equations between terms of $\mathcal{T}$.

Definition 3.3.17. *Let* $\mathcal{M}$ *be a* Σ*-structure,* t, u *terms in* $\mathcal{T}$*. We say that* $\mathcal{M}$ *is a* model *of* $t = u$*, written* $\mathcal{M} \models t = u$*, if and only if* $\mathcal{M}(t) \equiv \mathcal{M}(u)$*.*

Completeness

The completeness expresses the fact that if the query is entailed by the set of equations E and the theory X, it is proved true by $\mathsf{CC}(\mathsf{X})$. In other words, we need to prove that:

$$a =_{X,E} b \quad \Longrightarrow \quad \Delta_\omega[a] \equiv \Delta_\omega[b].$$

The first step of the proof is to build a Σ-structure $\mathcal{M}$ which models E and $=_{\mathsf{X}}$, and such that the interpretation in $\mathcal{M}$ coincides with Δ_ω on $[a]$ and $[b]$.

Definition 3.3.18. *Let* $\mathcal{M}$ *be the structure defined in the following way :*

- *the domain of* $\mathcal{M}$ *is the set* $\mathcal{R}$ *of semantic values*
- *for each symbol* $f \in \Sigma$ *of arity* n*, we distinguish whether* f *is interpreted in* X *or not :*
  - *if* $f \in \Sigma_{\mathsf{X}}$*, then* $f^{\mathcal{M}} \overset{def}{=} f^{\mathsf{X}}$
  - *if* $f \notin \Sigma_{\mathsf{X}}$*, and* $r_1, \ldots, r_n \in \mathcal{R}$*, then the idea is to use* Δ_ω *wherever we can :*

$$\begin{array}{lcll} f^{\mathcal{M}}(r_1, \ldots, r_n) & \overset{def}{=} & \Delta_\omega[f(t_1, \ldots, t_n)] & \left\{ \begin{array}{l} \textit{if } f(t_1, \ldots, t_n) \in T_\Pi \\ \textit{and } \forall i, r_i \equiv \Delta_\omega[t_i] \end{array} \right. \\ f^{\mathcal{M}}(r_1, \ldots, r_n) & \overset{def}{=} & \mathbf{1} & \textit{otherwise} \end{array}$$

Here, we use $\mathbf{1}$*, but we could use any element in* $\mathcal{R}$*, since we will see that it does not matter how we define interpretations in this case.*

[6] the base case being 0-ary function symbols, i.e. constants.

Proof. The very first thing we have to do is to prove that the definition we just gave is indeed a definition. In the case where $f^{\mathcal{M}}$ is defined in terms of Δ_ω, there may be several ways to pick the terms t_i and we have to show that the result does not depend on this choice. Let $t_1, \ldots, t_n$, $u_1, \ldots, u_n$ be terms such that $\Delta_\omega[t_i] \equiv r_i \equiv \Delta_\omega[u_i]$ for all i. By Proposition 3.3.13, we know that $\Delta_\omega[f(t_1, \ldots, t_n)] \equiv \Delta_\omega[f(u_1, \ldots, u_n)]$, which means exactly that the definition of $f^{\mathcal{M}}(r_1, \ldots, r_n)$ does not depend on the choice of the t_i. □

Now that $\mathcal{M}$ is a well-defined Σ-structure, we will first show that on all the terms in T_Π, the interpretation in $\mathcal{M}$ is exactly the function $\Delta_\omega[.]$.

Lemma 3.3.19. *For any term $t \in T_\Pi$, $\mathcal{M}(t) \equiv \Delta_\omega[t]$.*

Proof. We proceed by structural induction on terms.

Let $t = f(t_1, \ldots, t_n) \in T_\Pi$, we can apply the induction hypothesis to all the t_i because T_Π is closed by subterm. Thus, for all i, $\mathcal{M}(t_i) \equiv \Delta_\omega[t_i]$.

Now, if $f \notin \Sigma_{\mathsf{X}}$,

$$\begin{aligned} \mathcal{M}(f(t_1, \ldots, t_n)) &= f^{\mathcal{M}}(\mathcal{M}(t_1), \ldots, \mathcal{M}(t_n)) \\ &\equiv f^{\mathcal{M}}(\Delta_\omega[t_1], \ldots, \Delta_\omega[t_n]) \text{ by IH} \\ &\equiv \Delta_\omega[f(t_1, \ldots, t_n)] \text{ by definition of } f^{\mathcal{M}} \end{aligned}$$

If $f \in \Sigma_X$, then

$$\begin{aligned} \mathcal{M}(f(t_1, \ldots, t_n)) &= f^{\mathcal{M}}(\mathcal{M}(t_1), \ldots, \mathcal{M}(t_n)) \\ &\equiv f^{\mathcal{M}}(\Delta_\omega[t_1], \ldots, \Delta_\omega[t_n]) \text{ by IH} \\ &\equiv f^{\mathsf{X}}(\Delta_\omega[t_1], \ldots, \Delta_\omega[t_n]) \text{ by definition of } f^{\mathcal{M}} \\ &\equiv \Delta_\omega[f(t_1, \ldots, t_n)] \text{ by 3.3.14 since } f(t_1, \ldots, t_n) \in T_\Pi \end{aligned}$$

which concludes the proof. □

Finally, we show that $\mathcal{M}$ is a model of $=_{\mathsf{x}}$ and E, *i.e.* that it models all equalities in the congruence closure of X and E.

Lemma 3.3.20. *For all $u, v \in \mathcal{T}$, $u =_{X,E} v \implies \mathcal{M} \models u = v$.*

Proof. Since $\mathcal{M}$ is a structure whose domain $\mathcal{R}$ is the domain of semantic values of X, and since the interpretation in $\mathcal{M}$ of every interpreted symbol f is precisely its interpretation in X, namely f^{X}, $\mathcal{M}$ is a model of $=_{\mathsf{x}}$.

Moreover, let $t = u$ be an equation in E. Since t and u are in T_Π, the preceding lemma tells us that $\mathcal{M}(t) \equiv \Delta_\omega[t]$ and $\mathcal{M}(u) \equiv \Delta_\omega[u]$. By proposition 3.3.9, we know that since $t = u$ is in E, it has been treated at the end and $\Delta_\omega[t] \equiv \Delta_\omega[u]$. Thus, $\mathcal{M}(t) \equiv \mathcal{M}(u)$ for any equation $t = u$ in E, and $\mathcal{M} \models E$. □

Theorem 3.3.21 (Completeness)**.** $\forall a, b \in \mathcal{T}, a =_{X,E} b \implies \Delta_\omega[a] \equiv \Delta_\omega[b]$.

Proof. By lemma 3.3.20, $\mathcal{M}$ is a model of E and $=_{\mathsf{x}}$. Therefore, since $a =_{X,E} b$, it must be the case that $\mathcal{M}$ is also a model of $a = b$, in other words, that $\mathcal{M}(a) \equiv \mathcal{M}(b)$. Hence, by lemma 3.3.19, $\Delta_\omega[a] \equiv \Delta_\omega[b]$. □

We have established the completeness of $\mathsf{CC}(\mathsf{X})$.

3.4 Adding Disequalities

In the previous sections, we have presented a new algorithm called $\mathsf{CC(X)}$ which performs the congruence closure of a set of equations modulo a solvable theory X. In order to use such a system in an SMT solver, we need to turn it into an environment suitable for the DPLL procedure, as described in Section 2.3. The missing part in $\mathsf{CC(X)}$ as presented so far is that the SAT solver will feed the environment with both positive and negative literals, and query positive and negative literals as well. Therefore, we need to adapt our system such that it is able to deal with disequalities as well and we present such an extension in this section.

The modifications required to deal with disequalities can be roughly summarized as follows:

(a) we must account for inputs of the form $a \neq b$ where a and b are some terms: the algorithm will store an extra relation $N \subseteq \mathcal{T} \times \mathcal{T}$ which gathers all such constraints;

(b) there is a new way for the configurations to become inconsistent, namely when treating an equation which contradicts the constraints gathered in N: solving a (solvable) equation in Congr rule can merge two terms in Δ which are unequal according to N, and conversely, adding a disequality constraint can contradict the current Δ;

(c) when dealing with a negative query $a \overset{?}{\neq} b$, we must determine whether a and b can be equal or not: it is not sufficient to check the current constraints N because merging a and b can lead to more equalities (modulo X), one way to do so is to try and add the equation $a = b$ and test if the configuration becomes inconsistent.

Note that N must be an irreflexive, symmetric relation; adding the disequality $a \neq b$ to N yields the relation $N \cup \{(a, b); (b, a)\}$. The N structure can be implemented in a variety of ways, one possible way is to map terms to the set of terms which are different. For modification (b), it is necessary to check that the union-find Δ and the relation N are not contradictory.

Definition 3.4.1. *Let $N \subseteq \mathcal{T} \times \mathcal{T}$ a relation over terms and $\Delta : \mathcal{R} \to \mathcal{R}$ a union-find on semantic values. We say that N and Δ are* coherent *if:*

$$\forall a, b \in \mathcal{T}, (a, b) \in N \implies \Delta[a] \not\equiv \Delta[b]$$

We say that they are incoherent *otherwise.*

Because the relation N remains finite (since there are a finite number of inputs after a finite number of steps), this coherence check can be implemented without problem. Finally, in order to deal with modification (c), we define a couple of notations: if K is a configuration, we write $K \uparrow$ if there is a derivation from K to an inconsistent configuration, and $K \downarrow$ if the configuration remain consistent and all queries succeed.

We now present an extended set of inference rules which completes and corrects the rules in Figure 3.3. Configurations are extended with the N structure and are now written $\langle\ \Theta \mid \Gamma \mid \Delta \mid N \mid \Phi\ \rangle$ and the initial configuration K_0 is just as before with an empty $N = \emptyset$. The extended inference system is given in Figure 3.4. The rules Remove, Unsolv, Add, Query and Incons are left unchanged: the N structure is simply passed from one configuration to the next. The Congr rule is modified so that it only applies if the resulting union-find Δ' is coherent with the set of constraints; other than that it is left unchanged. The new rule IncohEq takes care of the case when Δ' and N are incoherent. There are three rules left, all new with respect to Figure 3.3, and they all deal with disequalities. Diff adds a new disequality constraint to the structure N, but only if that does not contradict the current map Δ. If it does, then IncohDiff applies and yields an inconsistent configuration. Finally, negative queries are handled by QueryDiff, which only accepts a query $a \neq b$ if adding the equality $a = b$ to the current configuration raises an inconsistency.

Adapting the proofs. It is straightforward to check that the proofs that we did in Section 3.3 still hold (for the most part) for this extended system. Indeed, the extended $\mathsf{CC(X)}$ deals with equalities (whether inputs or queries) in exactly the same way as the original $\mathsf{CC(X)}$: the only difference lurks in the fact that an equation can contradict some previous disequalities, *i.e.* the IncohEq rule. Therefore the extended system can yield more inconsistencies, but consistent configurations remain the same and therefore remain correct and complete (as far as equalities are concerned). More formally, if Π is the input problem, let us call E^+ the set of input equations in Π and E^- the set of disequalities. We can reproduce the exact same reasoning that led to Theorem 3.3.5 and deduce:

Theorem 3.4.2. *For any configuration* $\langle\ \Theta \mid \Gamma \mid \Delta \mid N \mid \Phi\ \rangle$, *we have:*

$$\begin{array}{llcl} \forall t_1, t_2 \in \mathcal{T}: & \Delta[t_1] \equiv \Delta[t_2] & \Longrightarrow & t_1 =_{X,E^+} t_2 \\ \forall t_1, t_2 \in \mathcal{T}: & t_1 = t_2 \in \Phi & \Longrightarrow & t_1 =_{X,E^+} t_2. \end{array}$$

Similary, the soundness of inconsistent $\mathsf{CC(X)}$ configurations, namely Theorem 3.3.6, can be obtained by replacing E with the only positive inputs E^+:

Theorem 3.4.3. *If an inconsistent derivation* $\langle\ \bot \mid \Phi\ \rangle$ *is derivable from* K_0 *using Unsolv, then* E^+ *and* X *are inconsistent. Consequently, the equation* $a =_{X,E^+} b$ *folds for any terms* a *and* b.

In order to complete the soundness proof for the extended system, we need invariants on the N structure along the derivation: all constraints in N must be consequences of the disequalities in E^-.

$$\textsc{Congr}\ \frac{\langle\ \Theta \mid \Gamma \mid \Delta \mid N \mid a = b\ ;\ \Phi\ \rangle}{\langle\ \Theta \mid \Gamma \uplus \Gamma' \mid \Delta' \mid N \mid \Phi'\ ;\ \Phi\ \rangle}\ a, b \in \Theta,\quad \Delta[a] \not\equiv \Delta[b]$$

where
Γ', Δ' and Φ' are computed as in the Congr rule in Figure 3.3
N and Δ' are coherent

$$\textsc{IncohEq}\ \frac{\langle\ \Theta \mid \Gamma \mid \Delta \mid N \mid a = b\ ;\ \Phi\ \rangle}{\langle\ \bot \mid \Phi\ \rangle}\ a, b \in \Theta,\quad \Delta[a] \not\equiv \Delta[b]$$

where
Γ', Δ' and Φ' are computed as in the Congr rule in Figure 3.3
N and Δ' are incoherent

$$\textsc{Diff}\ \frac{\langle\ \Theta \mid \Gamma \mid \Delta \mid N \mid a \neq b\ ;\ \Phi\ \rangle}{\langle\ \Theta \mid \Gamma \mid \Delta \mid N \cup \{(a,b);(b,a)\} \mid \Phi\ \rangle}\ a, b \in \Theta,\quad \Delta[a] \not\equiv \Delta[b]$$

$$\textsc{IncohDiff}\ \frac{\langle\ \Theta \mid \Gamma \mid \Delta \mid N \mid a \neq b\ ;\ \Phi\ \rangle}{\langle\ \bot \mid \Phi\ \rangle}\ a, b \in \Theta,\quad \Delta[a] \equiv \Delta[b]$$

$$\textsc{Unsolv}\ \frac{\langle\ \Theta \mid \Gamma \mid \Delta \mid N \mid a = b\ ;\ \Phi\ \rangle}{\langle\ \bot \mid \Phi\ \rangle}\ a, b \in \Theta,\quad \Delta[a] \not\equiv \Delta[b]$$

where $\bot = solve(\Delta[a], \Delta[b])$

$$\textsc{Remove}\ \frac{\langle\ \Theta \mid \Gamma \mid \Delta \mid N \mid a = b\ ;\ \Phi\ \rangle}{\langle\ \Theta \mid \Gamma \mid \Delta \mid N \mid \Phi\ \rangle}\ a, b \in \Theta,\quad \Delta[a] \equiv \Delta[b]$$

$$\textsc{Add}\ \frac{\langle\ \Theta \mid \Gamma \mid \Delta \mid N \mid C[f(\vec{a})]\ ;\ \Phi\ \rangle}{\langle\ \Theta \cup \{f(\vec{a})\} \mid \Gamma \uplus \Gamma' \mid \Delta \mid N \mid \Phi'\ ;\ C[f(\vec{a})]\ ;\ \Phi\ \rangle}\ \begin{cases} f(\vec{a}) \notin \Theta \\ \forall v \in \vec{a}, v \in \Theta \end{cases}$$

where Γ' and Φ' are computed as in the Add rule in Figure 3.3

$$\textsc{Query}\ \frac{\langle\ \Theta \mid \Gamma \mid \Delta \mid N \mid a \overset{?}{=} b\ ;\ \Phi\ \rangle}{\langle\ \Theta \mid \Gamma \mid \Delta \mid N \mid \Phi\ \rangle}\ a, b \in \Theta,\quad \Delta[a] \equiv \Delta[b]$$

$$\textsc{QueryDiff}\ \frac{\langle\ \Theta \mid \Gamma \mid \Delta \mid N \mid a \overset{?}{\neq} b\ ;\ \Phi\ \rangle}{\langle\ \Theta \mid \Gamma \mid \Delta \mid N \mid \Phi\ \rangle}\ \begin{array}{l} a, b \in \Theta,\quad \Delta[a] \equiv \Delta[b] \\ \langle \Theta \mid \Gamma \mid \Delta \mid N \mid a = b \rangle \uparrow \end{array}$$

$$\textsc{Incons}\ \frac{\langle\ \bot \mid e\ ;\ \Phi\ \rangle}{\langle\ \bot \mid \Phi\ \rangle}\ e \text{ equation or query}$$

Figure 3.4: The rules of CC(X) extended to deal with disequalities

Proposition 3.4.4. *If* $\langle\ \Theta \mid \Gamma \mid \Delta \mid N \mid \Phi\ \rangle$ *is derivable from* K_0*, then all terms* $(a, b) \in N$ *are such that* $a \neq b$ *or* $b \neq a$ *belong to* E^-*.*

Proof. The proof is easy by induction on the derivation; only the Diff rule adds elements to N, and no rule ever adds disequalities to the pending inputs. □

We can now prove the soundness of the system when it reaches inconsistent configurations.

Proposition 3.4.5. *If an inconsistent configuration* $\langle\ \bot \mid \Phi\ \rangle$ *is derivable from* K_0*, then the union of* E^+*,* E^- *and* X *are inconsistent,* i.e. *there exists no model* $\mathcal{M}$ *of* E^+, X *such that all disequalities in* E^- *are false in* $\mathcal{M}$*.*

Proof. By case analysis on the rule which made the configuration inconsistent. The case of the Unsolv rule is given by Theorem 3.4.3: E^+ and X together are already inconsistent.

If IncohEq is used then N and Δ' are incoherent. Therefore, there exists u, v two terms such that $(u, v) \in N$ and $\Delta[u] \equiv \Delta[v]$. By Theorem 3.4.2, we know that $u =_{X,E^+} v$ and by Proposition 3.4.4 that $u \neq v$ or $v \neq u$ belongs to E^-. Therefore, X, E^+ is inconsistent with E^-.

Finally, if IncohDiff is used then there is $a \neq b \in E^-$ such that $\Delta[a] \equiv \Delta[b]$. By Theorem 3.4.2, we know that $a =_{X,E^+} b$ and therefore X, E^+ is inconsistent with E^-. □

This last proposition also gives us the soundness of the treatment of negative queries, *i.e.* the soundness of the QueryDiff rule. Indeed, if $\langle\ \Theta \mid \Gamma \mid \Delta \mid N \mid a = b\ \rangle \uparrow$, then by Proposition 3.4.5, $a = b, E^+, X$ and E^- are inconsistent. Therefore if $\mathcal{M}$ models E^+, X and E^-, it is impossible that $\mathcal{M} \models a = b$ holds. In other words, $a \neq b$ is indeed a consequence of the inputs E and the theory X.

Now that we have established the soundness of the extended system, we prove its completeness. We use the same notations as in Section 3.3, for the fixed sequence of inputs E (which are split in equalities E^+ and disequalities E^-). Once again we only deal with consistent configurations since inconsistent configurations are necessarily complete. The completeness theorem is expressed in two parts, one for positive queries and one for negative queries.

Theorem 3.4.6. *Let* a, b *be two terms in* $\mathcal{T}$*.*

(i) Assume that $\forall \mathcal{M}, \mathcal{M} \models E, X \implies \mathcal{M} \models a = b$*.*
Then, $\langle\ \Theta_0 \mid \Gamma_0 \mid \Delta_0 \mid N_0 \mid E; a \overset{?}{=} b\ \rangle \downarrow$*.*

(ii) Assume that $\forall \mathcal{M}, \mathcal{M} \models E, X \implies \mathcal{M} \models a \neq b$*.*
Then, $\langle\ \Theta_0 \mid \Gamma_0 \mid \Delta_0 \mid N_0 \mid E; a \overset{?}{\neq} b\ \rangle \downarrow$*.*

Proof.

(i) Let us assume that $a = b$ is entailed by E, X. We want to prove that $\langle\ \Theta_0 \mid \Gamma_0 \mid \Delta_0 \mid N_0 \mid E; a \overset{?}{=} b\ \rangle \downarrow$, and such a derivation can only end with the Query rule therefore it is enough to prove that $\Delta_\omega[a] \equiv \Delta_\omega[b]$.

We now build a special Σ-structure $\mathcal{M}$ in exactly the same way as we did in Section 3.3, with domain $\mathcal{R}$ and such that it coincides with Δ_ω everywhere possible. By Proposition 3.3.19, we know that for every term t in the problem, $\mathcal{M}(t) \equiv \Delta_\omega[t]$. Adapting Lemma 3.3.20, we also know that $\mathcal{M}$ is a model of X, E^+. Finally, let $u, v \in E^-$: we know that E^- has been treated by a Diff rule, so (u, v) belongs to N. Because N_ω and Δ_ω are coherent, this means that $\Delta_\omega[u] \not\equiv \Delta_\omega[v]$. Thus, $\mathcal{M}$ is a model of E^- as well and altogether, $\mathcal{M} \models E, X$. By hypothesis, this means that $\mathcal{M} \models a = b$, that is to say $\mathcal{M}(a) \equiv \mathcal{M}(b)$. Since a and b are terms of the problem, $\Delta_\omega[a] \equiv \Delta_\omega[b]$.

(ii) We now assume that $a \neq b$ is entailed by E, X. We want to prove that $\langle\ \Theta_0 \mid \Gamma_0 \mid \Delta_0 \mid N_0 \mid E; a \overset{?}{\neq} b\ \rangle \downarrow$, and such a derivation can only end with the QueryDiff rule therefore it is easy to see that it amounts to proving that $\langle\ \Theta_0 \mid \Gamma_0 \mid \Delta_0 \mid N_0 \mid E; a = b\ \rangle \uparrow$.

We proceed *ab absurdo*: if $\langle\ \Theta_0 \mid \Gamma_0 \mid \Delta_0 \mid N_0 \mid E; a = b\ \rangle$ does not yield an inconsistent configuration, there is a final configuration with a map that we denote Δ_ω; we proceed as in the (i) part and build a Σ-structure $\mathcal{M}$ such that $\mathcal{M}$ is a model of E, X and such that $\mathcal{M}(t) \equiv \Delta_\omega[t]$ for all terms t appearing in $E; a = b$. By hypothesis, we know that $\mathcal{M} \models a \neq b$; on the other hand, because $a = b$ has been treated in Δ_ω, it must be the case that $\Delta_\omega[a] \equiv \Delta_\omega[b]$, which means that $\mathcal{M} \models a = b$. We have reached a contradiction. □

3.5 Conclusion

We have presented a new algorithm CC(X) which combines the theory of equality over uninterpreted function symbols with a solvable theory. Our method is inspired by Shostak's algorithm and its main novelty rests in the use of abstract data structures for class representatives; this allows efficient implementations of crucial operations. Our approach is also modular unlike *ad-hoc* extensions of congruence closure [NO80, NO07], CC(X) can be instantiated with an arbitrary solvable theory underlying the restrictions described in Section 3.2.

We gave a useful example of a solvable theory in Section 3.2.3 with the theory of linear rational arithmetic. The same theory can also be used to deal with linear integer arithmetic, which does not have a solver, but it can be incomplete. For instance, the formula:

$$\forall xyz : \mathbb{Z}, 2 * x = z \implies 2 * y \neq z + 1$$

cannot be established. Because such formulae are not frequent in program verification in practice, Alt-Ergo basically uses the theory of rational arithmetic in order to deal with integers[7]. This illustrates one interesting feature of the ability to use semantic values: this decision procedure for integers can manipulate and construct semantic values which do not correspond to terms, e.g. the constant polynom 1/2, which is not possible with Shostak's procedure. There are other theories of interest which happen to be solvable theories and are implemented in Alt-Ergo: a theory of pairs (similar to the theory given as an example by Shostak in [Sho84]) and a theory of finite vectors.

Nevertheless, solvable theories are still a quite strongly constrained class of theories. They are included in the class of Nelson-Oppen theories. They are stably infinite because the semantic values of a solvable theory need to be able to embed the set of all terms $\mathcal{T}$ (through the $[.]$ function). In particular, it is not possible to deal with a theory of finite types because CC(X) has a coarse treatment of disequalities: a term a in N can be constrained to be different from arbitrarily many terms and CC(X) will not detect inconsistencies due to an upper limit on the cardinality of a model. There are combination schemes which try to address cardinality constraints thoroughly: for instance, Tinelli and Zarba [TZ03] proposed a combination scheme in which any theory can be combined with a special kind of theory (*shiny* theories) which have a function to compute cardinality constraints. Finally, Ranise, Ringeissen and Tran proposed a combination scheme for a class of theories strictly included between Nelson-Oppen and Shostak theories [RRT04].

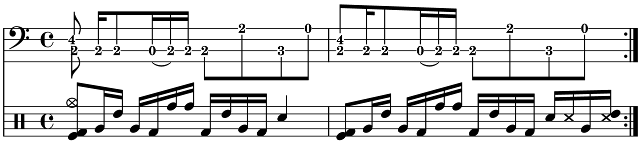
[7] Alt-Ergo still tries to do some integer-specific reasoning: for instance, strict inequalities $a < b$ are transformed into large equalities $a \leq b - 1$, thus using a fundamental characteristic of integers.

Part II

Ergo: a Reflexive Tactic for Automated Deduction in Coq

CHAPTER 4

Proving by Reflection in Coq



Tool, *Reflection*

Contents

The second part of this document is devoted to the presentation of a Coq formalization of the SMT techniques described in Part 1. We start in this chapter by giving a detailed introduction to the Coq proof assistant and the technique of proof by reflection. We first describe Coq's underlying logic and its general features in Section 4.1.1, before we detail the different techniques for proof automation in Coq and in interactive provers in general (Section 4.2). We finish with an outline of our reflexive SMT solver in Section 4.3.

4.1 Introduction to Coq

Coq is a proof assistant for higher-order logic whose development started in the middle of the 1980's, and which is now maintained and developed in the TypiCal project [Typ]. We start by describing its logical language before we deal with the proof assistant *per se*.

4.1.1 CIC: The Calculus of Inductive Constructions

The logical language on which Coq is based is an evolution of a calculus proposed by T. Coquand and G. Huet in [CH88], the *Calculus of Constructions* (CoC). This calculus is itself an extension of type theory and can be seen as a combination of the principles of two successful type-theoretic frameworks from the 1970's, Martin-Löf's *Intuitionistic Theory of Types* [ML75] and Girard's second-order λ-calculus F_ω.

One of the main specificities of Martin-Löf's theory is the dependent product Π which allows one to quantify over both objects and types and these dependent types allow one to express much more properties through types than in standard simply-typed λ-calculus. Through the Curry-Howard isomorphism, which identifies programs to proofs and types to logical propositions, this system can be used as a foundation of constructive mathematics.

In the CoC, λ-abstractions are typed with a dependent product noted $\forall$ using the following rule: if, for all x with type T, the term u has type U, the term $\lambda x : T.u$ has type $\forall x : T.U$, where U can mention the variable x. Therefore a product type $\forall x : T.U$ can be read both as the type of a dependent function, or as a universal quantification over objects of type T. When U does not mention x, the product becomes non-dependent and is written $T \to U$, which can be read as a traditional function type or as a logical implication.

Because of its higher-order nature, quantification in the CoC is not restricted to terms and as a matter of fact, terms and types are not distinguished in the CoC. Therefore types themselves have types, and these "types of types" are special terms called sorts: $\{\mathsf{Prop}, \mathsf{Set}, \mathsf{Type}_{i,i\in\mathbb{N}}\}$. Now, the fact that a term t has type T in the CoC can be seen in two dual ways: that t is an object of type T, but also that t is a proof of proposition T. Of course, not every type should be seen as a proposition, for instance basic datatypes like integers and functions are traditional "program types". The sorts above are used to ensure a strict separation between informative types (data types, programs) and logical types (propositions, proofs): the former category of types have type Set, while the latter have type Prop. In particular, the sorts Prop and Set differ by the fact that Prop alone is *impredicative*, *i.e.* quantifying over propositions still yields a proposition[1]. Quantification over

[1] Early versions of Coq implemented an impredicative version of the sort Set, but it was discovered to be inconsistent therefore Set has since been made predicative.

Set yields more complex objects, whose type is Type_0. In fact, both Prop and Set themselves have type Type_0, and the sorts Type_i form a hierarchy of sorts reminiscent of Martin-Löf's universes U_n, where each Type_i has type Type_{i+1}, which allows one to define arbitrarily complex objects.

Another decisive feature of the CoC, which it inheritates from simply-typed λ-calculus, is the fact that there is a natural notion of *reduction* of terms. The rules of reduction in CoC form a confluent, strongly normalizing, system and a very important typing rule allows one to take advantage of this reduction: the *conversion* rule says that if a term t has type T, it also has type T' as long as T' and T have the same normal form. This brings computational reasoning in the typing system: some typing judgments can now simply be verified by computing a normal form. For instance, if one has a proof of $P((15 * (75 - 7))/12)$, it is also a proof of $P(85)$, $P(5 * 17)$ or $P(100 - 15)$[2].

Finally, the CoC was extended with inductive definitions by T. Coquand and C. Paulin [CP90, PM93], and then to coinductive definitions by E. Giménez [Gim96] in what is now known as the *Calculus of Inductive Constructions* (CIC). Inductive definitions allow one to easily define datatypes in an intuitive manner, what was essentially only possible through tedious second-order encodings in the CoC. We demonstrate the use of inductive definitions in the next section.

4.1.2 The Coq Proof Assistant

The Coq proof assistant is a system based on the CIC presented above: it revolves around a small critical kernel whose role is to typecheck CIC terms. If one is able to build a term t of type T, then one is guaranteed to have a (constructive) proof of T. Depending on whether T is a proposition or not, this shows that Coq can be used both to prove propositions and to write pure functional programs. Coq is therefore really adapted to the task of writing programs, specifications, and proofs that these programs verify their specifications, all in one single system.

Inductive definitions. Coq users do not manipulate CIC terms directly; instead Coq provides a specification language called Gallina and a set of top-level commands called *vernaculars*. For instance, the datatype of Peano integers can be defined by the following inductive definition:

```
Inductive nat : Set :=
| O : nat
| S : nat → nat.
```

In effect, this definition actually corresponds to four separate definitions:

[2] We suppose here that integers and arithmetic operations have been defined, we will see in the next subsection how this can be done.

- the definition of a type `nat` of type Set;
- two symbols, called the *constructors* of the inductive type `nat`: `O` of type `nat` and `S` of type `nat → nat`;
- an *induction principle* `nat_ind` of type:

```
∀P : nat → Prop,
  P O → (∀n : nat, P n → P (S n)) → ∀n : nat, P n.
```

This induction principle is a second-order formula expressing the traditional induction principle used to prove properties by induction on integers. It states that integers are exactly built by application of the constructors `O` and `S`. Such an inductive definition also has two internal consequences (due to the introduction of inductives in the CIC). The first one is that it is possible to use pattern-matching to deconstruct a object of an inductive type. For instance, we can define a "predecessor" function in the following way:

```
Definition pred (n : nat) :=
  match n with
  | O ⇒ O (* -1 is not a nat *)
  | S m ⇒ m
  end.
```

The system checks that the pattern-matching is exhaustive, *i.e.* that all constructors are accounted for. Syntatic extensions in Gallina make it possible to use complex, nested pattern constructs, as is usually done in functional languages. The second consequence of the definition of an inductive datatype is the ability to write recursive functions, *i.e.* fix-points on the structure of an inductive type. For instance, we can define the addition operation `plus n m` by induction on the structure of the first argument:

```
Fixpoint plus (n m : nat) {struct n} :=
  match n with
  | O ⇒ m
  | S n' ⇒ S (plus n' m)
  end.
```

This special kind of definition, using the `Fixpoint` keyword, is possible as long as the recursive calls are performed on objects which are *structurally* smaller than the original argument. In this case, n' is obtained by destructing n and is therefore structurally smaller than n. This analysis ensures that all functions defined in Coq are terminating, and this is one of the strongest constraint in the language. When the structural condition is not verified, there are alternative ways of defining recursive functions, we will see some of these tricks in the following chapters. As a final remark, Coq allows the use of decimal representation to denote constant `nat`'s, for instance `4` stands for `S(S(S(S 0)))`.

Let's add the logic. As we have seen in the last section, CIC is an extension of λ-calculus, and does not contain built-in constructs for logical reasoning besides universal quantification (and of course the `Prop` sort). It is well-known that the usual connectives of first-order logic can be encoded using second-order quantification, and inductive definitions can be used to perform a similar encoding. For instance, the conjunction `and A B` of two propositions is defined in the following way:

```
Inductive and (A B : Prop) : Prop :=
| conj : A → B → and A B
where "A ∧ B" := (and A B) : type_scope.
```

There is only one constructor, *i.e.* one way to build the conjunction `and A B`, and unsurprisingly this is by giving proofs for `A` and `B`. The induction principle generated:

```
and_ind : ∀A B P : Prop, (A → B → P) → and A B → P
```

is the usual second-order encoding of conjunction. The definition above also introduces a syntactic notation for the conjunction `and A B`, namely the traditional `A ∧ B`. Notations are a very convenient feature of Coq and complex notations can be defined for user-defined constructs. The disjunction of two propositions can be defined inductively in a similar manner, with two constructors corresponding to either branch of the disjunction, and is noted `A ∨ B`. The special propositions `True` and `False` are respectively defined by an inductive type with a single trivial constructor, and by the empty inductive type:

```
Inductive True : Prop := I.
Inductive False : Prop :=.
```

Note that the elimination principle for `False` is the *ex falso quodlibet*[3] principle `∀P : Prop, False → P`. The negation of a proposition `P` is simply defined as:

```
Definition not (P : Prop) := P → False.
```

and is denoted `~P`. Finally, the existential quantification is denoted `∃x : T, P` and is defined inductively as:

```
Inductive ex (A : Type) (P : A → Prop) : Prop :=
| ex_intro : ∀x : A, P x → ex P.
```

In particular, an axiom-free proof of `∃x : T, P` must use `ex_intro` and must provide a *witness* of type `T` which verifies `P`, which is the trademark of an intuitionistic logic.

[3] From a false proposition, anything follows.

Interactive proofs. Using the logical definitions above, we can express propositions and try to prove them. As explained already, proving a proposition `P` amounts to giving a term of type `P`. This method is not practical except for the easiest propositions, for instance the term:

```
fun A : Prop ⇒ fun H : A ⇒ H
```

where `fun` is the Gallina syntax for λ-abstractions, is a proof of the proposition `∀(A : Prop), A → A` . To prove more complex properties, Coq provides an interactive mode, called *proof mode*, that allows the user to interactively construct proofs through the use of a language of commands called *tactics*. In their simplest form, tactics mimic the application of traditional introduction and elimination rules in natural deduction systems, or right and left rules in sequent calculi *à la* Gentzen. For instance, let us detail a proof of a simple propositional tautology[4]:

```
Theorem or_not_and : ∀(A B : Prop), ~A ∨ ~B → ~(A ∧ B).
Proof.
```

The `Theorem` command is one of the many available vernaculars (`Lemma`, `Property`, ...) which introduces a new goal to prove. Coq switches to proof mode and displays the current state. At every moment in proof mode, the state is described by a sequence of subgoals, each subgoal being a list of hypotheses and a conclusion to prove under these hypotheses. Only the first subgoal is displayed by Coq, with the conclusion separated from the hypotheses by a double bar. After starting the proof of the theorem above, the current state is the following single subgoal[5]:

```
1 subgoal

============================
∀(A B : Prop), ~A ∨ ~B → ~(A ∧ B)
```

We start the proof by using the introduction rule for universal quantification, four times. We write this using the `intros` tactic and explicitely provide names for the introduced objects.

```
  intros A B H N.
```

Note that the fourth introduction uses the fact that `~(A ∧ B)` is actually defined as the implication `A ∧ B → False` . After applying this tactic, the subgoal becomes:

[4] This theorem is intuitionistically valid but note that the converse of `or_not_and` is not an intuitionistic tautology, but is only valid in classical logic.

[5] To distinguish proof states from Gallina and tactic inputs, we will always present Coq's output in proof mode in a framed box.

```
1 subgoal

A : Prop
B : Prop
H : ~ A ∨ ~ B
N : A ∧ B
============================
 False
```

We now perform eliminations of the conjunction `N` and disjunction `H`: both eliminations can be performed with the same tactic, called `destruct`. Destructing the conjunction with `destruct N as [NA NB]` yields two new hypotheses `NA : A` and `NB : B` and does not change the conclusion. Destructing the disjunction with `destruct H` yields two different subgoals where hypothesis `H` is respectively a proof of `~A` and `~B`.

```
destruct N as [NA NB]. destruct H.
```

```
2 subgoals

A : Prop
B : Prop
H : ~ A
NA : A
NB : B
============================
 False
```

This first subgoal can be proved by eliminating the implication in `H`, in other words by “applying” hypothesis `H`, which is done with the `apply H` tactic. The remaining conclusion is `A`, which is true by hypothesis `NA`, and the goal can be discharged with the tactic `assumption`.

```
apply H. assumption.
```

This clears the first subgoal and therefore the user is left with the second subgoal to prove.

```
1 subgoal

  A : Prop
  B : Prop
  H : ~ B
  NA : A
  NB : B
  ============================
   False
```

This one is proved in a similar manner, only this time the assumption used will be `NB`.

```
  apply H. assumption.
```

Since all the subgoals have been proved, the proof is finished and the system displays so:

```
Proof completed.
```

The last thing to do is to close the proof with the `Qed` command.

```
Qed.
```

```
or_not_and is defined.
```

This last step is not anecdotal: it checks that the term which was progressively constructed by the tactics indeed has the type of the theorem. This mechanism ensures that tactics can be implemented without formal restriction and that possible bugs in the tactics are "double-checked" at the end of the proof by the kernel. Thus, only the kernel is critical for the correctness of the proof assistant and it is important for such a system to limit critical areas to the smallest possible part. As a matter of fact, there are a great number of tactics, many of which are much more complex than the ones presented here: the proof above could typically be performed by a single tactic call. We will present such complex tactics and the techniques behind them in detail below in Section 4.2.

Equality proofs. We now turn our attention to the treatment of equality in the Coq proof assistant. As with logical connectives, equality is not built-in in the CIC but is defined inductively by the following predicate:

```
Inductive eq (A : Type) (x : A) : A → Prop :=
| refl_equal : eq A x x.
```

and can be used with the usual `=` notation. The induction lemma associated with this definition is the well-known Leibniz's principle:

```
eq_ind : ∀(A : Type) (x : A) (P : A → Prop),
  P x → ∀y : A, x = y → P y
```

and allows to replace a term `x` by an equal term `y` in any proposition `P`. For this reason, this equality is often called *Leibniz equality* in Coq, in particular in constrat to other setoid equalities which can be natural for some types[6]. From the definition of `eq` and `refl_equal`, it may seem that the only equalities which are provable in an empty context are of the form `x = x` for some `x`, but this is where the *conversion* rule that we introduced earlier comes into play: it can be used to prove that two terms which reduce to the same term are equal. For instance, one can build a proof of `4 = 4` by considering `refl_equal nat 4`, but it turns out that the normal form

[6] Consider the type of propositions Prop and the equivalence relation ↔ for instance, or function spaces and pointwise equality.

of `2 + 2 = pred 5` is precisely `4 = 4`, therefore by conversion `refl_equal nat 4` is a proof of `2 + 2 = pred 5`. Most tactics in Coq perform modulo some kind of conversion, and it is possible to simply apply `refl_equal` in order to prove a definitional equality. The tactic `reflexivity` is precisely a shortcut for this:

```
Remark p : pred (pred (12 + 35)) = 45.
Proof.
  reflexivity.
Qed.
```

```
p is defined.
```

There are many tactics that explicitely perform some form of reduction or normalization, Coq even provides a virtual machine [GL02] to quickly reduce terms to their normal form; we will present these capabilities later. Note that the reduction mechanism is not limited to terms in sort `Set`, it can be used on any term in the CIC and in particular it is completely legitimate to reduce propositional proofs. Nevertheless, it is often the case that we do not want to compute through proofs:

- proof terms are often big and therefore slow to reduce;
- there is no point in reducing (or more generally observing) proof terms because most of the time, we do not care what the proof of a proposition looks like, but just that there exists a proof[7].

In order to be able to separate between reducible terms and non-reducible ones, Coq provides an *opacity* mechanism. When completing a proof with `Qed` as we did earlier, we are also making the corresponding theorem *opaque* and preventing that it be reduced in the future. In order to finish a proof and keep it *transparent*, one can use the `Defined` command. We will see in later chapters that a fine management of opacity can be critical for the efficiency of an algorithm implemented in Coq. Note though, that proof terms are never erased, even for opaque lemmas, and can still be inspected. The fact that proof terms are kept is one feature of Coq which differentiates it from many other provers like Isabelle or HOL, and this is why the size of proof terms is problematic when automatically constructing proofs through tactics (see Section 4.2).

Other features. There are many other features in the Coq proof assistant that allow one to write formalizations or programs in a more natural or a more convenient way. We will encounter some of them in the remaining of this document, but we cannot give an exhaustive list. Some of the more interesting capabilities are:

[7] This principle, called *proof irrelevance*, is not part of the CIC and therefore is not enforced by Coq; it is consistent to add it as an axiom though.

- a “batch-mode” executable coqc to compile files which can then be loaded and imported into other files, which enables separate compilation;

- a module system by J. Courant [Cou97] and J. Chrząszcz [Chr03] similar to OCaml’s module system, which permits to write structured programs and structured implementations, we will use it extensively in the following chapters;

- an extraction mechanism developed originally by C. Paulin [PM89a, PM89b] and then by P. Letouzey [Let03, Let08] which allows to effectively extract programs from specifications to OCaml or Haskell. The distinction Prop/Set that we explained earlier is critical for this mechanism, since extraction erases propositional contents and keeps informative contents. In particular, this is the reason why Coq prevents any object in Set to be constructed from destructing an object in Prop;

- a system of coercions which allow a form of automatic subtyping through the definition of coercions between types;

- a mechanism to deal with ad-hoc (setoid) equalities, and rewriting of setoid equalities through functions declared as morphisms for such equalities (initially developed by C. Sacerdoti Coen [Coe04] and reimplemented by M. Sozeau [Soz09]);

- a variety of external tools such as a documentation generator coqdoc, a library validator coqchk, and an integrated development environment CoqIDE.

4.2 Automation Techniques for Interactive Proving

In the last section, we have seen examples of simple tactics. Most realistic proofs will use many more different and complex tactics, some of which performing a lot of automated reasoning. In this section, we present a survey of the different automation techniques available in a proof assistant like Coq and the relevant existing tactics.

Note that interactive provers in general ensure their correctness by following the so-called LCF-style approach: every proof must be checked by a small, trusted part of the system (in Coq’s case, the kernel). Thus, a complex decision procedure inplemented in an interactive prover shall not only decide if a formula is provable or not, but it must also generate an actual proof object, which can be checked by the prover’s kernel. This is in contrast to a system like PVS, where a new decision procedure can be added to the system as a "black box".

4.2.1 Customized Tactics

The first technique that we present is perhaps the most recent in Coq's history, but it has the great advantage of not requiring any external tool or special knowledge about the internal representation of CIC terms. It does not require any proof reconstruction either, because it is based on the tactic language. This technique uses a language called $\mathcal{L}$tac and developed by D. Delahaye [Del00] which provides combinators for tactics, called *tacticals*, allowing the definition of complex tactics inside the prover. We cannot list all the tacticals exhaustively but we will present the most salient capabilities of $\mathcal{L}$tac.

The base of the language is formed by combinators for chaining tactics (`;`), repeating tactics (`do`, `repeat`), error catching (`try`) or throwing (`fail`), branching (`||`, `first`), displaying terms, tactics and arbitrary messages (`idtac`), checking for progress or termination in a subgoal (`progress`, `solve`). These tacticals already allow a lot of interesting combinations, for instance the following:

```
Ltac dintros := repeat (intro; try (destruct 0)).
```

defines a new tactic `dintros` which does as many introductions as possible (using `repeat`), and for each object introduced, tries to destruct it if it is possible (`destruct 0` refers to the last introduced hypothesis by its index, thus with number 0). In our example proof in the last section, we could have started the proof with that tactic in order to introduce and destruct all hypotheses. Because both remaining subgoals can be proved by the same `apply H; assumption` combination, we could use chaining and prove the theorem in a single line:

```
Theorem or_not_and : ∀(A B : Prop), ~A ∨ ~B → ~(A ∧ B).
Proof.
  dintros; apply H; assumption.
Qed.
```

```
or_not_and is defined.
```

Even more interesting is the ability to manipulate terms in $\mathcal{L}$tac definitions: one can construct terms, reduce terms, deconstruct terms using a pattern-matching construct. Pattern-matching can also be used against the goal and the hypotheses, which makes it possible to write tactics that perform different tasks according to the shape of the goal and the available hypotheses. For instance, consider the following definition:

```
Ltac equal :=
  match goal with
```

```
| ⊢ ?x = ?x ⇒ reflexivity
| H : ?x = ?y ⊢ ?y = ?x ⇒
  symmetry; assumption
| H : ?x = ?y, H' : ?y = ?z ⊢ ?x = ?z ⇒
  transitivity y; assumption
| ⊢ _ = _ ⇒ idtac "No proof found."
| _ ⇒ fail "The goal is not an equality."
end.
```

This tactic `equal` tries to prove an equality and proceeds by matching the current goal, and then depending on the result performs the relevant action. If the goal has the form `x = x`, it just applies reflexivity. The second and third branch try to find, in the hypotheses, equalities related to the conclusion and to apply respectively symmetry or transitivity. The next-to-last branch just reports that the tactic did not succeed in proving the goal (but does not fail), while the last branch raises a failure because the goal is not an equality. This example gives a small idea of the expressivity of ℒtac; note in particular that the matching is non-linear since the same variable can appear twice or more in a pattern, and must be matched to the same term. ℒtac is even higher-order because tactics can be parameterized by tactics (and by parameterized tactics...) and can also be defined recursively.

Here are a few examples of complex tactics developed in ℒtac in the Coq standard library or in the community:

- in the `Reals` library, containing an axiomatization of reals, the specialized tactics `discrR`, `prove_sup` and `Rcompute` are built with ℒtac;
- in the specification of `OrderedType`'s, *i.e.* types with a total decidable order, there is a dedicated tactic `order` which tries to prove a goal using total order and equivalence properties, it proceeds by saturating the context with all possible consequences of the hypotheses until it finds a contradiction; it is actually complete for that fragment;
- in the `FSets` library of finite sets (which we describe, as well as an alternative, in Chapter 5), A. Bohannon contributed a very complex tactic `fsetdec` which discharges goals about set memberships and common set operations;
- in his book [Chl], A. Chlipala gives many concrete examples of ℒtac usage, in particular his "swiss knife" tactic called `crunch`;
- A. Charguéraud proposes an extended set of tactics and tactic notations to help perform a variety of tasks [Cha].

Even if it has its own limitations and can be quite inefficient, ℒtac is very convenient because of its expressiveness and above all the fact that it does not require an external tool or "hacking" in the Coq sources, which, as we will see, is the biggest inconvenient of the other techniques.

4.2.2 Built-In Procedures

The vast majority of tactics available in Coq are not defined in external contributions or $\mathcal{L}$tac files, but are simply implemented in the Coq sources and are compiled and shipped with the proof assistant. All atomic tactics like `intro` and so on are actually built-in tactics and are implemented in the OCaml sources with the remaining of the system, but there are also several very useful tactics which do not perform atomic tasks, but a complex proof search.

tauto. The tactic `tauto` by C. Muñoz [Mn94] implements a decision procedure for intuitionistic propositional calculus based on Dyckhoff's contraction-free sequent calculi [Dyc92]: it automatically proves any goal which is intuitionistically valid (e.g. our theorem `or_not_and` could have been discharged by a simple call to `tauto`). It is also available as a simplifier called `intuition` which performs the same search tree as `tauto`, clears as many branches as possible and returns the simplified goals to the user, which can be very useful in practice.

omega. Another ubiquitous tactic in Coq is `omega`, which was implemented by P. Crégut after a decision procedure by W. Pugh [Pug92]. It is a decision for Presburger arithmetic which automatically solves quantifier-free formulae whose atoms are equalities, disequations or inequalities on natural or relative integers. Though `omega` is theoretically incomplete, it rarely happens in practice and this tactic is used a lot in any development dealing with arithmetic.

congruence. The `congruence` tactic was developed by P. Corbineau [Cor06] and implements a decision procedure for the theory of equality modulo the theory of constructors (*i.e.* injectivity of constructors, and discrimination of different constructors of the same datatypes). The procedure tries to prove the goal if it is an equality and to derive a discriminable (hence false) equality otherwise.

auto. The tactics `auto` and `eauto` perform an automatic backward proof search in a manner very similar to Prolog. They use a database of lemmas, called *hints*, as well as the hypotheses in the current context, and try to apply them eagerly and find a chain of lemmas proving a goal. `auto` is perhaps the most popular proof search tactic in Coq and tactics that generate a lot of subgoals like `induction` or `destruct` are often chained with `auto`.

There exists other built-in tactics which are not performing proof search *per se*, but are nonetheless quite complicated and can replace a lot of tedious manual manipulations, for instance the `autorewrite` or the `inversion` tactics.

The tactics above are therefore implemented in a standard programming language and there is of course no restriction as to how they perform their proof search. However, once they have found a proof (if any), they need to construct a proof term because, like the other tactics, they are just used to construct proof terms and are not trusted by the kernel. This reconstruction phase can take different forms: `tauto` and `auto`, for instance, build a complete proof term corresponding to the proof they have found ; `omega` and `congruence` also reconstruct a term from the proof found but use a variety of predefined ad-hoc lemmas in an attempt to simplify the reconstruction and also to obtain smaller proof terms.

Note that, in the most recent versions of Coq (≥8.2) and OCaml (≥3.11.0), it is possible to dynamically load ML plugins in a Coq session. Therefore, one can implement such a built-in procedure as a plugin and it can be distributed and used without having to recompile everything along with the Coq sources. Nevertheless, implementing one's own decision procedure and term reconstruction requires to use Coq as an API and therefore requires some amount of knowledge about the internal representation of proofs, terms and tactics. This is a much bigger effort than learning $\mathcal{L}$tac for instance.

4.2.3 External Tools

Another possible approach for the creation of an automation tactic is to use an external state-of-the-art decision procedure. The proof reconstruction phase requires the external tool to be able to return *proof traces* of its proof search, *i.e.* data which justifies the result claimed by the tool. Work must then be done in the interactive prover in order to reconstruct a suitable proof object from the output of the external tool. For instance, Weber and Amjad [WA09a] have successfully integrated two leading SAT solvers, zChaff [MMZ$^+$01] and MiniSat [ES04], with Higher Order Logic theorem provers. Integrations of resolution-based provers have also been realized in Coq [BHdN02, BDD07] and Isabelle [MQP06a]. The main advantage of this approach is the ability to use a very efficient external tool. Its main shortcoming is that the tool must be able to produce proof traces, which is not that common, and the reconstruction of a proof term from proof traces can be quite difficult to perform efficiently (see for instance the considerations in [WA09a]).

This approach is actually a special case of the previous one (Section 4.2.2), since nothing prevents a built-in procedure from using an external tool under the hood. It allows the use of faster, state-or-the-art procedures, but the proof traces may not be very well adapted to the proof reconstruction phase, whereas a procedure specifically developed for a given proof assistant can lead to an easier (maybe even smaller) proof term.

4.2.4 Traces and Reflection

The crux of the last two approaches, *i.e.* using a built-in procedure or an external tool, is the reconstruction of a proof term from the proof search. A built-in procedure could technically build a proof term directly during the proof search but this is probably not the most efficient thing to do, most procedures will go through some internal form of traces and reconstruct the term at the last moment; users of external tools have no choice whatsoever (unless the tool can output a Coq proof term directly, like Zenon [BDD07], but this is very rare) and need to perform reconstruction from some proof traces.

So far, we had implicitely assumed that the reconstruction was a meta procedure (*i.e.* not expressed in the prover) that, given some trace π, would create a CIC term t to be sent back to the prover for typechecking. In that sense, the reconstruction acts as an oracle which gives an hopefully adequate term to the prover. There is an alternative approach, which we now present: the so-called *proof by reflection* [Bou97]. In this setting, the reconstruction will be a function in the prover's logic and we will use the reduction mechanism and the conversion rule to execute this function during typechecking.

Reflection. Suppose we have a datatype `S`, and a predicate `P : S` → **Prop** on elements in this datatype. Suppose we have an oracle (the external procedure) which, given an `s : S`, will look for a proof of `P s` and, if any, will return some proof traces to justify this result. We assume that the proof traces can be represented by a datatype `T` in the prover, which is typically the case. Now, in order to use the oracle's proof traces in the prover, we just need the following:

- a function `check : S → T → bool` implemented in the prover and returning a boolean[8] such that `check s t` checks if the trace `t` is a good justification of the fact that `s` has property `P`;
- a proof, called a *reflection principle*, that the function `check` is correct:

  `check_correct : ∀(`s` : S) (`t` : T), check `s t` = true → P `s`.`

The function `check` is similar to proof reconstruction but does not construct anything, it just returns a boolean value to denote whether the traces were adequate or not. The reflection principle then relates the computational behaviour of `check` to its propositional meaning and proves that it is sufficient to check the result of `check` in order to verify the traces. Given a concrete `s` and some traces `t` returned by the oracle, the proof of `P s` is simply:

[8] Booleans in Coq are just a type with two values `true` and `false`. It has sort Set and should not be confused with the type of propositions Prop.

(check_correct s t (refl_equal _ true)) : P s

where the call of `refl_equal` is used to force reduction and verification that `check s t` indeed reduces to `true`. In comparison to standard proof reconstruction, we note that:

- the proof term is not explicitely reconstructed; typically, part of the work is performed by the `check` function, while remaining part is performed in the proof of `check_correct`, and is therefore factorized once and for all;
- the proof search will be faster because it does not have to reconstruct a proof term afterwards, but this is compensated by the fact that typechecking the proof now includes a computation;
- the size of the proof terms is now proportional to the size of the traces (and the original object) whereas reconstructed proof terms can be much bigger than the traces.

Levels of detail. Given one particular problem for which we would like to use the reflection technique described above, a natural question which arises is: what should the traces actually be like? There are indeed typically a broad range of choices in the amount of detail that the traces should include. To illustrate this fact, let us take a simple concrete example[9]: suppose that we are interested in proving that some Peano integers are *composite*, *i.e.* that they are not prime, using an external procedure. If for instance we are interested in the number 91, here are some of the answers that we might get from the procedure:

- Yes, 91 is composite.
- Yes, 91 is divisible by 7.
- Yes, 91 is divisible by 7 and the quotient is 13.
- Yes, $91 = 13 \times 7$, indeed 7×3 equals 21, carry the 2, 7×1 is 7, plus 2, makes 91.

Figure 4.1 schematizes this situation: it represents the possible proof traces on a scale from the most detailed (on the left) to the less detailed (on the right). The relevance of using reflection is in inverse proportion to the level of detail of the traces: in the leftmost case, the external procedure has produced a proof term and therefore there is no need for reflection at all; conversely, in the rightmost case, the trace is empty and the `check` function must do everything from scratch, which means that the external

[9] This example was drawn from G. Dowek's excellent popular science book [Dow08].

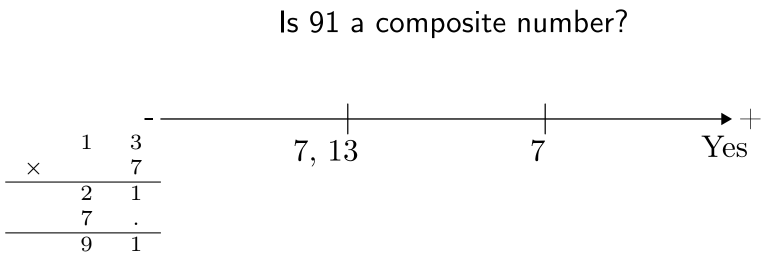


Figure 4.1: One problem, many solutions: a scale of proof traces

tool is basically useless and could be bypassed[10]. In the average cases, the procedure returns pieces of information that the `check` function can verify: try to divide 91 by 7 in one case, check that 7 by 13 is 91 in the other case.

In general, fine-grained traces make for an easier proof reconstruction but require a substantial amount of work in the decision procedure, including justifying steps that are often implicit in an efficient implementation. On the other hand, coarse-grained traces make proof reconstruction much harder since all implicit steps must be implemented in the proof assistant in the reflection principle. Tactics which are somewhere between the two extreme cases on this scale are usually called *semi-reflexive* and most tactics using reflection fall in that category. Examples of this intermediate approach are Corbineau and Contejean [CC05] and Contejean *et al.* [CCF$^+$]'works on integrations mixing traces and reflection. There also exists semi-reflexive versions of the `tauto` tactic, called `rtauto`, and of `omega`, called `romega`.

Tactics which do not use an external procedure at all are called *fully reflexive*. For instance, the tactics `ring` [GM05] and `field` [DM01], which respectively solve expressions on ring and field structures, are built along this reflection mechanism. The main advantage of the fully reflexive approach is the size of the generated proof term, which only consists in one application of the correctness property. The trade-off is that typechecking the proof term includes executing the decision procedure, therefore reflection can be used favourably in cases where the proof traces would not be comparatively simpler than the proof search itself. For instance, suppose we were interested in prime numbers instead of composite numbers: since there is no "simple" justification that a number is prime, it would be a good idea to use a fully reflexive procedure.

[10] One case where it would still be useful to run the external tool is when it runs at least an order of magnitude faster than the reflexive function `check`. In such a case, it makes sense to run the external procedure first, simply to know if it's worth running the reflexive one.

Reification. Until now, we have only dealt with properties on a concrete datatype, namely natural integers. In general, we might want to apply the reflection technique to a more general class of formulae, for instance all first-order formulae. This means the type `S` is now the sort of all propositions Prop, and it becomes impossible to write a function `check : S → T → bool` (remember that informative datatypes like `bool` cannot be constructed by deconstructing propositional objects). In such a case, we need a concrete, intermediate, representation of formula, *i.e.* an informative type `form`, along with an interpretation function `interp : form →` **Prop** . The reflection function and its correction lemma then become:

```
check : form → T → bool
check_correct : ∀(f : form) t, check f t = true → interp f
```

and in order to prove a formula `F :` Prop, the system cannot directly use the reflection principle but must infer a concrete `f : form` such that `interp f = F`, or `interp f → F` at the very least. The construction of this object `f` cannot be expressed in the prover's logic and is therefore a meta procedure; it is called *reification* and must be performed by an external oracle[11]. In particular, we will see in Chapter 6 that it introduces a "hole" which prevents a reflexive procedure to be formally complete.

4.3 Towards a Reflexive SMT Kernel

We have just presented a variety of techniques to implement automation tactics in the Coq proof assistant. Our goal is to integrate in Coq the kernel of our SMT solver, as presented in Chapters 2 and 3, in order to provide a tactic which effectively combines propositional, equality and arithmetic reasoning. We chose to use the fully reflexive approach in order to achieve this integration, for the following reasons.

First of all, as explained in Section 1.2, we are especially interested in proving proof obligations which arise from analysis of annotated programs, or more generally to discharge goals in usual Coq proofs. Our experience with our own prover Alt-Ergo is that these formulas' difficulty lies more in finding the pertinent hypotheses and lemmas' instances than in their propositional structure or the theory reasoning involved in their proofs. Consequently, these problems become rather easy as soon as we know which hypotheses and instances are sufficient for the proof, and we can thus solve formulae in this ground fragment by pure reflection. Moreover, Coq is particularly well suited for this approach because its formalism includes a full programming language, whose evaluation has been recently dramatically improved by an optimized bytecode-based virtual machine.

[11] Thanks to their ability to construct, match and destruct terms, $\mathcal{L}$tac tactics can typically be used to perform this reification step. This avoids, in principle at least, the

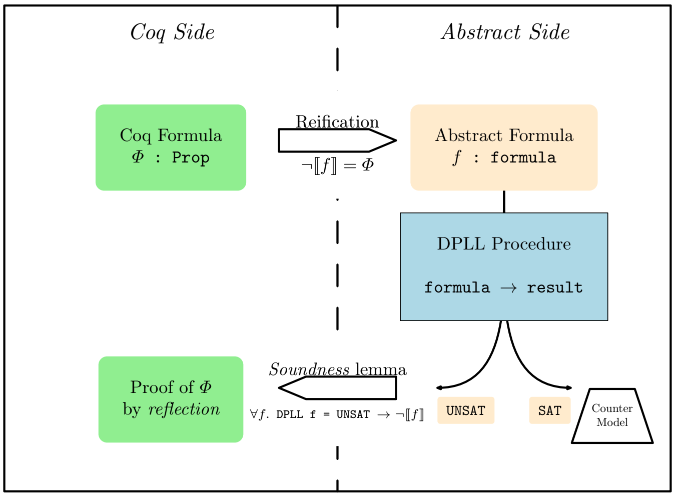


Figure 4.2: An overview of our reflexive tactic

Second, we are interested in a tactic that could be used by all Coq users, and it should be available *out-of-the-box* with the system, without requiring the installation of an external solver like Alt-Ergo. Also, such an external dependency is a concern for maintainability since the proof reconstruction mechanism will be very dependent of the exact format of the proof traces: it would have to be kept up-to-date with the changes in the external tool, and would have to be totally revamped in order to support another external tool[12].

Third, it is not easy to instrument an SMT solver to generate proof traces, in particular in underlying decision procedures such as the congruence closure algorithm and Fourier-Motzkin, and to choose the adequate level of detail. Without any traces on the reasoning of underlying theories, the reflection principle would become nearly has hard as the fully reflexive procedure; with details on the reasoning of underlying theories, proof traces and proof objects could get quite large and it would be a problem for a prover like Coq which saves proof objects in typechecked files. With a fully reflexive tactic, we ensure a proof term which is linear in the size of the goal.

Finally, we have formalized in Coq the algorithms and proofs presented in Chapters 2 and 3 in order to formally verify these proofs, and it is natural to try and take advantage of this formalization in order to use these algorithms in Coq using reflection. An overview of our reflexive tactic's architecture is given in Figure 4.2.

need any need for an external OCaml procedure.

[12] Although there is an ongoing effort in the SMT community to design a standard, common, format for SMT proof traces, no such format has been adopted yet.

Outline. The following chapters are devoted to the presentation of the different parts of this Coq reflexive tactic. We start in Chapter 5 by defining a library of first-class finite sets and finite maps which is used intensively in our development. Chapter 6 presents the propositional solver and Chapter 7 extends it with an original lazy CNF conversion mechanism. The extension to SMT and the development of the congruence closure algorithm is described in Chapters 8 and 9. Finally, we show how to instantiate our congruence closure with a theory of integer linear arithmetic in Chapter 10.

CHAPTER 5

A Coq Library of First-Class Containers

Contents

As a programming language, it is natural to endows Coq with libraries of generic data structures. Indeed, mainstream programming languages usually come with libraries to manipulate these structures which are widely used: lists, finite sets, association tables, etc. For instance, C++ programmers can rely on the STL [SL95](*Standard Template Library*), whereas OCaml programmers are provided with a fair number of modules (including lists, queues, sets and maps, hashtables, ...) in the OCaml standard library. The genericity of these data structures, that is, the fact that they can be used to hold elements of any type, is ensured in different ways depending on the programming language: *polymorphism* in languages of the ML family, *templates* in C++ or *generics* in Java.

For their implementation to be efficient, some data structures require certain properties on the elements they can contain, such as a comparison or a hash function. This kind of genericity, called *ad-hoc polymorphism*, is made possible by the use of *functors* in OCaml and *type classes* in Haskell [WB89]. Even if these two paradigms can be used to solve a similar design issue, they are fundamentally different and both have their advantages and their shortcomings [WM06]. For a few years, Coq has featured a full-blown module system similar to OCaml's [Chr03] and P. Letouzey and J-C. Filliâtre used it in order to develop a comprehensive library of finite sets and finite maps [FL04], called `FSets`. Such structures are very important for developing our SMT solver kernel: in Part 1, we have used sets of literals, maps for union-find structures, maps to sets of terms for the Γ data structure, etc. We have used this `FSets` library in developing the tactic presented in this document and have been confronted with issues which were inherent to the module system. Since Coq has been recently enhanced with a type class system based on dependent records [SO08], we decided to build on this new functionality and reimplement the existing `FSets` library using type classes. We present this library in detail in this chapter.

Section 5.1 quickly presents Coq's type class system, as well as the problems which motivated our work. We then introduce the cornerstone of our libary, ordered types, in Section 5.2, before describing the actual interfaces of finite sets and dictionaries (Section 5.3). We follow by giving a few concrete instantiations of these structures, before comparing in detail our library with the modular version in Section 5.5.

5.1 Preliminaries and Motivations

5.1.1 Type Classes

In this section, we present Coq's new type class system and its basic features. For a more detailed and involved description, the interested reader can refer to [SO08].

A type class can be seen as a way to package a number of definitions and properties together, much like a record[1]. Classes can be parameterized by types or other constructions, and one can for instance define the class of types which are equipped with a decidable equality in the following way:

```
Class decidable (A : Type) := {
  eq : A → A → bool;
  eq_dec : ∀xy, eq x y = true ↔ x = y
}.
```

This `decidable` class is parameterized by a type `A` and contains two fields: a boolean equality on this type `A` and a proof that this equality test really decides logical equality. Objects of type `decidable T` for a type `T` are called *instances* and must be defined in a special way using the **Instance** keyword. This is how we can define an instance for the type of booleans:

```
Definition bool_eq (x y : bool) := if x then y else negb y.
Property bool_eq_dec : ∀xy, bool_eq x y = true ↔ x = y.
Proof. .... Qed.
Instance bool_dec : decidable bool :=
  { eq := bool_eq; eq_dec := bool_eq_dec }.
```

An instance's fields can also be initialized directly or proved interactively at the time of the definition. Type classes reach their full potential with the conjunction of two mechanisms:

- the ability to define objects parameterized by type classes and use these objects without explicity providing these parameters;
- a mechanism for automatically inferring type class instances using all instances already defined by the user.

For instance, one can prove the following lemma for any type which has an instance of `decidable`[2]

```
Lemma decides_eq `{decidable A} :
  ∀(x y : A), x = y ∨ x ≠ y.
Proof. .... Qed.
```

This lemma is parameterized by a type `A` and an instance of `decidable A`, but both parameters are declared as implicit using the special `{...}` delimiters. The backquote character ` is just a way to ask Coq to automatically generalize the lemma on the fresh type `A`. When we subsequently use this lemma by applying it to two terms of some type `B`, an instance of `decidable B` is automatically searched and inferred using already defined

[1] suitably, Coq's type classes are implemented using dependent records.

[2] bear in mind that Coq's logic is intuitionistic, therefore this lemma really means that equality on type `A` is decidable.

instances. For instance, one can write the term `decides_eq true false` which is well-typed and will implicitely and automatically use the `bool_dec` instance provided above.

Automatically inferring instances becomes particularly useful when one defines families of instances, or instances parameterized by other instances. To give an example, we can write an instance for any product `A × B` given instances for the types `A` and `B`.

```
Instance prod_dec ‘{decidable A, decidable B} :
  decidable (A × B) := {
  eq := fun x y ⇒ eq (fst x) (fst y) &&
                  eq (snd x) (snd y);
  eq_dec := ...
}.
```

The system can then infer instances for any product of decidable types, for example with `bool_dec` and `prod_dec`, an instance for the type `bool × bool × bool` can be used automatically:

```
Check (decides_eq (true, (false, true))
                  (false, (false, true))).
```

Let us conclude this introduction to type classes by noting that it is possible to build hierarchies of classes, and a system of automatic coercions guarantees that an instance of some class can be used as an instance of its sub-classes. We will demonstrate this feature later in Section 5.2.

5.1.2 Motivations

In the light of the features provided by the type class system, we can explain the reasons why we turned to this system instead of using the already available module-based `FSets`. These were the motivations in starting this reimplementation of a containers library.

Automatic instantiation. In our development, we manipulate sets of numerous different types, including sets of sets, and for each new element type, we need to create a finite set module for this type. This creation must be performed manually by applying the adequate functor to an ordered type module (packing the element type and a comparison function together[3]). Namely, given ordered type modules `Int`, `IntPair`, `BoolList` for integers, pairs of integers and lists of booleans, one must write[4]:

```
Module IntSet := FSetList.Make Int.
Module IntPairSet := FSetList.Make IntPair.
Module BoolListSet := FSetList.Make BoolList.
```

[3] the corresponding signature is given in Section 5.2.1

[4] `FSetList.Make` is a functor creating a module of finite sets based on sorted lists.

in order to be able to use sets on these three kinds of elements. This may look like a lesser evil, but one quickly finds himself instantiating not only the `FSetList.Make` functor, but also other functors creating useful properties on ordered types and on finite sets, which are invaluable to start working with the data structures created above.

```
Module IntFacts := OrderedType.OrderedTypeFacts Int.
Module IntSetEqProps := FSetEqProperties.EqProperties IntSet.
Module IntSetProps := IntSetEqProps.MP.
Module IntSetFacts := IntSetEqProps.MP.Dec.FM.
Module IntPairFacts := OrderedTypeFacts IntPair.
...
```

This sort of definitions, which every `FSets` user has encountered, rapidly becomes tedious to read and maintain. Moreover, functor applications are not free and it is not uncommon to spend a couple of seconds solely on instantiating these various objects.

Overloading. Because the module system does not offer any overloading mechanism[5], one must refer to members of a module by qualifying their names with the module's name. In our example case, this means that every usage of a function, a lemma or a type provided by these modules (`IntSet, IntPairSet, IntSetProps,` ...) must be properly qualified. This quickly makes proof scripts and definitions verbose and hardly readable. One often ends up giving very short names to these modules (`IS, IPS, ISP, ...`) and then the script loses in clarity what it gained in compactness. Through the use of implicit type class arguments, the type class system does not require such qualification of identifiers and provides a real overloading of types and operators.

Performance and modularity. In order to ensure a good modularity in our development, some parts of the system ought to be parameterized by modules which, among other things, bring types and data structures on these types. For instance, it is not uncommon for an OCaml programmer to write signatures like this one:

```
(* some abstract type *)
type t
(* finite sets of elements of type t *)
module TS = Set.S with type elt = t
(* finite maps indexed by elements of type t *)
module TM = Map.S with type key = t
...
```

When parameterizing our development in a similar fashion in Coq, we encountered a performance issue related to modules' instantiation and type-checking. In practice, the functors' applications with such signatures were

[5] notations can help making up for the absence of overloading, but are limited and can be fragile in general.

taking unreasonable time: our topmost functor would require about 15 seconds. Although this seemed to be an implementation issue[6] rather than a theoretical limit with modules, it can be a real showstopper for an application based on modules, and type classes do not suffer from the same limitation.

First-class values. To further stress the previous point, type class instances in Coq are actually first-class values, and therefore the cost of an instantiation is reduced to typechecking the argument (since it is really just applying a function to an argument). This means that one could possibly perform "interactive" instantiations of a procedure parameterized by type classes. An example that arose in our work was that of a reflexive tactic: such a tactic must be invoked interactively and each time, a new instantiation of a parameterized procedure had to be made depending on the context where the tactic was called. This kind of dynamic instantiation is not possible with a functorized procedure since the instantiation time at each invocation would be prohibitive.

Amongst these motivations, the first two are inherent to modules and class types, whereas the last two are more specific to a given implementation, in our case the Coq proof assistant v8.2. Although the third one was the actual initial reason why we started using typeclasses, the first two points proved important enough in practice to justify choosing one paradigm over the other.

5.2 Ordered Types

To be implemented in an efficient way, structures of finite sets and finite maps require that the elements be equipped with a total decidable order. In this section we show how we formalize the class of such types.

5.2.1 `OrderedType`

An ordered type is a type which has an equality (an equivalence relation), a strict order (a transitive irreflexive relation) and such that these relations are decidable. Coq already provides a type class named `Equivalence` for equivalence relations, as well as the notations `x === y` and `x =/= y` for equalities and disequalities with respect to equivalence relations. We define the class of strict orders modulo an equivalence relation. This class is parameterized by the type of elements, the equivalence relation and the order relation:

```
Class StrictOrder {A} lt eq {Equivalence eq} := {
  StrictOrder_Transitive :
    ∀(x y z : A), lt x y → lt y z → lt x z;
```

[6] in particular, using finite sets implemented as AVL trees instead of sorted lists would multiply this time by four without apparent reason.

```
  StrictOrder_Irreflexive :
    ∀(x y : A), lt x y → x =/= y
}.
```

Note that only the order and equality are explicit arguments of this class.

We now look at the existing implementations of ordered types: the `FSets` library brings two different signatures for ordered types, respectively in the modules `OrderedType` (Figure 5.1) and `OrderedTypeAlt` (Figure 5.2).

```
Inductive Compare {A} lt eq x y :=
| LT : lt x y → Compare lt eq x y
| EQ : eq x y → Compare lt eq x y
| GT : lt y x → Compare lt eq x y.

Parameter t : Type.
Parameter eq : t → t → Prop.
Parameter lt : t → t → Prop.
(* equivalence axioms for eq *)
...
Axiom lt_trans : ∀xyz, lt x y → lt y z → lt x z.
Axiom lt_not_eq : ∀xy, lt x y → ~ eq x y.
Parameter compare : ∀xy, Compare lt eq x y.
```

Figure 5.1: Existing `OrderedType` module

```
Inductive comparison := Lt | Eq | Gt.

Parameter t : Type.
Parameter compare : t → t → comparison.

Parameter compare_sym : ∀xy,
  compare y x =  match compare x y with
                  | Eq ⇒ Eq | Gt ⇒ Lt | Lt ⇒ Gt
                  end.
Parameter compare_trans :
  ∀cxyz, compare x y = c →
    compare y z = c → compare x z = c.
```

Figure 5.2: Existing `OrderedTypeAlt` module

`OrderedType` brings a type `t`, an equivalence relation `eq` and a strict order `lt` on `t`, as well as the corresponding properties. The decidability of these relations is given by the `compare` function which is completely specified by its return type: the `Compare` inductive datatype. More precisely, the `compare` function performs the comparison of two elements but also returns a proof of the relation between these elements. This formalization is quite convenient to use: in particular, when reasoning by case analysis on the comparison between two elements, the hypotheses corresponding to each branch are naturally added to the context. A possible inconvenient, however, is the fact that the `compare` function is not purely computational, but *informative*, and it can become an issue and be a source of inefficiencies when it is used very frequently in an algorithm. Alternatively, to ensure a separation between computations and proofs, `OrderedTypeAlt` revolves around a pure comparison function `compare`, whose return type `comparison` is the 3-value type `Lt | Eq | Gt`. Unfortunately, this function's specification through properties of symmetry and transitivity is really tedious to reason with.

In order to keep the best of both alternatives, we choose a purely computational comparison function, but specify it with the following inductive definition:

```
Inductive compare_spec {A} eq lt (x y : A) :
  comparison → Prop :=
| compare_spec_lt : lt x y → compare_spec eq lt x y Lt
| compare_spec_eq : eq x y → compare_spec eq lt x y Eq
| compare_spec_gt : lt y x → compare_spec eq lt x y Gt.
```

Unlike `Compare`, this inductive is not the return type of the comparison function, but it relates each comparison value to the corresponding adequate hypothesis. It is then enough to prove that all the function's results belong to this relation for the function to be correct: namely, for a function `f` of type `T → T→ comparison` to be deciding some equality ≡ and order ≺ on `T`, it is sufficient and necessary to have:

```
 ∀xy, compare_spec ≡ ≺ x y (f x y).
```

Using such a specification, we are now able to write the class `OrderedType` of ordered types:

```
Class OrderedType (A : Type) := {
  _eq : relation A;
  _lt : relation A;
  OT_Equivalence :> Equivalence _eq;
  OT_StrictOrder :> StrictOrder _lt _eq;
  compare : A → A → comparison;
  compare_dec :
    ∀xy, compare_spec _eq _lt x y (compare x y)
}.
```

This class is parameterized by the type `A` of elements and contains the equality and strict order relations. Subclasses `Equivalence` and `StrictOrder`, introduced by `:>`, are used to specify these relations. The last part is the comparison function and its specification, which are given as explained above. This version is as easy to use as the original despite the purely computational return type of `compare`. Indeed, in a context where `compare a b` appears, it is enough to invoke the tactic `destruct (compare_dec a b)` in order to perform case analysis on this comparison: `compare a b` is then replaced in each branch by its value (`Eq`, `Lt` or `Gt`) and the corresponding hypothesis is added to the context. In this regard, the `compare_spec` inductive is similar to the reflexive "views" of the SSReflect extension [GM08].[7]

Once the class for ordered types is defined, numerous useful lemmas (like the fact that the order relation is a morphism for equality) and notations are established and can be used for any ordered type. The following table summarizes the available notations and the corresponding "views" for non-propositional objects:

Notation	Meaning	View
`x === y`	x equal to y	
`x =/= y`	x not equal to y	
`x <<< y`	x smaller than y	
`x >>> y`	x greater than y	
`x =?= y`	`compare x y`	`compare_dec`
`x == y`	`true` iff `x =?= y` returns `Eq`	`eq_dec`
`x << y`	`true` iff `x =?= y` returns `Lt`	`lt_dec`
`x >> y`	`true` iff `x =?= y` returns `Gt`	`gt_dec`

5.2.2 Special Equalities

When writing a piece of code which is parameterized by an ordered type, it is very frequent to require a certain type to be ordered with the constraint that the equality relation be some special equality, typically Leibniz equality. The module system allows one to express such a constraint by specializing the signature: `OrderedType with Definition eq := ...`. Unfortunately, this kind of constraints cannot be expressed with type classes unless the part we wish to specialize is a parameter of the type class and not a field. To make the use of specific equalities possible, we introduce a special class `SpecificOrderedType`, which is parameterized by the equivalence relation, and also show that any instance of this class is also an instance of `OrderedType`.

```
Class SpecificOrderedType (A : Type)
  (eqA : relation A) (Equivalence A eqA) := {
  SOT_lt : relation A;
  SOT_StrictOrder : StrictOrder SOT_lt eqA;
  SOT_compare : A → A → comparison;
  SOT_compare_spec :
    ∀xy, compare_spec eqA SOT_lt x y (SOT_compare x y)
}.
```

[7] this discussion assumes Coq v8.2 ; Coq's next version is going to introduce a mixed signature taking advantage of type classes and a specification *à la* `compare_spec`, inspired by this one.

```
Instance SOT_as_OT ‘{SpecificOrderedType A eqA equivA} :
  OrderedType A := {
  _eq := eqA;
  _lt := SOT_lt;
  ...
}.
```

We also add a notation `UsualOrderedType` to denote the particular and yet frequent case where the wanted equality is Leibniz equality. These ordered types with specific equalities will come in handy when defining containers in Section 5.3.

5.2.3 Automatic Instances Generation

After classes, generic lemmas and definitions have been defined, we declare instances of `OrderedType` for all basic types and usual type constructors. When possible, we declare instances of `UsualOrderedType`, including for type constructors[8]. The library provides instances for Peano integers, binary integers (whether positive, natural or relative), rationals, booleans, lists, products, sums and options. At this point, generic functions on ordered types can therefore be used on all combinations of these types and type constructors without manual intervention, thanks to the automatic inference of type classes:

```
Goal ∀(x y : ((nat × bool) + (list Z × Q))), x === y.
```

To typecheck this goal, an instance of `OrderedType` is inferred for the type of `x` and `y`. In particular, an effective comparison function is available to compare elements of this type.

In practice however, a type like the one above will typically be defined directly as a two-branch inductive:

```
Inductive t :=
| C1 : nat → bool → t
| C2 : list Z → Q → t.
```

The type classes system cannot automatically infer instances for such inductive types, but we have implemented a new vernacular command in OCaml which can handle such cases automatically. This command is invoked by `Generate OrderedType <type>`, takes an inductive type as argument and tries to generate the equality, the strict order relation, the comparison function and all the mandatory proofs, before declaring the corresponding instance. To do that, it potentially uses other instances already defined and available in the context. In the generated order relation, constructors are ordered arbitrarily, and parameters on a single constructor

[8] for instance, if `A` and `B` are ordered types for Leibniz equality, then so are their product and their sum.

are ordered lexicographically[9]. For instance, when invoking the command for the type `t` above, the following definitions are performed automatically :

```
Inductive t_eq : t → t → Prop :=
| t_eq_C1 : ∀(x1 y1 : nat) (x2 y2 : bool),
  x1 === y1 → x2 === y2 → t_eq (C1 x1 x2) (C1 y1 y2)
| t_eq_C2 : ∀(x1 y1 : list Z) (x2 y2 : Q),
  x1 === y1 → x2 === y2 → t_eq (C2 x1 x2) (C2 y1 y2).

Inductive t_lt : t → t → Prop :=
| t_lt_C1_1 : ∀(x1 y1 : nat) (x2 y2 : bool),
  x1 <<< y1 → t_lt (C1 x1 x2) (C1 y1 y2)
| t_lt_C1_2 : ∀(x1 y1 : nat) (x2 y2 : bool),
  x1 === y1 → x2 <<< y2 → t_lt (C1 x1 x2) (C1 y1 y2)
| t_lt_C1_C2 : ∀(x1 : nat) (x2 : bool) (y1 : list Z) (y2 : Q),
  t_lt (C1 x1 x2) (C2 y1 y2)
| t_lt_C2_1 : ∀(x1 y1 : list Z) (x2 y2 : Q),
  x1 <<< y1 → t_lt (C2 x1 x2) (C2 y1 y2)
| t_lt_C2_2 : ∀(x1 y1 : list Z) (x2 y2 : Q),
  x1 === y1 → x2 <<< y2 → t_lt (C2 x1 x2) (C2 y1 y2)
```

and this comparison function is generated:

```
Definition t_cmp (x y : t) :=
  match x with
  | C1 x1 x2 ⇒
    match y with
    | C1 y1 y2 ⇒
        match x1 =?= y1 with
        | Eq ⇒ x2 =?= y2
        | Lt ⇒ Lt
        | Gt ⇒ Gt
        end
    | C2 _ _ ⇒ Lt
    end
  | C2 x1 x2 ⇒
    match y with
    | C1 _ _ ⇒ Gt
    | C2 y1 y2 ⇒
        match x1 =?= y1 with
        | Eq ⇒ x2 =?= y2
```

[9] but note that the command should typically be used in cases where any well-defined order relation is suitable, not unlike the `Pervasives.compare` polymorphic comparison in OCaml.

```
        | Lt ⇒ Lt
        | Gt ⇒ Gt
        end
    end
  end.
```

We do not show the proofs and instances generated along with these definitions. Note that we used inductive predicates to define the equality and order relations: there are other ways to generically define such relations (as a function predicate for instance) but we chose to use inductives because it makes proofs easier and shorter.[10] It is important to keep the proofs as short as possible since they can be quite large: in particular, the proof of transitivity of t_lt grows in cubic proportion to the number of constructors in t, and a call to Generate OrderedType can take several seconds on a large type.

The Generate OrderedType command will work with all (mutually) recursive inductive definitions, including uniform parameters, which makes it a very useful addendum to the library. For instance, the following commands demonstrate its use for automatically comparing strings of characters. An instance is generated for the type ascii of 8-bit characters, and then for the type string of strings, which uses ascii.

```
Generate OrderedType ascii.
Generate OrderedType string. (* string uses ascii *)
Eval vm_compute in ("long" =?= "small").
(* this computation returns Lt *)
```

5.3 Finite Sets and Maps

The ordered types we described in Section 5.2 are a type class on which it is possible to implement a few efficient structures of containers. The goal of our library is to provide such structures, and we know present and define the interface for finite sets in detail and also address finite maps.

5.3.1 Interfaces and Specifications

The class of the finite sets containing elements of an ordered type A is defined in the following way:

[10] here is one way to see why inductive predicates make proofs shorter: suppose you know t_eq x y for some x and y, inverting this hypothesis will yield the two possible cases, one for each constructor in t_eq. With a non-inductive specification, one would have to reason by analysis on x and y, which yields four cases: the two absurd cases must be eliminated manually.

```
Class FSet `{H : OrderedType A} := {
  set : Type;
  In : A → set → Prop;
  empty : set;
  mem : A → set → bool;
  add : A → set → set;
  ...
  FSet_OrderedType :>
    SpecificOrderedType set (Equal_pw set A In) _
}.
Implicit Arguments set [[H] [FSet]].
```

This class is parameterized[11] by a type `A` and an ordered type `OrderedType A`. It brings the type `set` of all finite sets of elements of type `A` as well as the various operations available on these sets. The field `In` is the membership predicate for these sets and is the only logical field in this class: all operators are consequently specified in terms of this predicate. The field `FSet_OrderedType` requires explanations: it guarantees that the type `set` is itself an ordered type, what's more an ordered type for a very specific equality; it does so by introducing a subclass `SpecificOrderedType` as described in Section 5.2.2. This equality is the pointwise extension of the membership predicate `In`, *i.e.* two sets are equal if they have the same elements, and it is defined in the following way for any container type `ctr` and element type `elt`:

```
Definition Equal_pw (ctr elt : Type)
  (In : elt → ctr → Prop) (s s' : ctr) : Prop :=
  ∀a : elt, In a s ↔ In a s'.
```

These definitions allow one to consider sets as ordered types (and in particular build sets of sets of sets of ...), for instance by writing `s === empty`. They also ensure that this equality is convertible with the pointwise equality, which is the one used in the original `FSets` library. The last line of the definition, right after the definition of the class, declares two arguments of the `set` projection as implicit. More precisely, `set` normally expects three arguments, the type of elements, an instance of `OrderedType` for this type, and an instance of `FSet`: we declare that the type of elements should be passed explicitely, but that the instances for ordered type and finite sets will be inferred automatically. The consequence of this is that the type of sets of elements of a type `A` can be denoted simply as `set A`. Given an instance of `FSet` for an ordered type `A`, we can then manipulate sets of `A` easily:

```
Definition add_all (x y z : A) (s : set A) :=
  add x (add y (add z s)).
```

[11] this design choice, far from being benign, is discussed further in Section 5.5.

In interactive proof manipulations, it is not advisable that the used instance `FSet` be fully unveiled to the user, ie. that the projections `set`, `add`, etc, can be reduced and reveal the actual implementations beneath. In particular, one's definitions and proofs should not depend on the actual set implementations but only on the interface and the provided specifications, which guarantees an encapsulation of the actual implementation of the structures, and the genericity of the code that uses the library. To that end, we make the various fields of the `FSet` class *opaque*[12]:

```
Global Opaque set In empty mem add ... .
```

The `FSet` class only contains the computational interface for the finite sets' structure and not its specification. We made this choice in order to separate operations and specifications for pragmatic reasons: definitions of functions and algorithms only need the computational interface, which remains relatively small, whereas proofs and only proofs will require the specifications. Before we define these specifications in detail, we can already define a few generic predicates and notations on finite sets, among which `Equal s t` for pointwise equality, `Subset s t` for the subset relation and `Empty s` to denote the fact that the set `s` is empty. The available notations are listed in Table 5.3.

`s [=] t`	`Equal s t`
`s [<=] t`	`Subset s t`
`v ∈ s`	`In v s`
`{}`	`empty`
`{v}`	`singleton v`
`{v ; s}`	`add v s`
`{s ~ v}`	`remove v s`
`v in s`	`mem v s`
`s ++ t`	`union s t`
`s \ t`	`diff s t`

Figure 5.3: Available notations on finite sets

All the specifications for the `FSet` class could be packaged in a single large class `FSetSpecs` parameterized by an `FSet` instance, but we instead choose to specify each operation in a separate class. For instance, the specifications for the fields `empty` and `add` are given by the following classes, and are straightforward to understand:

```
Class FSetSpecs_empty `(FSet A) := {
  empty_1 : Empty empty
```

[12] this does not prevent computations with the `compute` and `vm_compute` tactics, but only the δ-conversions, i.e. the unfolding of definitions, which are performed by some tactics.

```
}.
Class FSetSpecs_add `(FSet A) := {
  add_1 : ∀s x y, x === y → In y (add x s);
  add_2 : ∀s x y, In y s → In y (add x s);
  add_3 : ∀s x y, x =/= y → In y (add x s) → In y s
}.
```

We make this choice for two reasons. First, when writing proofs, it is very common to ask the system about all lemmas available on some identifier, say `add`, using the command **SearchAbout** `add` or one of its variants. If all the specifications – about fifty – are bundled in a single class, this command will unfortunately display this class's constructor and the elimination principle associated with it, and both are very large objects. This is rather unfortunate, and makes it almost useless in such cases. The other, more general, reason for our choice is that it makes it possible to have proofs only depend on what is really necessary. For instance, if some specific data structure implementing finite sets does not feature all the sets operations described in the interface, but one's application does not use the missing operations, one can still rely on our library and its generic interface since the missing specifications will never be required. Pushing this even further, we can imagine a development which does not involve any proof (such as a procedure used as an oracle for some larger algorithm), and would only use the computational interface `FSet`. For those systems that require the interface with full specifications, we define a superclass which embeds specifications for all operations:

```
Class FSetSpecs `(F : FSet A) := {
  FFSetSpecs_In :> FSetSpecs_In F;
  FFSetSpecs_mem :> FSetSpecs_mem F;
  FFSetSpecs_add :> FSetSpecs_add F;
  ...
}.
```

Together, this specification class and the interface class `FSet` correspond exactly to the interface `FSetInterface.S` in the existing library.

Finite maps. In this paper, we only present the interface for finite sets in order to remain concise, but the library also provides an interface for finite maps. It is adapted from the standard library's finite maps (`FSets.FMaps`) in a similar way to what we just described for finite sets. In particular, the same choices were made as far as the separation of operations and specifications, and the splitting of specifications.

5.3.2 A Library of Properties

The FSets library contains several modules of generic results and properties about finite sets: FSetFacts, FSetDecide, FSetProperties, and FSetEqProperties. The task of adapting these modules to the typeclass-based interface presented above was a first good way to check our interface and its ease of use. We adapted all the aforementioned modules without any major issue, the most delicate point certainly being FSetDecide and its tactic fsetdec initially contributed by A. Bohannon and which performs automatic reasoning on the theory of finite sets. One slight difference is that the original tactic only dealt with one single type of sets at a time, while our port of the tactic deals with all hypotheses related to sets at the same time; this can lead to minor incompatibilities. As a whole, all lemmas and properties keep the same name as in the original library, which minimizes the amount of work necessary to port one's code from the modular version to the one we present here (cf. Section 5.5.3 for more details).

We have also added some properties in order to facilitate reasoning on functions like mem, choose or min_elt, using inductive views to write their specifications. For example, choose's specification is available in the following fashion:

```
Inductive choose_spec (s : set elt) :
  option elt → Prop :=
| choose_spec_Some :
  ∀x (Hin : In x s), choose_spec (Some x)
| choose_Spec_None :
  ∀(Hempty : Empty s), choose_spec None.
Property choose_dec : ∀s, choose_spec (choose s).
```

and can be used very easily by doing case analysis on the result of choose_dec.
Higher-order iterators. Elements in a container are traditionally enumerated using step-by-step iterators in imperative languages, and higher-order iterators *à la* fold in functional languages. In FSets as well as in our library, there is one such iterator function fold ; in our interface for elements of type A, it appears as:

```
fold :  ∀ {B : Type}, (A → B → B) → set → B → B
```

where the type B is the type of what is commonly called the *accumulator*. The specification for this function is given in terms of the traditional fold_left function on lists, and the function elements returning the list of elements of a set:

```
fold_1 : ∀f s i,
  fold f s i = fold_left (fun a e ⇒ f e a) (elements s) i
```

This indirect specification is really tedious to use because in order to reason by induction on a finite set, it requires to express all the other hypotheses relative to `s` in terms of `elements s` and to proceed by induction on the list of elements. Because `fold` is used a lot when programming with finite sets, reasoning about `fold` is very frequent and the above procedure must be done repeatedly. To avoid the tedious process of using `fold_1`, we designed an induction principle for `fold`. The idea is that the induction principle lets one prove an invariant over the accumulator by proving that the invariant is true for the initial accumulator and is preserved with each iteration step.

```
fold_ind :
  ∀`{OrderedType A} (B : Type) (P : B → Type)
    (f : A → B → B) (i : B) (s : set A),
    P i →
    (∀(e : A) (a : B), In e s → P a → P (f e a)) →
    P (fold f s i).
```

The preservation is expressed by the fact that if `a` has the invariant `P` and an element `e`, belonging to the set `s`, is added to the accumulator, the resulting accumulator `f e a` still verifies `P`. This principle is still rather weak, because in general, one may need more information in order to properly express the invariant and prove its preservation. For that reason we provide the following stronger, more generic, induction principle:

```
fold_ind_gen :
  ∀`{OrderedType A} (B : Type) (P : set A → B → Type)
    (f : A → B → B) (i : B) (s : set A),
    (∀(s s' : set A) (a : B), s === s' → P s a → P s' a) →
    P i →
    (∀(e : A) (a : B) (vis : set A),
      In e s → ~In e vis → P vis a → P {e; vis} (f e a) →
    P s (fold f s i).
```

In the latter principle, the invariant takes one extra argument, the set of elements already visited by the iterator. There is one extra hypothesis to make sure that the invariant is a morphism for pointwise equality, and in the preservation step, the new element is such that it has not been visited yet. The conclusion of the principle is that the invariant is verified for the whole fold when all elements have been visited. For example, here is how to write a filtering function[13] on sets of integer using `fold` and then prove its specification using `fold_ind`:

```
Definition filter_pos (s : set nat) :=
  fold (fun e s ⇒ if e >> 0 then {e; s} else s) s {}.
Definition filter_pos_invariant (s acc : set nat) :=
  ∀e, In e acc ↔ In e s ∨ e >>> 0.
Theorem filter_pos_spec : ∀s, P s (filter_pos s).
```

[13] the generic `filter` function is actually part of the `FSets` interface, we do not use it here in order to demonstrate `fold`.

```
Proof.
  intro s; unfold filter_pos.
  apply fold_ind with (P := filter_pos_invariant).
  ...
Qed.
```

As a final note, these principles are available with our library but we first developed them for the original FSets library, and therefore they are also available for FSets starting with Coq v8.2.

5.4 Applications

5.4.1 Lists and AVL trees

The existing library FSets proposes two kind of implementations of sets and finite maps, the ones based on sorted lists, and the others on balanced binary search trees (AVL) [G. 62].

We have adapted the finite sets and maps based on sorted lists, as well as those sorted on AVL trees. Let us detail for instance the case of finite sets based on sorted lists. In practice, the implementation of sorted lists is the same in the modular version and in our version, and they differ only marginally[14]. The original development of sorted lists in the FSets library is a functor parameterized by a module of signature OrderedType, whereas the development for sorted lists in our version is parameterized by an instance of the OrderedType class. This is achieved by using Coq's sectioning mechanism and the Context command which introduces instance variables in a section:

Modular version

```
Module Make (X : OrderedType)
  <: S with Module E := X.
  Module E := X.
  Definition elt := X.t.
  ...
End Make.
```

Type class version

```
Section Make.
  Context `{OrderedType elt}.
  ...
End Make.
```

In the `{OrderedType elt} context, elt is a fresh type featuring a decidable order. The definitions in the section can then use elt as an ordered type, and they are automatically generalized at the time the section is closed.

Once the definitions of sorted lists and their various operations, as well as the adequate proofs, have been completed, we are only left with the task of declaring the instances corresponding to the classes presented earlier in Section 5.3.1. We can package all these definitions in a specific module

[14] it is thus natural to be concerned about the issue of code duplication between both versions ; we discuss this point in Section 5.5.

SetList, which doesn't have to be imported by an external user, since only the instances providing the interface are necessary. In the case of sorted lists, which provide a structure of finite sets for any ordered type, we define a generic instance of FSet parameterized by an ordered type:

```
Instance SetList_FSet `{Helt : OrderedType elt} :
  FSet := {
  set := SetList.set elt;
  In := @SetList.In elt Helt;
  empty := ...
}.
```

This definition really declares a whole family of instances, in other words it gives a way to obtain a finite set structure for any ordered type elt. Similarly, we can define a family of specifications for each of these structures indexed by an ordered type elt:

```
Instance SetList_FSetSpecs `{Helt : OrderedType elt} :
  FSetSpecs SetList_FSet := {
  FFSetSpecs_In := ...;
  FFSetSpecs_mem := ...;
  ...
}.
```

With these instances defined in a file (resp. module), it is enough to import that file (resp. module) to be able to use finite sets on any ordered type.

5.4.2 Usage

The simplicity with which our library can be used is one of its main interests. To work with finite sets, it is enough to import the module Sets which exports the following functionalities:

- the notion of ordered type, along with a library of instances and results about ordered types ;
- the generic instances for finite sets based on AVL trees, whose design is completely similar to the one based on sorted lists ;
- the interfaces, specifications, notations and basic properties relative to finite sets.

A first thing to note is that the library loads AVL trees by default instead of sorted lists. This is justified by the fact that AVL are more efficient in general and there is no penalty in terms of loading time with respect to sorted lists. This is unlike the modular version where applying the AVL functors takes much longer than applying the sorted lists' version. A user who wishes to use sorted lists instead of AVL trees can still load the adequate instances ;

she could also manually specify what instance to use if the circumstances demand it[15]. A module Maps also exists, which loads all the infrastructure required to work with finite maps ; in particular, it loads the maps based on AVL trees.

Once Sets has been imported, one can use all the generic definitions and notations on sets, with the only restriction that they be applied to ordered types. If, as is often the case, the necessary instances of OrderedType can be automatically inferred as described in Section 5.2.3, then the use of finite sets becomes totally transparent to the user, and becomes completely similar to fully polymorphic structures such as lists. The following example demonstrates the computation of a set of integers:

```
Require Import Sets.

Fixpoint fill n s :=
  match n with
  | O ⇒ s
  | S n0 ⇒ fill n0 {n0; s}
  end.
Eval vm_compute in mem 6 (fill 7 {42}).
(* this computation returns 'true' *)
```

Finite sets for different types can coexist peacefully in the same context, in the same functions ; in particular, thanks to the FSet_OrderedType field in the FSet class (cf. 5.3.1), we can manipulate sets of sets:

```
Definition map_fill (s : set nat) : set (set nat) :=
  fold (fun n S ⇒ {fill n {}; S}) s {}.
Eval vm_compute in cardinal (map_fill (fill 3 {})).
(* this computation returns 3 *)
```

Similarly easy is the use of lemmas from the library during proofs. For instance, to apply the first part of the specification of the add operation, called add_1, it is enough to apply the lemma directly and all implicit arguments are correctly inferred:

```
Goal ∀(x : option nat) s, In x {x; s}.
Proof. intro; apply add_1; reflexivity. Qed.
```

To conclude that section, here is an example involving finite maps and some of the notations associated to maps. The type of finite maps binding keys of type key to values of type elt is written Map[key, elt]. The notation s[k ← v] denotes the insertion (or the update) of a binding in the map s, [] is the empty map and s[k] is the value associated to the key k in s, if any.

[15] in such cases, the gain in verbosity compared to the modular version is reduced to zero, but this explicit instantiation can almost always be avoided.

```
Require Import Maps.

Fixpoint fill (s : Map[nat,nat]) (n : nat) :=
  match n with
  | 0 ⇒ s
  | S n0 ⇒ fill s[n0 ← S n0] n0
  end.
Eval vm_compute in (fill [] 7)[4].
(* this computation returns ’Some 5’ *)
```

The library is available for download at the following URL:
`http://www.lri.fr/~lescuyer/Containers.fr.html`.

5.5 Discussion

In this section, we take a closer look at the comparison between our library and the existing one, and discuss a couple of choices and limitations in our current implementation.

5.5.1 Performances

In order to compare the performances of our library with the module-based implementation, we added a file called `BenchMarks.v` which tests the basic functions over finite sets. The test consists in creating a set of integers from a (pseudo-)randomly generated sequence, and in making various membership tests in the resulting set. This process is repeated for sets based on type classes and sets based on modules. The result is satisfying since, when the comparison functions for the elements are the same[16], the two alternatives show the exact same performance.

To understand why the mere fact that the performances are similar is satisfying, it is important to notice that the convenient and concise formulation that comes from using type classes is actually made to the expense of the terms’ size. Indeed, although the various type classes parameters are implicit and are automatically filled in, one must not forget that these arguments are present in the proof terms, and that the corresponding instances must be passed on and reduced during the computations. For example, the simple expression `{1; {}}` (or `add 1 empty`), which denotes the singleton set containing 1, actually corresponds to the following sybilline expression:

[16] several comparison functions, for relative integers in particular, were not completely computational in the existing library and because of that, were being five times slower than the purely computational functions in our library. This does not denote any significative difference between modules and type classes, but rather underlines the importance of having an interface which encourages one to write purely computational comparison functions. We of course corrected the slow comparison functions from the existing library before running our benchmarks.

```
@add nat (@SOT_as_OT nat (@eq nat)
           (@eq_equivalence nat) nat_OrderedType)
  (@SetAVLInstance.SetAVL_FSet nat
     (@SOT_as_OT nat (@eq nat)
       (@eq_equivalence nat) nat_OrderedType))
  1
  (@empty nat
     (@SOT_as_OT nat (@eq nat)
       (@eq_equivalence nat) nat_OrderedType)
     (@SetAVLInstance.SetAVL_FSet nat
        (@SOT_as_OT nat (@eq nat)
          (@eq_equivalence nat) nat_OrderedType)))
```

whereas the corresponding expression with modules would simply be:

```
NatSet.add 1 NatSet.empty.
```

To sum this up, functor applications are replaced by applications of extra arguments in all set-related operations.

If the performances of the computations do not suffer from this hidden complexity, this is unfortunately not the case for the time spent typechecking these objects when compiling a file, or simply when manipulating them in an interactive proof. We get back to this important point *infra* in Section 5.5.4.

5.5.2 Upgrade of Existing Code

The task of updating earlier versions of our tactic to this library represented a good benchmark to judge how hard it was to adapt existing code, based on `FSets/FMaps`, to our alternative library. The code base is indeed about 30 000 lines of Coq and used various different types of finite sets, including sets of sets.

The conclusion of this experience was very positive since the modification of our existing code went on without a significant issue. As a matter of fact, because the names of operations and lemmas have been preserved between the original library and ours, the modifications one has to make to one's existing code are almost automatic:

- for all modules verifying the signature `OrderedType`, define the corresponding `OrderedType` instance[17] or use the translation functors described in the next section ;
- replace all occurences of set types like `NatSet.t` with `set nat` ;
- replace invocations of `detruct compare` in proof scripts by `destruct compare_dec` ;

[17] this is only needed if the instance cannot be inferred automatically by the system, nor generated with the `Generate OrderedType`.

- “unqualify”all references to objects belonging to modules of finite sets or properties over finite sets, for instance replace any qualified reference to `NatSet.add`, to the lemma `NatSet.add_3` or to the tactic `NatSetDec.fsetdec` by `add`, `add_3` and `fsetdec`.

These modifications can be applied seamlessly and also make one's code more concise and more readable. Therefore, they should not deter one from switching from one library to the other.

5.5.3 Code Sharing

When we presented the interfaces in Section 5.3 and the concrete implementations in Section 5.4, we emphasized how the library of generic properties and the developments of lists and AVL trees were almost exactly the same in our library and in the original one. Therefore, it is natural to wonder about how we can avoid code duplication between the two versions: for obvious reasons, it wouldn't be satisfactory if the code remained duplicated.

In order to share most of that which is duplicated over the two versions of the library, it is possible to only write the version based on type classes, and then obtain the modular version with very little boilerplate. We show this construction on the example of ordered types. Given the signature `OrderedType` and the type class `OrderedType` as in Section 5.2, we can build the following functor which translates an instance of `OrderedType` in a module of signature `OrderedType`:

```
Module Type S.
  Parameter t : Type.
  Instance Ht : OrderedType t.
End S.

Module OT_to_FOT (Import X : S) <: OrderedType.
  Definition t := t.
  Definition eq : t → t → Prop := _eq.
  Definition lt : t → t → Prop := _lt.

  Definition eq_refl : ∀(x : t), eq x x :=
    reflexivity.
  Definition eq_sym : ∀(x y : t), eq x y → eq y x :=
    symmetry.
  ...
  Definition compare : ∀x y, Compare lt eq x y.
  Proof. ... Qed.
End OT_to_FOT.
```

The signature `S` is just a way to package an ordered type with its instance in a module. The functor itself is parameterized by a module of signature

S, in other words by an ordered type, and creates a module of signature OrderedType for the type and relations passed in the parameter. The instance and the definitions for a given ordered type t can therefore be defined once and for all, and the user of the modular library can get the corresponding module via this functor. It is interesting to note that one can also build the converse translation, that is a functor parameterized by an OrderedType module which returns a module of signature S containing the corresponding instance. Of course, this translation has a lot less interest because it requires the user to manually and explicitely define each instance he needs by applying this functor, which is precisely what type classes are there to avoid. The functor OT_to_FOT, on the contrary, is not more constraining to use than the existing module-based system.

The sharing we obtain in this fashion can be generalized to other parts of the system, for instance we could define a functor returning a module of finite sets for a type A from an instance of FSet A. This way, we would only have to duplicate the interfaces of the different parts of the system, all still sharing the same concrete implementations. Our library features a module called Bridge which contains such functors, albeit only for ordered types.

5.5.4 Designing the Interface

In Section 5.3.1, we chose to parameterize the FSet type class with the (ordered) type of the elements. We could also have written the FSet class without this parameter, in the following way:

```
Class FSet := {
  set : ∀A {OrderedType A}, Type;
  In : ∀‘{OrderedType A}, A → set A → Prop;
  empty : ∀‘{OrderedType A}, set A;
  ...
}.
```

This class should be interepreted in a slightly different way from the one defined in Section 5.3.1: the class itself is not parameterized by an ordered type anymore, but each field is. Hence, an instance of this class provides implementations of finite sets for any possible ordered type and not for a single particular one. For instance, sorted lists and AVL trees, as presented in Section 5.4.1, are potential instances of this class because they can be used on any ordered type. This is in contrast to specific structures like Patricia sets [OG98] which can only be used to form sets of binary integers. The advantage of this alternative formalization is that one can use different instances in the class definition itself, for instance we could add the traditional map operation:

```
  map : ∀‘{OrderedType A, OrderedType B},
          (A → B) → set A → set B
```

whereas this would neither be possible with our parameterized interface, nor with the module system. It seems to us that this kind of benefits is less important than the ability to deal with implementations that are specific to certain element types (like integers), and therefore we decided to keep the formalization where `FSet` is parameterized.

Unfortunately, this choice is not without consequences on the size of the terms created using the library. We illustrated in Section 5.5.1 how implicit type class arguments were leading to larger terms even though they were hidden to the user. This effect gets amplified by the parameterization of the `FSet` class: indeed, all operations in `FSet` are themselves parameterized with the same arguments as the class itself, and these arguments appear twice in the proof term for each operation. For instance, suppose `F` is a generic instance of `FSet` and `nat_OT` has type `OrderedType nat`, then the expression `add 5 {}` will actually become:

```
@add F nat nat_OT 5 (@empty F nat nat_OT)
```

if the class is not parameterized (second, rejected, alternative) whereas it becomes:

```
@add nat nat_OT (F nat nat_OT) 5
  (@empty nat nat_OT (F nat nat_OT))
```

when the class is parameterized (our original, retained, alternative). The difference may seem insignificant but we have measured its effect with accuracy on a development which uses finite sets extensively, and we found out that the total size of proofs and definitions would grow by about 40%, as well as the time devoted to type-checking the source files. The increase in the size of terms and type-checking time is one of the only downsides of using type classes, and it is really unfortunate that this gets amplified by the (otherwise useful) parameterization of the `FSet` interface[18]. In practice, the time we gained in functor's instantiations still outweighed the time lost because of the size of terms.

5.5.5 Type Classes and Modules

The work presented here is not a general criticism of modules compared to type classes, let alone a criticism of the existing `FSets` library. As demonstrated in [WM06], modules and type classes are not interchangeable and each one can claim benefits over the other. In particular, modules allow a good control of the namespace, unlike type classes. Modules are also very well suited to splitting a large system in smaller parts with well defined interfaces ; functors allow one to easily replace one part of such a system by

[18] it is interesting to note that the duplicated expressions, or parameters, appear as siblings in the Coq terms and are thus typed in the same context. Therefore, some form of memoization or hash-consing in the Coq type checker would surely cancel these negative effects.

another with the same interface and this can come in very handy to test alternative algorithms or compare choices in a larger system.

However, we think that generic data structures like sets or maps are not good candidates for a modular design since it is common to need several different instances of these structures at the same time, which raises the issues mentioned in Section 5.1.2. For such cases, it seemed to us an interesting experiment to try and take advantage of the new type class system in order to provide alternative implementations of such structures.

5.6 Conclusion

We have presented a Coq library of finite sets and finite maps which reproduces much of the features of the existing `FSets/FMaps` library, but which is based on the new type class system instead of the module system. Thanks to the use of type classes, this library facilitates the use of these structures and leads to faster, more concise development of algorithms in Coq. It also avoids a couple of performance issues related to the module system. Existing implementations which rely on the standard library can be easily adapted to this version. We are convinced that such a library contributes greatly to improving Coq as a programming language since it provides easy access to standard, generic, commonly used data structures.

CHAPTER 6

A Reflexive SAT-Solver

Comment pouvez-vous identifier un doute avec certitude?
- A son ombre! L'ombre d'un doute, c'est bien connu.

Raymond Devos, *A plus d'un titre*

Contents

We start the formalization of our reflexive tactic by the propositional solver. In Chapter 1, we have emphasized how Alt-Ergo's architecture is modular and we will reproduce this modular architecture in our formalization as well. In particular, the propositional solver, as described in Chapter 2, can lead to a reflexive tactic for propositional logic, and this is what we will describe in this chapter. In Section 6.1, we describe a Coq formalization of this DPLL procedure and we prove its soundness and completeness. We then use this procedure in Section 6.2 in order to build a reflexive tactic solving propositional goals. We finish by showing how to use modularity and define a better strategy in Section 6.3.

6.1 Formalizing DPLL in Coq

In this section, we present a Coq formalization of the inference system presented in Section 2.1.3 page 25, for which we prove soundness and completeness with respect to a notion of semantics for formulae.

6.1.1 Literals

We start by defining how literals shall be represented. To do so, we will make use of Coq's module system [Cou97, Chr03]. Coq *module types* allow one to pack together types, functions and related axioms by keeping a high level of abstraction. One can then create *functors*, *i.e.* modules which are parameterized by other modules of a certain signature and which can then be *instantiated* on any modules that match the expected signature.

```
Module Type LITERAL.
  Parameter t : Set.
  (* t is an ordered type *)
  Instance t_OT : OrderedType t.
  (* Negation function and its properties *)
  Parameter mk_not : t → t.
  Axiom mk_not_invol : ∀l, mk_not (mk_not l) === l.
  Axiom mk_not_compat : ∀l l', l === l' ↔ mk_not l === mk_not l'.
  ...
  (* Sets of literals, clauses and sets of clauses *)
  Notation lset := (set t).
  Notation clause := (set t).
  Notation cset := (set clause).
End LITERAL.
```

Figure 6.1: A module type for literals

Therefore, in order to take advantage of Coq's module system, we will first define module types for literals and formulae, and we will then be able to develop our decision procedure in a way that is independent of the actual representation of the input. The signature at the base of our system is the module type `LITERAL` of literals and is presented in Figure 6.1. This module type provides a type `t` for literals, a function `mk_not` which builds the negation of a literal and some axioms about this function (like the fact that it is involutive). Literals also come with a decidable equality and a total order, which are necessary to later define finite sets of literals: this is done by requiring an instance of `OrderedType t` in the signature, as described in the previous chapter. Note that there is no way to construct literals from scratch with this signature, this is indeed not required by the DPLL procedure.

Finally, the last part of the `LITERAL` signature introduces notations for sets of literals and sets of sets of literals. We actually use two different notations, namely `lset` and `clause`, to denote finite sets of literals. Although they represent the same type, the reason we make that distinction is because our intent is that they will represent different objects and will be used in different places. Having different names ensures better maintenance and less confusion[1]. Precisely, `lset` will be used to build partial assignments, *i.e.* sets of literals that are considered to be true, whereas `clause`, as its name suggests, will be used to represent clauses, *i.e.* disjunction of literals. The last notation `cset` will be used to represent conjunctions of clauses, in other words CNF formulae. Note that in defining these notations, we used the fact that an instance for `OrderedType` for the type of literals was introduced before (in order to build the sets of literals), and also that our containers library ensures that sets of elements form an ordered type themselves (cf. Section 5.3.1), thus allowing to build sets of sets of literals.

6.1.2 Semantics and Formulae

In the previous subsection, we defined module types for literals and we now turn our attention to defining a notion of semantics, *i.e.* what it means for a formula to be "true". We cannot directly (nor do we want to) rely on the prover's notion of truth because we are dealing with abstract formulae and not native Coq propositional formulae.

Once again we use Coq's functorization system and define semantics as a functor with respect to a module `L` of type `LITERAL`. The first thing we need for semantics is a notion of *model*: in accordance with Definition 2.1.1 page 27, a model should be a function assigning a truth value to a literal. We will simply define a model as any type which *can be seeen* as a function from literals to propositional values:

```
Module Type SEM_INTERFACE (Import L : LITERAL).
  Parameter model : Type.
  Parameter model_as_fun : model → (L.t → Prop).
  Coercion model_as_fun : model ↣ Funclass.

  ...
End SEM_INTERFACE.
```

[1] In practice, we also took advantage of that distinction in order to use different finite sets implementations for `lset` and `clause`, namely AVL trees for the former and ordered lists for the latter, because they were used in a quite different manner in the algorithm: partial assignments were mainly used with membership tests, while clauses were mainly iterated upon. Therefore, the cost of keeping a balanced tree in order to obtain logarithmic lookup time was not justified for clauses. In such a case, the notations `lset` and `clause` represent different types. We simplify the presentation in this document, but note that the fact that we had made that syntactic distinction between `lset` and `clause` from the start made it much easier to use different implementations later on.

The type `model` is left abstract and can be transformed into a function from literals to propositional values using `model_as_fun`. The `Coercion` declaration ensures that we can implicitely use a model as a function over literals.

Not any function from literals to `Prop` can be considered as a model for literals, it also has to verify some properties which we require by adding some axioms to the signature:

```
Axiom morphism : ∀M l l', l === l' → (M l ↔ M l').
Axiom consistent : ∀M l, M l → ~(M (mk_not l)).
Axiom total : ∀M l, ~~(~(M l) → M (mk_not l)).
```

The first one is technical and simply expresses that a model must be a morphism for the equality on literals and is required because we did not enforce equality on literals to be Leibniz equality. The other two axioms denote the logical meaning of a model:

- `consistent` expresses that a model should not assign a true proposition to both a literal and its negation;
- `total` expresses that a model should be total, in the sense that given any literal, itself or its negation should be true in the model. It is stated with a double negation because Coq's logic is intuitionistic and we would not be able to prove this axiom without double negation for the type of models we have in mind. For instance, suppose `M l` is some propositional value `~A`, and as one can expect, `M (mk_not l)` is `A`; it is not true in general in intuitionistic logic that `~~A → A` and therefore the model would not be necessarily "total" for literal `l`. By adding the double negation, we make sure that this property is provable in intuitionistic logic.

Note that together, `total` and `consistent` are equivalent to the property `∀M l, ~~(M l ↔ ~ M (mk_not l))`, *i.e.* they express that the interpretation of the negation of a literal `l` should be the negation of the interpretation of `l`. Only the `total` part of this equivalence requires a double-negation, hence we split this property in the two axioms above.

It is now straightforward to define what it means for a model to satisfy a clause or a set of clauses, and when a formula in CNF is unsatisfiable:

```
Definition sat_clause (M : model) (C : clause) :=
  ∃l ∈ C, M l.
Definition sat_goal (M : model) (D : cset) :=
  ∀C ∈ D, sat_clause M C.
Definition unsatisfiable (D : cset) :=
  ∀(M : model), ~sat_goal M D.
```

This gives us a notion of satisfiability for clauses and formulae, but we also need to take the context of a sequent into account. As we did in the proofs of Chapter 2, we need a notion of how a set of literals can be a *submodel* of some model:

```
  Definition submodel (G : lset) (M : model) := ∀l ∈ G, M l.
```

Note that this definition of a submodel implies that G is a well-formed partial assignment, in the sense that it does not contain both a literal and its negation. From this notion of submodel naturally follows the definition of incompatibility between a partial assignment and a set of clauses:

```
  Definition incompatible (G : lset) (D : cset) :=
    ∀(M : model), submodel G M → ~sat_goal M D.
```

We can now define a module type CNF for formulae, as shown in Figure 6.2. This signature provides a type formula for the concrete representation of formulae. Because the type of formulae will depend on some notion of literals, the signature CNF also embeds a module L of signature LITERAL through the Declare Module vernacular. Another module is required in the interface, with the signature SEM_INTERFACE L, which brings a notion of model and semantics for the module of literals L. Finally, an instance of CNF instance shall provide a "CNF conversion" function called make that takes a formula and returns a sets of clauses (as defined in the module of literals). Such a formalization (having the module bringing its own abstract type of formulae and conversion function) allows instances that only accept formulae that are already in CNF, and where make is just the identity function for instance.

```
Module Type CNF.
  Parameter formula : Set.

  Declare Module L : LITERAL.
  Declare Module Sem : SEM_INTERFACE L.

  Parameter make : formula → L.cset.
End CNF.
```

Figure 6.2: A module type for formulae

6.1.3 Sequents and Derivations

We can now start the definition of a functor SAT parameterized by a module F of type CNF and which will implement our SAT solving algorithm without any knowledge about the actual representation of formulae or literals. The development can only use elements that are defined in F's signature and this ensures modularity as well as reusability. The functor starts with the definition of sequents: a sequent, noted G ⊢ D, is simply a record with a partial assignment G and a set of clauses D, as discussed in Section 2.1.3. For conveniency, we “redefine” incompatibility for sequents using incompatibility from the semantics module Sem.

```
Module SAT (Import F : CNF).
  Import L.
  Record sequent : Type := {G : lset; D : cset}.

  Definition incompatible (S : sequent) :=
    Sem.incompatible (G S) (D S).
  ...
End SAT.
```

The next step is the definition of the rules system presented in Fig. 2.1. We use an inductive definition shown[2] in Fig. 6.3 by enumerating all possible ways a derivation can be built from a given sequent. We call this inductive derivable and an object of type derivable (G ⊢ D) represents a proof derivation of sequent G ⊢ D. Note that each constructor faithfully follows from a rule of the original system. For instance, Assume describes unit propagation, and Elim and Red together describe the two rules for boolean constraint propagation.

```
Inductive derivable : sequent → Prop :=
| Conflict : ∀G D, ∅ ∈ D → derivable (G ⊢ D)
| Assume : ∀G D l, {l} ∈ D → derivable (G, l ⊢ D \ {l}) →
    derivable (G ⊢ D)
| Elim : ∀G D l C, l ∈ G → l ∈ C → C ∈ D →
    derivable (G ⊢ D \ {C}) → derivable (G ⊢ D)
| Red : ∀G D l C, l ∈ G → l̄ ∈ C → C ∈ D →
    derivable (G ⊢ D \ C, C \ {l̄}) → derivable (G ⊢ D)
| Split : ∀G D l, derivable (G, l ⊢ D) → derivable (G, l̄ ⊢ D) →
    derivable (G ⊢ D).
```

Figure 6.3: The inductive definition of the proof system

6.1.4 The Decision Procedure

Using the semantics we defined earlier, we can now proceed to prove the fundamental theorems about our derivation system. First in line is the soundness of the proof system:

[2] In this figure and in the following, we use mathematical notations for set-related operations, rather than Coq’s concrete syntax, for the sake of readability.

if there exists a derivation of the sequent $\emptyset \vdash$ `D`*,* `D` *is unsatisfiable*

and as in the proofs in Chapter 2 we prove something more general than this statement, using the notion of incompatibility that we just described:

```
Theorem soundness : ∀S, derivable S → incompatible S.
```

The special case where the context of sequent `S` is empty yields exactly the above statement. This theorem can be proved by a structural induction on the derivation of `S` following the arguments from Theorem 2.1.5, and the Coq proof is not difficult (about 50 lines of tactics).

Conversely, the completeness of the algorithm could be expressed by the following statement:

```
Theorem completeness :
  ∀S, wf_context (G S) → incompatible S → derivable S.
```

which corresponds to Theorem 2.1.9. There are at least two reasons why we do not prove completeness in this particular form:

- We do not only want full equivalence between the notions of derivability and incompatibility, but we also want a decision procedure, *i.e.* a function capable of telling if a given formula is unsatisfiable or not. Proving such a theorem of completeness would certainly give us an equivalence between the derivability of a sequent and its incompatibility, thus bringing the problem of deciding satisfiability down to the one of deciding derivability. However, deciding derivability amounts to try and build a derivation for a given sequent if possible, and it is a proof that actually encompasses the completeness theorem presented above. Thus, we want to avoid doing the same job twice.

- We want to be able to use that procedure in Coq through the mechanism of reflection, *i.e.* by actually computing the proof search in the system. Of course, an intuitionistic completeness proof is constructive and therefore can give a derivation, as an algorithm, but it is well known that procedures with propositional contents cannot be executed as efficiently as purely computational functions, because in the first case, proofs need to be replayed along with computations. Thus, we do not want to encode the decision procedure as part of a general completeness theorem.

For these reasons, we will build the decision procedure in two steps: first we will program a function without propositional content to implement the actual decision procedure, and then we will show that its results are correct. This function will not return any “complex” information, but only `Sat G` if it has found a partial model `G`, and `Unsat` otherwise:

```
Inductive Res : Type :=
| Sat : lset → Res
| Unsat : Res.
```

The decision procedure *per se* can now be implemented as a recursive function returning such a result:

```
Fixpoint proof_search (G ⊢ D : sequent) n {struct n} : Res :=
  match n with
    | O ⇒ Sat ∅ (* Absurd case *)
    | S n₀ ⇒
      if is_empty D then Sat G (* Model found! *)
      else
        if ∅ ∈ D then Unsat (* Rule Conflict *)
        else ...
        ...
          let l := pick D in (* Rule Split *)
          match proof_search (G, l ⊢ D) n₀ with
            | Sat M ⇒ Sat M
            | Unsat ⇒ proof_search (G, l̄ ⊢ D) n₀
          end
        ...
  end.
```

Because the recursion is not structural, we use an extra integer argument n, and we will later make sure that we call the function with an integer large enough so that n never reaches 0 before the proof search is completed. This short excerpt of the function `proof_search` shows that it proceeds by trying to apply some rules one after another, one rule at a time, with a given *strategy*. Here, the function first checks if the problem is empty, in which case it returns the current context as a model; otherwise, it checks if the empty clause is in the formula, in which case it returns `Unsat`. We then skip some parts of the function, where we try to apply the rules for elimination, reduction or unit propagation. The last part corresponds to the `Split` constructor: some literal `l` is picked in the problem using the `pick` function and the proof search is called recursively with the literal added to the partial assignment, which corresponds to the left branch of the `Split` rule. If this branch is satisfiable, the whole formula is satisfiable in the same model. If it is unsatisfiable, we call the proof search again for the right branch and return the result.

The first theorem about `proof_search` states that when it returns `Unsat`, it indeed constructed a derivation on the way:

```
Theorem proof_unsat :
  ∀n S, proof_search S n = Unsat → derivable S.
```

The proof follows the flow of the function and shows that each recursive call that was made corresponds to a correct application of the derivation rules. One may wonder why we didn't construct this derivation in `proof_search`, so as to return it with `Unsat`: the reason is that a derivation contains proofs (in side conditions) and had we done so, our function would not have been 100% computational anymore.

The second theorem about `proof_search` is the one that encompasses completeness: it states that if `Sat M` has been returned, it is indeed a model of the formula and of the context[3].

```
Theorem proof_sat :
 ∀n S M, μ(S) < n → wf_context (G S) →
   proof_search S n = Sat M →
   (G S) ⊆ M ∧ sat_goal M (D S).
```

A couple of remarks about this theorem are necessary:

- μ is a *measure* of a sequent that we have defined in Coq, and for which we proved that it decreases for every recursive call in the algorithm. We could have defined the function by a well-founded induction on this measure, but it is computationally slightly more efficient to use the extra integer. This is a well-known technique to transform non-structural inductions in structural inductions [BC04]. A suitable measure of a sequent `G ⊢ D` here is the size of `D` plus the number of literals which appear in `D` and are unbound (positively or negatively) in `G`. When calling `proof_search` on a sequent `S`, a suitable integer is $\mu(\mathtt{S}) + 1$: it is large enough for `proof_sat` to be applicable, in other words for the procedure to be complete;

- we need an extra hypothesis that the context remains well-formed (`wf_context (G S)`), which means that it doesn't contain a literal and its negation. This is not guaranteed by the derivation rules because the side conditions were purposely very loose in order to allow any kind of strategy. Here, it is our strategy that guarantees this invariant is never broken, and this is part of the completeness proof.

Together with the soundness theorem, this shows that `proof_search` is a decision procedure for unsatisfiability and we can now define this "top-level" `dpll` function and prove the corresponding soundness theorem:

```
Definition dpll (f : formula) : Res :=
  let S := ∅ ⊢ (make f) in
  proof_search S (μ(S)+1).
```

[3] Technically, the set returned is not a model because it is only partial; it can be completed into a model though, as long as it is a valid partial assignment, and we simplified the actual details here since they seem cumbersome.

```
Theorem dpll_correct :
  ∀f, dpll f = Unsat → Sem.incompatible ∅ (make f).
```

The definition of `proof_search` and the proofs of its properties require 700 lines of code.

6.2 Deriving a Reflexive Tactic

We now show how to use the procedure we have developed so far as a tactic to solve goals in our proof assistant.

6.2.1 Reification

In order to use our SAT solver on Coq propositional formulae, we need to instantiate the `SAT` functor. This raises the question of the actual representation of formulae and literals: we need to build modules of types `LITERAL` and `CNF` that will represent Coq formulae.

A natural choice for the type of literals would be to directly use the type `Prop` of propositions, but this is impossible because the type of literals must be an `OrderedType`, and in particular we need to be able to decide if two given propositions are equal or not. Indeed, consider the formula `A ∧ ~A`: we need to know that the propositional variable `A` is the same on both sides to conclude that this formula is unsatisfiable. Since the only decidable equality on sort `Prop` is the one that is always true, we cannot use `Prop` as the type of literals.

Instead, we resort to Coq's metalanguage $\mathcal{L}$tac, which we introduced in Section 4.2.1. This language provides pattern-matching facility on Coq terms, and thereby allows us to check the syntactic equality of propositional terms at a metalevel. We will use this language to build, for a given propositional formula `F`, an *abstract representation* of `F` on which we will be able to apply the algorithm. This process, called *reification* or sometimes *metaification*, has been introduced earlier in Section 4.2.4.

Using $\mathcal{L}$tac, we first build a function `get_vars` which traverses a formula `F` and retrieves a list of all the propositional variables of `F`. We define another function `list_to_map` that turns such a list into a balanced map. This map now contains all the propositional variables of `F` and provides an efficient way to search for a particular variable into a map. The type of these maps is called `varmap` and is defined as a parameterized binary tree:

```
Inductive varmap (A : Type) :=
| Empty_vm : A → varmap A
| Node_vm : A → varmap A → varmap A → varmap A.
```

For instance, if `F` is the following formula:

```
  F:  A ∧ (~B ∨ (p A C)) ∧ (∀D, (p D D))
```

the result of `list_to_map (get_vars F)` will be a map containing the variables `A`, `B`, `(p A C)` and `∀D, (p D D)`. In particular, the last variable is abstracted because our propositional language does not include quantifiers. Given this map, we are able to represent variables by their path in the map: the type of paths is `index` and is defined as

```
Inductive index : Set :=
| Left_idx : index → index
| Right_idx : index → index
| End_idx : index.
```

As long as the varmap is built in a balanced way, the representation of literals through indices is logarithmic in the total number of variables in the formula. It is now straightforward to create the module `LPROP` of literals, where a literal is just an `index` in the map and a boolean saying if it is negated or not, and the `mk_not` function a simple inversion of this boolean:

```
Module LPROP <: LITERAL.
  Definition t := index × bool.
  Definition mk_not (p,b) : t := (p, negb b).
  ...
End LPROP.
```

We can move on to defining the corresponding types for formulae. We will for now assume that we only deal with formulae in conjunctive normal form, and we address the problem of conversion to CNF later in Chapter 7. In Fig. 6.4, we show an excerpt of the module `CNFPROP` of type `CNF`, which implements our type of formulae. Its literals are, of course, the literals of the module `LPROP` we just defined. Formulae and clauses are defined in a very natural way by two inductives: a formula is either a clause or a conjunction of formulae; a clause is a literal or a disjunction of clauses. This representation makes the function `make` converting a formula to a set of sets of literals (not shown here) really straightforward.

The `CNFPROP` module is not finished yet since we also need to provide a module of interface `SEM_INTERFACE LPROP`, *i.e.* semantics for the propositional literals. A natural model for literals is a map of type `varmap Prop`, since it binds literals to their propositional value:

```
Module SEMPROP <: SEM_INTERFACE LPROP.
  Definition model := varmap Prop.
  Definition model_as_fun (v : model) (l : L.t) : Prop :=
    match l with | (id, true)  ⇒   lookup id v
                 | (id, false) ⇒ ~(lookup id v) end.
  ...
End SEMPROP.
```

where `lookup id v` returns the proposition bound to `id` in the map `v`, and the default proposition `True` if `id` is not bound in the map. The coercion

```
Module CNFPROP <: CNF.
  Module L := LPROP.

  Inductive clause : Set :=
  | COr : clause → clause → clause
  | CLit : L.t → clause.
  Inductive formula : Set :=
  | FAnd : formula → formula → formula
  | FClause : clause → formula.
  ...
End CNFPROP.
```

Figure 6.4: A module for propositional formulae

`model_as_fun` can be seen as a way to *interpret* literals in a varmap, and we can interpret clauses and formulae using this interpretation of literals:

```
  Fixpoint cinterp (v : model) (c : clause) : Prop :=
    match c with
    | CLit l → v l
    | COr c1 c2 → cinterp v c1 ∨ cinterp v c2
    end.
  Fixpoint interp (v : model) (f : formula) : Prop :=
    match f with
    | FClause c → cinterp v c
    | FAnd f1 f2 → interp v f1 ∧ interp v f2
    end.
```

This interpretation function `interp` is such that `interp v f` interprets an object `f` of type `formula` to its propositional counterpart in Coq, and is the reverse operation of reification.

The last step of the reification process is to build a tactic in $\mathcal{L}$tac, that, for a given formula `F` in Coq's propositional language, builds an abstract formula `f` of type `formula` and a map `v` such that `interp v f = F`. We have already covered the construction of the map `v`. The construction of the formula `f` is realized by a couple of recursive $\mathcal{L}$tac tactics which analyze the head symbol of the current formula to construct the corresponding abstract version. For instance, the top-level function matches conjuncts and goes like this:

```
  Ltac reify_formula F v :=
    match constr:F with
      | and ?F1 ?F2 ⇒
        let f1 := reify_formula F1 v
      with f2 := reify_formula F2 v in
        constr:(FAnd f1 f2)
    | ?F ⇒
      let c := reify_clause F v in constr:(FClause c)
  end.
```

Now, if we go back to our previous example, and if we take this formula as our current goal, we can use the tactics we just described to build a suitable map, reify the goal in an abstract formula `f`, and replace the current goal by the interpretation of `f`.

```
1 subgoal

============================
A ∧ (~ B ∨ (p A C)) ∧ (∀D : Prop, (p D D))
```

```
match goal with | ⊢ ?F ⇒
  let varmap := list_to_map (get_vars F) in
  let reif := reify_formula F varmap in
  set (v := varmap); set (f := reif);
  change (interp v f)
end.
```

```
1 subgoal

v := Node_vm Prop (...) (...) : varmap Prop
f := FAnd (FClause ...) (FAnd ... ...) : CNFPROP.formula
============================
interp v f
```

In particular, the `set` tactics introduce the varmap and the reified formula in the context, and the tactic `change` asks Coq to change the goal using conversion: it computes `interp v f` and checks that it is indeed equal to the original goal.

6.2.2 The Generic Tactic

At this point, in order to turn our development into a user-friendly tactic, we still need to address a couple of issues.

Conversion to normal form. Before running the actual proof search, a formula should be put in CNF. If it is not in CNF, then some subformulae will be abstracted (like the quantified part in our example above). We address the issue of conversion to normal form in detail in Chapter 7, where we propose an original way of performing this conversion in a lazy, on-the-fly, fashion. For now, let us suppose that we use tacticals to convert formulae in the context, prior to applying the tactic. Coq provides a tactic named `autorewrite` which performs automatic rewriting of expressions. When fed with a set of (oriented) equalities describing a normalizing system, `autorewrite` will transform an expression into its normal form with respect to this system. Thus, we encode the conversion into CNF as a set of rewriting rules[4]: linearizing implications, pushing negations to the atomic variables, distributing disjunction over conjunction, etc.

Lifting the Semantics. We now have a reification mechanism which allows us to transform propositional formulae in our Coq context into objects of the form `interp v f` for some map `v` and concrete object `f`. In order to obtain reflexive proofs, we still need to lift our notion of semantics on propositional literals `LPROP.t` and formulae `CNFPROP.formula` to Coq's notion of truth. Recall that we have defined models as varmaps containing propositional values. For each formula reified in a varmap `v`, there is a "canonical" model, which is `v` itself. Indeed, if `l` is a literal representing a variable `A` of type `Prop` in the map, this canonical model satisfies `l` if and only if there is a proof of `A`. This result lifts to clauses and formulae, and we can prove this `adequation lemma`:

Theorem `adequation :`
  $\forall v$ `(`f `: formula), interp` v f $\to$ `sat_goal` v `(make` f`).`

This theorem can be read as : "if there is a proof of a formula `F`, then its reified counterpart `f` is satisfiable", and a satisfying model is the varmap in which `F` was reified. Together with the soundness of the decision procedure, this gives us the following fact:

Corollary `validity :` $\forall v$ `(`f `: formula),`
  `dpll` f `= Unsat` $\to$ `~(interp` v f`).`

Note how similar that theorem is to the `check_correct` theorem that we introduced in Section 4.2.4, page 102. It is the reflection theorem for our `dpll` procedure as it reflects the computational result of `dpll` (or equivalently `proof_search`) to a propositional proof `~(interp v f)`. In particular, the conclusion of this reflection theorem is a negation, which shows that our procedure can only proceed by refutation (since it checks that a formula is unsatisfiable) of the context.

Wrapping up. We can now wrap everything up in a high-level tactic `unsat` which performs the following steps:

1. introducing as many hypotheses from the goal to the context as possible, and building the conjunction `F` of all the hypotheses in the context, changing the goal to `False`;

[4] In practice, we use several complementary rewriting systems, because for efficiency reasons, some transformations must be done before others, e.g. rewriting of implications.

2. converting `F` to CNF using rewriting as described above;

3. reifying `F` into a concrete formula `f` and a map `v` for interpreting variables;

4. changing `F` to `interp v f` using the conversion rule;

5. applying the `validity` theorem to `v` and `f` in order to bring the current goal down to a proof of `dpll f = Unsat`;

6. asking Coq to compute this equality, thus triggering the actual proof search;

7. if the procedure returns `Unsat`, the goal becomes `Unsat = Unsat` and is thus trivially proved; if however the goal is `Sat M = Unsat` for some `M`, then the context is not satisfiable, the tactic fails and prints out the countermodel `M`, since it can be very useful to the user in order to understand why the tactic did not succeed.

Users of classical logic assume the excluded-middle in their developments, and therefore they can use the same mechanism to prove the *validity* of a current goal `F`, by first applying double negation, introducing `~F` and trying the `unsat` tactic on `~F`. We provide a tactic called `valid` that performs these operations. The definitions and proofs for `unsat` and `valid` represent about 500 lines.

Examples. We finish this section by giving a small example of how the tactic `unsat` can be used in practice. Suppose our goal is the following propositional formula where variables `A` to `D` have type `Prop`:

```
1 subgoal

A : Prop
B : Prop
C : Prop
D : Prop
============================
A ∧ (C ∨ ~B ∧ (~D → ~A)) → D ∧ D ∧ ~A
```

If we try to apply `unsat` to this goal, the tactic will try to show that the left-hand side of the implication is unsatisfiable. Since it is not, the tactic fails and prints out the countermodel shown below: indeed, one can easily verify that this valuation makes the goal false.

```
unsat.
```

```
The formula in not valid.
The following countermodel has been found :
D  :  true
B  :  false
A  :  true
```

We can use this countermodel to add complementary hypotheses to our formula, for instance that `B` is true and `A` is false. By doing so, we see that the `unsat` tactic now succeeds in about one tenth of a second:

```
1 subgoal

A : Prop
B : Prop
C : Prop
D : Prop
============================
A ∧ (C ∨ ~B ∧ (~D → ~A)) → B ∧ ~A → D ∧ D ∧ ~A
```

```
Time unsat.
```

```
  Proof completed.
  Finished transaction in 0. secs (0.108007u,0.s)
```

6.2.3 About Completeness

We have seen that, so far, only the soundness of our decision procedure was useful in developing the reflexive tactic: it allowed us to establish the reflection theorem `validity`. The soundness of the procedure formally guarantees that when our tactic succeeds, the goal was indeed valid. However, our decision procedure was not only sound, but also complete, and we made no use of the completeness theorem yet.

First of all, it is technically possible to use the completeness theorem in a similar way to how we used the soundness theorem. We have seen how a result of `Unsat` for the proof search reflects to a proof of `(interp v f)`; we could similarly reflect a result of `Sat M` to a proof that the conjunction of all literals in `M` implies `interp v f`. In practice, if `M` contains literals which interpret to propositions $\mathtt{A}_1$, $\mathtt{A}_2$, …, $\mathtt{A}_n$, this would amount to adding a new hypothesis of type:

```
compl : A1 → A2 → ... An → F
```

to the context, where `F` is the formula which reifies to `f`. In particular, an hypothesis of type `F` is already in the context and therefore this new hypothesis would be of no use. This is why we just output the counter model to the user.

Even without explicitely using the countermodel when the formula is satisfiable, a legitimate concern is to know whether the tactic is "complete" or not. Although the procedure is complete in the sense of the propositional semantics defined in the `SEM_INTERFACE` procedure, this property does not lift to Coq's notion of truth; in other words, the formula `F` is not necessarily satisfiable because the $\mathtt{A}_i$ do not necessary form a consistent conjunction:

- Coq's logic is much richer than propositional logic and as one can expect, the procedure can find a counter model with literals which are inconsistent in general. For instance, it could add the literal `~(0 = 0)`, or the two mutually exclusive literals $\forall x$`, p` x and `~p` t.
- More annoyingly, the procedure can fail because the reification introduces an abstraction layer which cannot be formally proved. For instance, if the reification of `A` $\wedge$ `~A` is not performed adequately and maps `A` to some variable l, and `~A` to some other variable l', instead of the negation of l, the procedure will determine that the formula can be satisfied with $\{l; l'\}$.

Knowing when a procedure is complete can help understand the results of a tactic; in particular, any unexpected failure shall be a consequence of an unexpected behaviour of the reification process. Displaying the counter model when the tactic fails is one way to let the user check if the formula is indeed satisfiable, or if there is anything wrong in the model displayed. Nevertheless, the reflexive tactic only formally relies on the soundness property, and the reflexive approach can be used with semidecidable or undecidable properties, as long as the procedure is sound. In the remaining of this document, when presenting evolutions of this first reflexive tactic, we will only address the issue of soundness.

6.3 A Better Strategy

The decision procedure `proof_search` presented in Section 6.1.4 is rather coarse and applies the possible rules in turn, one after another. It is one of the most basic possible strategy to build a derivation and we now implement a much better strategy, which we use in practice. Once we have formalized the derivation system and proved its soundness, we are indeed free to implement any strategy and derive a reflexive tactic just as we did in the last section. The module system can help us do that in a modular manner.

We define a module type `DPLL`, parameterized by a module of signature `CNF`, which describes the interface that a procedure shall verify in order to be usable in the reflexive tactic:

```
Module Type DPLL(Import F : CNF).
  Inductive Res : Type :=
  | Sat : L.lset → Res
  | Unsat : Res.

  Parameter dpll : formula → Res.
  Axiom dpll_correct :
    ∀f, dpll f = Unsat → Sem.incompatible ∅ (make f).
End DPLL.
```

The signature requires a `Res` datatype similar to the one we have seen above, and a function `dpll` taking a `formula` and returning a `Res`, along with a proof that it is correct. This function is the real proof search, and it is straightforward to check that our functor `SAT F` presented in Section 6.1 has signature `DPLL F`. The whole development of the reflexive tactic can then be implemented as a functor parameterized by such a module; it is not parameterized by a `CNF` module though, since much of the development (the reification, the tactic and the reflection theorem) depends on the particular representation of literals and formulae. For instance, the development of the tactic presented above for propositional literals is wrapped in the following functor:

```
Module LoadTactic (Import D : DPLL CNFPROP).
  ...
  Ltac unsat := ...
End LoadTactic.
```

This makes it easy to define several different strategies, generate a tactic for each one and compare the tactics obtained for each of these strategies.

We implemented various strategies with their soundness proofs, but we now quickly present our fastest strategy. Incidentally, this strategy is exactly the same as the one used in Alt-Ergo. It is based on the following observation: although the derivation and sequents are expressed in terms of sets of literals, and sets of clauses, it is not mandatory that the procedure uses these data structures, as long as it is possible to relate what the procedure does to sequents and derivations. During the proof search, the partial assignments are used exclusively for adding elements and membership tests, therefore an efficient structure of finite sets (like AVL) seems adequate. On the other hand, an efficient strategy for propagating boolean constraints on the sets of clauses is to iterate on every clause, and every literal in every clause, trying to eliminate and reduce as many clauses as possible. To perform such a task, keeping clauses as AVLs or ordered lists is not required, and basic lists can prove much more efficient. Therefore, in this strategy, the partial assignment will have type `lset` and the set of clauses will have type `list (list L.t)`. Lists of literals and lists of lists of literals can be converted back to `clause` and `cset` using the adequate functions:

```
Fixpoint l2s (l : list L.t) : clause :=
  match l with | nil ⇒ ∅ | a::q ⇒ {a; l2s q} end.
Fixpoint ll2s (l : list (list L.t)) : cset :=
  match l with | nil ⇒ ∅ | a::q ⇒ {l2s a; ll2s q} end.
```

and the main recursive function in the strategy has the following type, and its correctness lemma is expressed using `ll2s`:

```
Fixpoint proof_search (G : lset) (D : list (list L.t))
  {struct n} (n : nat) : Res := ...
Theorem proof_search_unsat :
  ∀n G D, proof_search G D n = Unsat → derivable (G ⊢ ll2s D).
```

The strategy uses two auxiliary functions, `reduce` and `bcp`. Function `reduce` is used to reduce a clause with respect to a given partial assignment as much as possible:

```
Inductive redRes : Type :=
| redSome : list L.t → bool → redRes
| redNone : redRes.
Fixpoint reduce (C : list L.t) : redRes :=
  match C with
  | nil ⇒ redSome nil false
  | l::C' ⇒
    if l ∈ G then redNone
    else
      match reduce C' with
      | redNone ⇒ redNone
      | redSome Cred b ⇒
        if mk_not l ∈ G then redSome Cred true
        else redSome (l::Cred) b
      end
  end.
```

If it finds a true literal in the clause, it returns `redNone` denoting that the clause can be eliminated from the problem. Otherwise, it returns the reduced clause, with an extra boolean which is true iff the clause has changed. For instance, here is one of the properties of `reduce`, namely its soundness when it returns a clause:

```
Corollary reduce_correct : ∀C Cred bred,
  reduce C = redSome Cred bred →
  derivable (G ⊢ {l2s Cred; D}) →
  derivable (G ⊢ {l2s C; D}).
```

Note how this statement can be read as an advanced inference rule, the fact that we can prove it means that this rule is derivable from the basic set of rules. The `bcp` function does the boolean constraint propagation on the clauses of a problem. It proceeds with respect to a partial assignment by reducing all clauses (using `reduce`), assuming literals in unitary clauses along the way.

```
Inductive bcpRes : Type :=
| bcpSome : lset → list (list L.t) → bool → bcpRes
| bcpNone : bcpRes.
Fixpoint bcp (G : lset) (D : list (list L.t)) : bcpRes :=
  match D with
  | nil ⇒ bcpSome G nil false (* no clauses *)
  | C::D' ⇒
    match reduce G C with
    | redNone ⇒  (* elim C *)
      match bcp G D' with
      | bcpNone ⇒ bcpNone
      | bcpSome G' D' _ ⇒ bcpSome G' D' true
      end
    | redSome nil bred ⇒ bcpNone (* conflict *)
    | redSome (l::nil) _ ⇒ (* unit *)
      match bcp (add l G) with
      | bcpNone ⇒ bcpNone
      | bcpSome G' D' _ ⇒ bcpSome G' D' true
      end
    | redSome Cred bred ⇒ (* reduce C *)
      match bcp G D' with
      | bcpNone ⇒ bcpNone
      | bcpSome G' D' b ⇒
        bcpSome G' (Cred::D') (bred || b)
      end
    end
  end.
```

It returns `bcpNone` if one of the clauses reduced to the empty clause along the way, and `bcpSome G' D' b` otherwise, where `G'` is the extended partial assignment, `D'` the simplified set of clauses and `b` a boolean true if and only if there was any progress. For instances, here are some of the properties of `bcp` which prove its soundness, and can be seen as derived inference rules:

```
Theorem bcp_correct : ∀D G Gext Dred b,
  bcp G D = bcpSome Gext Dred b →
  derivable (Gext ⊢ ll2s Dred) →
  derivable (G ⊢ ll2s D).
Theorem bcp_unsat : ∀D G,
  bcp G D = bcpNone → derivable (G ⊢ ll2s D).
```

Finally, the toplevel function `proof_search` just applies `bcp` repeatedly until it returns `bcpNone` (in which case the problem is unsatisfiable) or until it does not progress any more, in which case it splits on a literal and searches recursively in the left branch, and then in the right branch if no model was found.

```
Fixpoint proof_search (G : lset) (D : list (list L.t))
  (n : nat) {struct n} : Res :=
  match n with
    | O ⇒ Sat empty (* assert false *)
    | S n0 ⇒
      match bcp G D with
        | bcpNone ⇒ Unsat (* conflict *)
        | bcpSome newG newD b ⇒
          match newD with
            | nil ⇒ Sat newG (* empty *)
            | _ ⇒
              if b then (* progress *)
                proof_search newG newD n0
              else (* G = newG, D = newD *)
                let l := pick D in
                match proof_search {l; G} D n0 with
                | Sat M ⇒ Sat M
                | Unsat ⇒ proof_search {l̄; G} D n0
                end
            end
        end
    end.
```

With the various properties of `bcp`, we can establish the correctness of this procedure and give it the expected signature `DPLL F`:

```
Definition dpll (f : formula) :=
  let D0 := make f in
  let L0 := List.map elements (elements D0) in
    proof_search ∅ L0 (μ(D0)+1).
Theorem dpll_correct :
  ∀f, dpll f = Unsat → Sem.incompatible ∅ (make f).
```

6.4 Conclusion

We have presented a formalization of a propositional solver and its use as a reflexive decision procedure for propositional logic. We have shown how using the module system can be beneficial, just as in a usual programming language. First, we were able to develop a procedure independent of the actual representation of formulae, and we could use it to decide the satisfiability of boolean logic without much pain, by defining the suitable `CNFBOOL` of

`CNF`. We will use many more representations of literals in the next chapters. Also, we can factorize the development of reification and of the top-level tactic in a functor parameterized by the underlying procedure. This allows us to easily develop different strategies and derive reflexive tactics for these strategies.

The strategy that we presented in Section 6.3 is not the only possible, nor the fastest possible of course. We have actually tried a fairly good number of different strategies, but this one is particularly interesting for two reasons: first it is precisely the strategy used by the Alt-Ergo theorem prover, and therefore it was worth investigating its correctness; second, this strategy can be adapted easily to the modifications which we will apply to the general design of our propositional solver in the next chapters (adding a lazy CNF conversion, and then generalized environments instead of partial assignments), which was not the case of all the strategies we tried.

Of course, another way of improving the procedure is to use a more refined inference system, such as the ones with backjumping or conflict-driven clause learning presented in Chapter 2. We have formalized these systems and their proofs in Coq, in the similar fashion to what we did in this chapter, but we do not present them in this document. Our main reason is that, even if they allow more efficient SAT solving tactics, we will not use these optimizations in the more general setting of SMT solving which we will describe in the following chapters, and we do not think describing these systems here has much interest. The formalization and the proofs simply follow the description in Section 2.2. For reference, we give in Appendix B benchmarks comparing reflexive propositional procedures obtained with the basic and optimized derivation systems and for various strategies.

CHAPTER 7

Dealing with CNF Conversion

> Que la paresse soit un des péchés capitaux nous fait douter des six autres.
>
> Robert Sabatier

Contents

In the previous chapter, we have designed a tactic based on a SAT solver which can be used to decide the validity of propositional formulae in Conjunctive Normal Form (CNF). In order for our tactic to be able to deal with the full propositional fragment of Coq's logic and be useful in practice, we must perform a conversion into CNF before applying the procedure. This conversion step can be critical for the efficiency of the whole system since it can transform a rather easy problem into one that is much too hard for our decision procedure. In the previous chapter, we relied on a simple rewriting of Coq formulae prior to the reification process, but this is not a satisfactory solution. A much better solution, which is used in Alt-Ergo as well as in other SMT solvers, is to rely on a lazy conversion mechanism such as Simplify's [DNS05]. Because this mechanism must be tightly coupled to the decision procedure, this requires adapting the DPLL rules. It also rules out the use of an external tool and takes advantage of our approach of proof by full reflection.

In this chapter, we show how to adapt our fully certified standard DPLL procedure in order to take a lazy conversion scheme into account. In Section 7.1, we start by some preliminary considerations about CNF conversion techniques. We describe our abstraction of the lazy CNF conversion method in Section 7.2 as well as the necessary modifications to the DPLL procedure. Section 7.3 then presents how the lazy CNF conversion can be efficiently implemented in Coq. Finally, we compare our tactic with other methods in Section 7.4 and argue about its advantages and how they could be useful in other settings.

7.1 The CNF Conversion Issue

In order for a reflexive tactic based on a SAT solver to deal with the full propositional fragment of Coq's logic, it needs to be able to take any arbitrary formula in input and convert it into CNF, which is the only class of formulae that the DPLL procedure can handle. Looking at Fig. 4.2 page 103 once again, which shows an overview of our reflexive tactic, there are two possibilities as to where this CNF conversion can occur: on the Coq side or on the abstract side, *i.e.* before or after the formula is reified into an abstract Coq object.

When conversion is performed on the Coq side, every manipulation of the formula is actually a logical rewriting step and ends up in the proof term. Each rewriting step contains the whole context in which it is performed, therefore each step is linear in the size of the whole formula. Moreover, it is very slow in practice because the matching and rewriting mechanism, which is used to rewrite the formula adequately, is not very efficient. Altogether, this CNF conversion can yield really big proof terms on average-sized formulae and it easily ends up taking much longer than the proof search itself. Performing the CNF conversion on the abstract side, however, can be summarized in the following way:

- we implement a function `conversion : formula` $\rightarrow$ `formula` that transforms an abstract formula as wanted;
- we show that for all formula `F`, `conversion F` is in CNF and is equivalent (or at least equisatisfiable) to `F` itself.

This method ensures that CNF conversion takes a constant, thus neglectible, size in the final proof term, and can be performed efficiently since it is executed by Coq's virtual machine.

Once we decide to implement the CNF conversion as a function on abstract formulae, there are different well-known techniques that can be considered and that we implemented.

1. The first possibility is to do a naive, traditional, CNF conversion that uses de Morgan laws in order to push negations through the formula to the atoms' level, and distributes disjunctions over conjunctions until the formula is in CNF. For instance, this method would transform the formula $A \vee (B \wedge C)$ in $(A \vee B) \wedge (A \vee C)$. It is well-known that the resulting formula can be exponentially bigger than the original.

2. Another technique that avoids the exponential blow-up of the naive conversion is to use Tseitin's conversion [Tse68]. It adds intermediate variables for subformulae and *definitional clauses* for these variables such that the size of the resulting CNF formula is linear in the size of the input. On the $A \vee (B \wedge C)$ formula above, this method returns $(A \vee X) \wedge (\bar{X} \vee B) \wedge (\bar{X} \vee C) \wedge (X \vee \bar{B} \vee \bar{C})$ where X is a new variable.

3. A refinement of the previous technique is to first convert the formula to negation normal form and use Plaisted and Greenbaum's CNF conversion [PG86] to add half as many definitional clauses for the Tseitin variables. In the above example, the resulting formula is $(A \vee X) \wedge (\bar{X} \vee B) \wedge (\bar{X} \vee C)$.

The Need for Another CNF Conversion. The CNF conversion techniques that we have considered so far remain unsatisfactory. The first one can cause an exponential increase in the size of the formula, and the other two add many new variables and clauses to the problem. All of them also fail to preserve the high-level logical structure of the input formula, in that sense they make the problem more difficult than it was originally. There has been lots of work on more advanced CNF conversion techniques but their implementation in Coq raises some issues. For instance, Plaisted and Greenbaum's method was originally intended to preserve the structure of formulae, but in order to do so, it requires that equal subformulae be shared. Other optimization techniques [NRW98, dlT90] are based on renaming parts of the subformula to increase the potential sharing. However, it is hard to implement such methods efficiently as a Coq function, *i.e.* in a pure applicative setting with structural recursion. Even implementing and proving the standard Tseitin conversion proved to be much more challenging than one would normally expect.

For the same reason, it is undeniable that our reflexive Coq decision procedure cannot reach the same level of sheer performance and tuning than state-of-the-art SAT solvers, which means that we cannot afford a CNF conversion that adds too many variables, disrupts the structure of the formula, in a word that makes a given problem look harder than it actually is. Results presented in Section 7.4 show that this concern is justified. Constraints due to CNF conversion also arise in Isabelle where formulae sent to the Metis prover are limited to 64 clauses. In the description of their Simplify theorem prover [DNS05], Nelson *et al.* describe a lazy CNF conversion method they designed in order to prevent the performance loss due to Tseitin-style CNF conversion. Their experience was that "introducing lazy CNF into Simplify avoided such a host of performance problems that [..] it converted a prover that didn't work in one that did." In the next sections of this chapter, we describe how we formalized and integrated this lazy CNF conversion mechanism in our DPLL-based tactic. To our knowledge, this work represents the first effort at a formal description and proof of this method.

7.2 A DPLL Procedure with Lazy CNF Conversion

In this section, we formally describe how a DPLL procedure can be adapted to deal with literals that represent arbitrary formulae.

7.2.1 Expandable Literals

In a Tseitin-style CNF conversion, new literals are added that represent subformulae of the original formula. To denote this fact, clauses must be added to the problem that link the new literals to the corresponding subformulae. The idea behind lazy CNF conversion is that new literals should not merely *represent* subformulae, but they should *be* the subformulae themselves. This way, there would be no need for additional definitional clauses. Detlefs et al. [DNS05] present things a bit differently, using a separate set of *definitions* for new variables (which they call *proxies*), and make sure the definitions of a given proxy variable are only added to the current context when this variable is assigned a boolean value by the procedure. Our abstraction will require less changes to the DPLL procedure.

In order for literals to be able to stand for arbitrary complex subformulae, we extend the signature of literals given in Fig. 6.1 page 132 in the following way:

```
Module Type EXPLITERAL.
  (* Negation, OrderedType... as before *)
  Include Type LITERAL.
  (* Expansion *)
  Parameter expand : t → list (list t).
  ...
End EXPLITERAL.
```

In other words, expandable literals always come with negation, comparison, and various properties, which are copied from the `LITERAL` signature using the `Include Type` capability, but they have an additional *expansion* function, named `expand`, which takes a literal and returns a list of lists of literals, in other words a CNF of literals. For a genuine literal which just stands for itself, this list is simply the empty list. For another literal that stands for a formula F, *i.e.* a proxy F, this function allows one to unfold this literal and reveal the underlying structure of F. This underlying structure must be expressed as a conjunction (list) of disjunctions (lists) of literals, but since these literals are also expandable literals, they can stand for subformulae of F themselves. Therefore, this CNF does not have to be the full conjunctive normal form of F: `expand` can undress the logical structure of F one layer at a time, using proxy literals to represent the direct subformulae of F. This means that the CNF conversion of formula F can be performed step after step, in a *call-by-need* fashion. In [DNS05], the `expand` function would be a look-up in the set of proxy definitions.

As an example, let us consider the formula $A \vee (B \wedge C)$ once again. A proxy literal for this formula could expand to its full CNF, namely the list of lists $[[A; B]; [A; C]]$. But more interestingly, it may also reveal only one layer at a time and expand to the simpler list $[[A; X]]$, where X itself expands to $[[B]; [C]]$. Note that this variable X is not a new variable in the sense of Tseitin conversion, it is just a way to denote the *unique* literal that expands to $[[B]; [C]]$, and which therefore stands for the formula $B \wedge C$. This unicity will be the key to the structural sharing provided by this method. In Section 7.3, we will describe how these expandable literals can be implemented in such a way that common operations are reasonably efficient, but for now let us see how the DPLL procedure should be adapted.

7.2.2 Adaptation of the DPLL Procedure

In order to use expandable literals in the DPLL procedure, we have to adapt the inference rules presented in Fig. 2.1 page 25, which we later formalized in Chapter 6. Let us consider a proxy literal f for a formula F. If this proxy is assigned a true value at some point during the proof search, this means that the formula F is assumed to be true. Therefore, something should be added to the current problem that reflects this fact in order to preserve the semantic soundness of the procedure. To this end, we use the `expand` function on f in order to unveil the structure of F, and add the resulting list of clauses `expand`(f) to the current problem.

The revised version of our inference rules system is given in Fig. 7.1. The only modifications between this system and the one presented in Fig. 2.1 concern rules which change the current assignment Γ : Assume and Split. When a literal l is assumed in the current context, it is expanded and the resulting clauses are added to the current problem Δ. Intuitively, if l is a proxy for F, `expand`(l) can be seen as "consequences" of F and must be added in order to reflect the fact that F shall now be satisfied. Now, given

$$\textsc{Assume}\ \frac{\Gamma, l \vdash \Delta, \texttt{expand}(l)}{\Gamma \vdash \Delta, l} \qquad \textsc{Red}\ \frac{\Gamma, l \vdash \Delta, C}{\Gamma, l \vdash \Delta, \bar{l} \vee C}$$

$$\textsc{Elim}\ \frac{\Gamma, l \vdash \Delta}{\Gamma, l \vdash \Delta, l \vee C} \qquad \textsc{Conflict}\ \frac{}{\Gamma \vdash \Delta, \emptyset}$$

$$\textsc{Split}\ \frac{\Gamma, l \vdash \Delta, \texttt{expand}(l) \qquad \Gamma, \bar{l} \vdash \Delta, \texttt{expand}(\bar{l})}{\Gamma \vdash \Delta}$$

Figure 7.1: The DPLL procedure adapted to expandable literals

an arbitrary formula F, instead of explicitely converting it into a CNF Δ_F and searching a derivation for $\emptyset \vdash \Delta_F$, it is enough to build a proxy literal l_F for F and attempt to find a derivation for $\emptyset \vdash l_F$ instead. This allows us to use a DPLL decision procedure with the lazy conversion mechanism. Note that correctness does not require proxy literals to be added to the current assignment Γ; however, doing so has a dramatic effect on formulae that can benefit from sharing, *e.g.* $l \wedge \neg l$, where l stands for a big formula F: in that case, adding the proxy literal l to the assignment will allow the elimination of $\neg l$ in one single step. Such formulae are not as anecdotal as they seem, and we discuss this further in Section 7.4.2.

We spent most of Chapter 6 describing how to formalize DPLL's basic inference system in Coq, proving its correctness and implementing a computable strategy to use in a reflexive tactic. In order to adapt these constructions to this new DPLL system with expandable literals, there are quite a few changes that must be made, but there is nothing fundamentally different in the method and the approach followed. Therefore we do not detail these changes but the most important can be summarized as follows:

- the definition of `derivable`, the inductive inference system, must be adapted as above with the expansion of assumed literals in Γ;

- proofs must be adapted, but are very close to the original proofs; one of the main differences is that, in order to be well-formed, partial assignments not only need to be consistent with the negation of literals, but also with their expansion, which is guaranteed by the strategy used;

- the semantics must be adapted so that models now account for proxy literals: if a proxy for `F` is in a model `M`, then `M` must satisfy `F`; in other words, models are exactly determined by their non-proxy literals;

- the proof search procedure must expand literals properly and its proofs must be extended;

- on the front-end, when the tactic fails, only non-proxy literals in the countermodel are displayed to the user.

The most interesting and difficult point is how to adapt the implementation of literals to expandable literals, and is the topic of the next section.

7.3 Implementing Lazy Literals in Coq

In this section, we show how to design a suitable literal module on which we can instantiate the procedure we described in Section 7.2.2.

7.3.1 Raw Expandable Literals

Expandable literals are either standard propositional atoms, or proxies for a more complex formula. Because a proxy shall be uniquely determined by its expansion (in other words, proxies that expand to the same formula stand for the same formula, and therefore should be equal), we choose to directly represent proxies as their expansion. Also, the implementation of expandable literals can be defined in a way that does not depend on the representation of the actual non-proxy literals. In other words, we suppose we are given a module `L` of traditional literals as defined in the previous chapter, and we implement expandable literals as a functor parameterized by `L`. This leads us to the following definition of raw expandable literals as a Coq inductive type:

```
Module RAW (L : LITERAL).
  Inductive t : Type :=
  | Proxy (pos neg : list (list t))
  | Lit (l : L.t).
  ...
End RAW.
```

Standard literals are represented by the `Lit` constructor which takes a literal `L.t` as argument. More interestingly, the `Proxy` constructor expects two arguments: the first one represents the formula that the proxy literal stands for, while the other one corresponds to the expansion of its negation. We proceed this way in order to be able to compute the negation of a literal in constant time, whether it is a proxy or not. Thus, the second parameter of `Proxy` should just be seen as a memoization of the negation function. As a matter of fact, we can easily define the negation function:

```
  Definition mk_not (l : t) : t :=
    match l with
      | Proxy pos neg ⇒ Proxy neg pos
      | Lit l ⇒ Lit (L.mk_not l)
    end.
```

Negating a standard literal is just done via a call to `L.mk_not`, while negating a proxy amounts to swapping its arguments. This memoization of the negation of a proxy literal is really critical for the efficiency of the method because literals are negated many times over the course of the DPLL proof search. In Section 7.3.3, we will show how these proxies are created in linear time.

The implementation of the expansion function is straightforward and requires no further comment:

```
Definition expand (l : t) : list (list t) :=
  match l with
    | Proxy pos _ ⇒ pos
    | Lit _ ⇒ []
  end.
```

We are left with implementing an instance of `OrderedType` for these literals. For instance, the total comparison function goes like this :

```
Fixpoint compare (x y : t) : comparison :=
  match x, y with
    | Lit l, Lit l' ⇒ l =?= l'
    | Lit _, Proxy _ _ ⇒ Lt
    | Proxy _ _, Lit _ ⇒ Gt
    | Proxy xpos xneg, Proxy ypos yneg ⇒
      compare_list_list compare xpos ypos
  end.
```

Recall that the notation `l =?= l'`, introduced in Chapter 5, is the effective comparison of two elements `l` and `l'`; we can use it here because the base literals' module `L` brings an instance of `OrderedType` for `L.t`. In this definition, `compare_list_list` recursively applies the comparison function `compare` in a lexicographic manner to lists of lists of literals. The part that is worth noticing is that we only compare proxies' first component and we skip the negated part. This of course ensures that the comparison of proxies is linear in the size of the formula they stand for; had we compared the second component as well, it would have been exponential in practice. The issue with such optimizations is that we have to convince Coq that they make sense, and the next section is devoted to that point.

7.3.2 Adding Invariants to Raw Literals

When implementing expandable literals in the previous section, we made a strong implicit assumption about a proxy `Proxy pos neg`, namely that `neg` was indeed containing the "negation" of `pos`. We need to give a formal meaning to this sentence and to ensure this invariant is verified by all literals. It is not only needed for semantical proofs about literals and the DPLL procedure, but for the correctness of the simplest operations on literals, starting with comparisons. Indeed, considering the comparison function `compare` presented above, it should verify the properties required by `OrderedType` and by `Literal` in general, in particular the following should be true:

```
compare x y = Eq ↔ compare (mk_not x) (mk_not y) = Eq
```

for all literals x and y, since `compare` should return `Eq` if and only if its arguments are equal, and negation should be a morphism for equality (axiom `mk_not_compat` in signature `LITERAL`). Proving this property for standard literals is straightforward, but as far as proxies are concerned, the fact that the equality test returns true only tells us that the first component of the proxies are equal: there is no guarantee whatsoever on the second component. Therefore, this property is not provable as is and we need to add some relation between the two components of a proxy. This relation also ought to be symmetric since the `mk_not` function swaps the first and second components and should of course preserve the invariant as well.

We are going to link the two components of a proxy literal by ensuring that each one is the image of the other by an adequate function $\mathcal{N}$. Intuitively, this function $\mathcal{N}$ must negate a conjunction of disjunction of literals and return another conjunction of disjunction of literals; it can be recursively defined in the following way

$$\mathcal{N}((\bigvee_{i=1}^{n} x_i) \wedge C) = \bigwedge_{i=1}^{n} \bigwedge_{D \in \mathcal{N}(C)} (\bar{x}_i \vee D)$$

where the x_i are literals and C is a CNF formula. Once this function is implemented, we can define an inductive predicate that specifies *well-formed* literals:

```
Inductive wf_lit : t → Prop :=
| wf_lit_lit : ∀l, wf_lit (Lit l)
| wf_lit_proxy : ∀pos neg, 𝒩 pos = neg → 𝒩 neg = pos →
    (∀l t, l ∈ pos → t ∈ l → wf_lit t) →
    (∀l t, l ∈ neg → t ∈ l → wf_lit t) →
    wf_lit (Proxy pos neg).
```

The first constructor expresses that all atomic literals are well-formed. The second one brings up requirements on proxy literals: not only should the two components be each other's image by $\mathcal{N}$[1], but all literals appearing in these expansions should recursively be well-formed. In particular, if two proxies are well-formed, their second components are equal if and only if their first components are equal, which means that we can establish the needed properties about the comparison function.

Packing everything together. In Coq, one can use *dependent types* in order to define a type of objects that meet certain specifications. We use this

[1] This constraints the form of possible proxies since $\mathcal{N}$ is not involutive in general.

feature in our Coq development in order to define a module of well-formed expandable literals. Using functors once again, we defined this module as a functor parameterized by `L : LITERAL` which uses the `RAW` functor seen above. In this functor, we define the type of literals as the dependent type of raw literals packed with a proof that they are well-formed:

```
Module LLAZYFY (L : LITERAL) <: EXPLITERAL.
  (* Imports all the raw definitions *)
  Module Import RAW := RAW L.

  Definition t : Type := {l | wf_lit l}.
  ...
End LLAZYFY.
```

We then have to redefine the required operations on literals. In most cases, it is just a matter of "lifting" to well-formed literals the definition we made for raw literals by showing that the operation preserves well-formedness. For instance, the negation function is (re)defined this way:

```
  Property wf_mk_not : ∀l, wf_lit l → wf_lit (mk_not l).
  Proof. ..... Qed.
  Definition mk_not (l : t) : t :=
    exist (mk_not π1(l)) (wf_mk_not π1(l) π2(l)).
```

where π_1 and π_2 respectively access to the raw literal and its well-formedness proof in a well-formed literal. We have presented a simplified version here and the real development contains more invariants that are required throughout various proofs about literals and their operations. In particular, in order to enable the definition of recursive functions over the structure of expandable literals, or simply guarantee the termination of the proof search, we had to add a notion of `size` of literals, along with proofs that the literals appearing in the expansion of a proxy are smaller than the proxy itself. Altogether, we obtain a module with the signature of literals as expected by the DPLL procedure, and where every operation is totally certified.

7.3.3 Converting Formulae to Lazy Literals

Once we have a module implementing lazy literals as described above, we are left with the task of constructing such literals out of an input formula.

First, note that we should not build arbitrary literals but only literals that are well-formed. Therefore we have to make sure that the proxies we build respect the invariants that we introduced in the last section. Assume we want to build a proxy for a formula $F = F_1 \vee F_2$ and we know how to build proxies l_1 and l_2 for the formulae F_1 and F_2. A suitable proxy for F is the one that expands positively to the list $[[l_1; l_2]]$, and to the list $[[\bar{l_1}]; [\bar{l_2}]]$ negatively. We can check that these two lists are indeed each other's image

Proxy	*pos*	*neg*
$X \equiv P$	$\{P\}$	$\{\bar{P}\}$
$X \equiv F \vee G$	$\{F \vee G\}$	$\{\bar{F}\}\{\bar{G}\}$
$X \equiv F \wedge G$	$\{F\}\{G\}$	$\{\bar{F} \vee \bar{G}\}$
$X \equiv (F \rightarrow G)$	$\{\bar{F} \vee G\}$	$\{F\}\{\bar{G}\}$
$X \equiv (F_1 \vee F_2 \vee \ldots \vee F_n)$	$\{F_1 \vee F_2 \vee \ldots \vee F_n\}$	$\{\bar{F_1}\}\{\bar{F_2}\}\ldots\{\bar{F_n}\}$
$X \equiv (F_1 \wedge F_2 \wedge \ldots \wedge F_n)$	$\{F_1\}\{F_2\}\ldots\{F_n\}$	$\{\bar{F_1} \vee \bar{F_2} \vee \ldots \vee \bar{F_n}\}$

Figure 7.2: Proxy construction for each logical connective

by $\mathcal{N}$. In practice, we define a function constructing such a proxy and we prove that its result is well-formed:

```
Definition mk_or_aux f g :=
  Proxy [[f;g]] [[mk_not f];[mk_not g]].
Property wf_mk_or : ∀(l l' : t), wf_lit (mk_or_aux l l').
Proof. ..... Qed.
Definition mk_or f g : t :=
  exist (mk_or_aux f g) (wf_mk_or f g).
```

The last command uses `mk_or_aux` and `wf_mk_or` to define a function that creates a well-formed proxy literal for the disjunction of two well-formed literals. We create such smart constructors for each logical connective: the table in Fig. 7.2 sums up how proxies are constructed for the usual logical connectives. Creating a proxy for an arbitrary formula is then only a matter of recursively applying these smart constructors by following the structure of the formula. We have implemented such a function named `mk_form` and proved that for every formula `F`, `mk_form F` $\leftrightarrow$ `F`. This theorem is very important since it is the first step that must be done when applying the tactic: it allows us to replace the current formula by a proxy before calling the DPLL proof search. Note that the converted formula is *equivalent* to the original because no new variables have been added, whereas with Tseitin-like methods, the converted formula is only *equisatisfiable.* Note also that the proxies constructed for $\bar{F} \vee G$ and $F \rightarrow G$ are equal, and so are $F \vee G$ and $G \vee F$ for instance, therefore the proxy construction not only identifies formulae that are syntactically equal, but also sometimes semantically.

Constructing proxies for N-ary operators. Figure 7.2 also contains proxy definitions for n-ary versions of the $\wedge$ and $\vee$ operators. We have implemented an alternative version of the `mk_form` function above which tries to add as few levels of proxies as possible. When constructing a proxy for a disjunction (resp. conjunction), it tries to regroup all the disjunctive (resp. conjunctive) top-level structure in one single proxy. In this setting, equivalences are interpreted either as conjunctions or as disjunctions[2] in order to minimize the number of proxies.

7.4 Results and Discussion

7.4.1 Benchmarks

	tauto	CNF$_C$	CNF$_A$	Tseitin	Tseitin2	Lazy	LazyN
hole3	–	0.72	0.06	0.24	0.21	0.06	**0.05**
hole4	–	3.1	0.23	3.5	6.8	0.32	**0.21**
hole5	–	10	2.7	80	–	1.9	**1.8**
deb5	83	–	0.04	0.15	0.10	0.09	**0.03**
deb10	–	–	0.10	0.68	0.43	0.66	**0.09**
deb20	–	–	**0.35**	4.5	2.5	7.5	**0.35**
equiv2	0.03	–	0.06	1.5	1.0	**0.02**	**0.02**
equiv5	61	–	–	–	–	0.44	**0.42**
franzen10	0.25	16	0.05	0.05	0.03	**0.02**	**0.02**
franzen50	–	–	0.40	1.4	0.80	**0.34**	0.35
schwicht20	0.48	–	0.12	0.43	0.23	**0.10**	**0.10**
schwicht50	8.8	–	0.60	4.3	2.2	**0.57**	0.7
partage	–	–	–	13	19	**0.04**	0.06
partage2	–	–	–	–	–	0.12	**0.11**

Figure 7.3: Comparison of different tactics and CNF conversion methods. Timings are given in seconds and – denote time-outs (>120s).

We benchmarked our tactic and the different CNF conversion methods on valid and unsatisfiable formulae described by Dyckhoff [Dyc97]; for instance *holen* stands for the pigeon-hole formula with n holes. We used two extra special formulae in order to test sharing of subformulae : *partage* is the formula $hole3 \wedge \neg hole3$, while *partage*2 is *deb*3 where atoms have been replaced by pigeon-hole formulae with varying sizes. Results are summarized in Fig. 7.3, where CNF$_C$ and CNF$_A$ are naive translations respectively on the Coq side (*i.e.* with rewriting steps) and on the abstract side (*i.e.* through a Coq function), Tseitin and Tseitin2 are the two variants of Tseitin conversion described in Section 7.1. The last two columns, Lazy and LazyN, are devoted to our lazy conversion, with only LazyN using proxies for n-ary operators. On each line, the best timings are emphasized with bold typeface. These results show that our tactic outperforms `tauto` in every single case (see discussion below for differences between our tactic and `tauto`), solving in less than a second goals that were beyond reach with the existing tactic. About the different CNF conversions, it turns out that the Tseitin conversion is almost always worse than the naive abstract CNF conversion

[2] The equivalence $F \leftrightarrow G$ is logically equivalent to the conjunction $(F \to G) \wedge (G \to F)$ and the disjunction $(F \wedge G) \vee (\bar{F} \wedge \bar{G})$.

because of the extra clauses and variables. The lazy tactics always perform at least as well as CNF$_A$ and in almost all cases they perform much better, especially when some sharing is required.

7.4.2 Discussion and Limitations

Comparison with `tauto/intuition`. As explained in Chapter 4, the tactic `tauto` is actually a customized version of the tactic `intuition`. When it can't solve a goal completely, `intuition` is able to take advantage of the search-tree built by its decision procedure in order to simplify the current goal in a set of (simpler) subgoals; `tauto` simply calls intuition and fails if any subgoals are generated. Unlike `intuition`, our tactic is unable to return a simplified goal when it cannot solve it completely, and in that sense it can be considered as less powerful. However, `intuition`'s performance often becomes an issue in practice[3], therefore we are convinced that the two tactics can prove really complementary in practice, with `intuition` being used as a simplifier and `unsat` as a solver.

Classical reasoning in an intuitionistic setting. The DPLL procedure is used to decide classical propositional logic whereas Coq's logic is intuitionistic. In our development, we took great care in not using the excluded-middle for our proofs so that Coq users who do not want to assume the excluded-middle in their development can still use our tactic. The reason we were able to do so lies in the observation that the formula $\forall A. \neg\neg(A \vee \neg A)$ is intuitionnistically provable: when the current goal is `False`, this lemma can be applied to add an arbitrary number of ground instances of the excluded-middle to the context. In other words, if a ground formula Φ is a classical tautology, $\neg\neg\Phi$ is an intuitionistic tautology[4]. Noticing that $\neg\neg\neg\Phi$ implies $\neg\Phi$ in intuitionistic logic, this means that if $\neg\Phi$ is classically valid, it is also a tautology in intuitionistic logic. Because the DPLL procedure proceeds by refuting the context Φ, *i.e.* proving $\neg\Phi$, we can use it in intuitionsitic reasoning even if it relies on classical reasoning.

In practice, the use of classical reasoning in our development is mainly for the correctness of the Split rule and of the different CNF conversion rules (*e.g.* $F \to G \equiv \bar{F} \vee G$). This led us to proving many intermediate results and lemmas in double-negation style because they were depending on some classical reasoning steps[5], but the nice consequence is that our tactic

[3] As Coq users, we often let `tauto` run for a few seconds to try and make sure that a goal is provable. When `tauto` succeeds, albeit not immediately, we then proceed to manually prove the goal or simplify it in easier subgoals.

[4] This is not true for first-order formulae, because the formula $\neg\neg(\forall A. A \vee \neg A)$, where the quantification lies below the double negation, is not intuitionnistically provable.

[5] Typically, see the characterization of the totality of a model on page 134, in the semantics of formulae: we use `∀M l, ~~(~(M l) → M (mk_not l))` instead of the simpler `∀M l, ~(M l) → M (mk_not l).`

produces intuitionistic refutation proofs and thus can really replace `tauto` when the context becomes inconsistent. Users of classical reasoning can use our tactic for classical validity by simply refuting the negation of the current goal, as explained in Chapter 6.

Impact of sharing. The results presented above show that the number of proxies has less effect on the performance than the sharing they provide. Depending on the formula, it may not be the best idea to minimize the number of proxies as LazyN does, because this minimizes the number of subformulae that are shared. Once again, we can use our modular development to provide these different alternatives as options to the user. We wrote in Section 7.2.2 that adding proxies to the current assignment made it possible to reduce a whole subformula of a problem in one single step, and this is why sharing is beneficial. We gave the obvious, rather crafted, example of $l \wedge \bar{l}$ where l is a big formula, but there is a less obvious and much more frequent situation where it happens. Practical formalizations often involve predicate definitions $p(x_1, \ldots, x_n) = \Phi(x_1, \ldots, x_n)$ where Φ can be a big formula, p is then used as a shortcut for Φ throughout the proofs. Now, when calling a DPLL procedure, one has to decide whether occurrences of p should be considered as atoms or whether they should be unfolded to Φ. There is no perfect strategy, since proofs sometimes depend on p being unfolded and sometimes do not; there is a conservative strategy since always unfolding p suffices, but it leads to performance losses if it wasn't required. Proxies make the DPLL procedure completely oblivious to such intermediate definitions, and this is a great asset when dealing with proof obligations from program verification.

7.4.3 Application to Other Systems

The advantages of the CNF conversion that we have implemented go beyond the scope of our tactic. It generally allows subformulae to be structurally shared which can give a big performance boost to the procedure. Moreover, in standard programming languages, proxies can be compared in constant time by using *hash-consing* [FC06], which removes the main cost of using lazy literals.

Lazy literals also provide a solution to a problem that is specific to SMT solvers: definitional clauses due to Tseitin-style variables appearing in contexts where they are not relevant can not only cause the DPLL procedure to perform many useless splits, but they also add ground terms that can be used to generate instances of lemmas. De Moura and Bjorner report on this issue in [dMB07], where they use a notion of *relevancy* in order to only consider definitional clauses at the right time. Lazy CNF conversion is a solution to this issue, and it is the method we currently use in our own prover Alt-Ergo.

Finally, one may wonder whether this method can be adapted to state-of-the-art decision procedures, including common optimizations like back-jumping and conflict clause learning. Adapting such procedures can be done in the same way that we adapted the basic DPLL and is really straightforward; an interesting question though is the potential impact that lazy CNF conversion could have on the dependency analysis behind these optimizations. We have not thoroughly studied this question but our experience with Alt-Ergo suggests that lazy CNF conversion remains a very good asset even with a more optimized DPLL.

7.5 Conclusion

We have presented how our reflexive tactic for propositional logic presented in the previous chapter can be adapted to use a lazy conversion scheme in order to bring arbitary formulae into clausal form without deteriorating the performance of the procedure. We use this method in Alt-Ergo and it is very satisfactory to be able to formalize and verify it in the Coq proof assistant. It also turns out that this method brings very good results in the reflexive tactic as well and outperforms the other CNF conversion techniques that we have tried.

CHAPTER 8

From Propositional Logic to Theory Reasoning

> L'Anglais est un praticien qui n'a pas de théories ;
> l'Allemand, un théoricien qui applique ses théories ;
> le Français, un théoricien qui ne les applique pas :
> c'est ce qu'on appelle chez nous avoir du bon sens.
>
> Antoine Detœuf (1902)

Contents

In this chapter, we show how to extend our formalization of a reflexive propositional tactic in order to introduce theory reasoning, as described in Section 2.3. We start in Section 8.1 by adapting our DPLL formalization and procedure to accept generalized environments instead of simple partial assignments. Then, in Section 8.2, we address an issue which is specific to Coq and our reflexive approach, namely the issue of reifying not only propositional variables but equalities between terms in an arbitrary signature, and finally their semantics in Section 8.3.

8.1 A Generalized Environment for DPLL

8.1.1 Environments

In Section 2.3, we described how the DPLL procedure can be generalized by replacing the partial assignment with a notion of *environment*, thus allowing the procedure to be used to solve the SMT problem rather than just the SAT problem. We now formalize this approach and start by the definition of the signature of environments. Recall that we described environments in Section 2.3 as data structures which provide *assumption* and *query* operations in order to add literals and check the truth value of a literal. Our formalization of the signature of environments follows this description:

```
Module Type ENV_INTERFACE (Import F : CNF).
  Parameter t : Type.

  Parameter empty : t.
  Parameter assume : L.t → t → Exception t.
  Parameter query : L.t → t → bool.
  Notation "e ⊨ l" := (query l e = true).


  ...
End ENV_INTERFACE.
```

The signature, called `ENV_INTERFACE`, is parameterized by a module of signature `CNF`, as described in the previous chapter. It provides the type t of environments and the two expected operations `assume` and `query`. It also provides the empty environment `empty`, otherwise it would be impossible to construct environments with that signature. We made the observation in Section 2.3 that the `assume` operation was a partial operation: indeed, adding a literal to the environment can make it inconsistent and in that case it cannot return a valid environment. To account for this, the return type of `assume` is `Exception` t, where `Exception` is just an “option” datatype defined like this:

```
Inductive Exception (A : Type) :=
| Normal (env : A)
| Inconsistent.
```

The interface of environments also introduces a handy notation for queries, namely $e \models l$ to denote that the query of l in e returns true. To complete `ENV_INTERFACE`, we need to add the necessary requirements on these operations, and in order to express these requirements, we need the set of literals which were explicitely assumed in an environment:

```
  Parameter assumed : t → L.lset.
  Axiom assumed_empty : assumed empty === ∅.
```

```
Axiom assumed_assume : ∀e l E,
  assume l e = Normal E → assumed E === {l; assumed e}.
```

The function which returns this set is called assumed and is completely specified by the assumed_empty and assumed_assume axioms. Now, we can express the requirements for the environment to be sound:

```
Axiom query_true : ∀e l, e ⊨ l →
   (∀M, Sem.submodel (assumed e) M → M l).
Axiom assumed_inconsistent : ∀e l,
  assume l e = Inconsistent → e ⊨ L.mk_not l.
```

The first axiom is the soundness of the query operation and expresses that if a query succeeds on l, it is indeed justified, in the sense that every model of the literals added to the environment is a model of l. This axiom is not sufficient and we add a second axiom for the soundness of the assume operation: it states that assuming l only returns Inconsistent if $\bar{l}$ is true in the environment.

Strictly speaking, the signature we have written so far is sufficient to describe sound environments, and as we explained in Section 6.2.3, we are only interested in the soundness of our procedure when developing a reflexive tactic. In practice, there is a part of the completeness of a procedure which we want to address nonetheless, and that is termination. More precisely, we need some reasonable completeness properties on our structure in order to ensure that some functions will behave correctly[1]. Here are the main two completeness properties which we require on environments:

```
Axiom query_assumed : ∀e l, l ∈ assumed e → e ⊨ l.
Axiom query_monotonic :
  ∀e e' l, assumed e ⊆ assumed e' → e ⊨ l → e' ⊨ l.
```

The first one ensures that assumed literals are true in the environment, while the second guarantees that assuming more literals can only make more literals true, not less.

8.1.2 A Simple Environment

We can give a simple example of an environment by encoding normal partial assignments as a module of signature ENV_INTERFACE.

```
Module ENV (Import F : CNF) <: ENV_INTERFACE F.
  Definition t := L.lset.
```

[1] Consider for instance the two versions of proof_search function described in Chapter 6: they use a natural integer in order to ensure termination, but we want to be able to call them with large enough integers in order to avoid unfinished computations. This is a part of the completeness theorem which is not strictly necessary but which we want to prove nonetheless, and it requires properties on the structures used: typically, once a literal has been supposed and the problem has been simplified by BCP, this literal should not appear anymore in the problem.

```
  Definition empty := ∅.
  Definition assume l e :=
    if mem (L.mk_not l) e then Inconsistent
    else Normal {l; e}.
  Definition query l e := mem l e.

  Definition assumed e := e.
  ...
End ENV.
```

The definition of the operations are self-explanatory and all the required properties are completely straightforward to prove. This environment can be seen as a "default" environment which allows to solve SAT modulo the trivial theory, *i.e.* satisfisability in propositional logic.

8.1.3 Adapting DPLL

With the signature of environments defined as above, we can adapt our formalization of DPLL to use environments, as we did with inference rules in Fig. 2.7 page 45. The functor `SAT`, which we introduced for the first time in Section 6.1.3, is adapted by adding an environment module as new parameter:

```
Module SAT (Import F : CNF)(Import E : ENV_INTERFACE F).
  Record sequent : Type := { G : E.t; D : L.cset }.

  Definition incompatible (S : sequent) : Prop :=
    Sem.incompatible (assumed (G S)) (D S).
  ...
End SAT.
```

The functor `SAT` is now parameterized by a `CNF` module and an environment module `E` for that `CNF` module (interestingly, notice how signature of parameters can depend on earlier parameters). The sequents are defined accordingly with an environment `E.t` in place of a partial assignment. Note how incompatibility of a sequent is rephrased using the set of literals assumed in the environment. The derivability predicate of such sequents is very similar to the one we have presented earlier; it is adapted as in Fig. 2.7 and starts like this:

```
  Inductive derivable : sequent → Prop :=
  | Conflict :
    ∀G D (i : ∅ ∈ D), derivable (G ⊢ D)
  | Assume :
    ∀G D l G', {l} ∈ D → assume l G = Normal G' →
      derivable (G' ⊢ (L.expand l) ∪ (D \ {l})) →
      derivable (G ⊢ D)
  | Elim :
    ∀G D l C, G ⊨ l → l ∈ C → C ∈ D →
      derivable (G ⊢ {D ~ C}) →
      derivable (G ⊢ D)
  ...
```

This excerpt shows that the `Conflict` rule does not change; more interestingly, the `Assume` rule, in order to extend the partial assignment, uses `assume` and can only be applied if the result of this assumption is not `Inconsistent`; finally, the `Elim` rule tests the state of a literal in the current environment by using `query` in order to eliminate a clause. The soundness proof of this notion of derivability is stated in the exact same way:

```
  Theorem soundness : ∀S, derivable S → incompatible S.
```

and is proved using the same reasonings, with the help of soundness properties from module `E` to replace earlier reasoning on partial assignments. The new return type of the proof search procedure is now:

```
  Inductive Res : Type :=
  | Sat : E.t → Res
  | Unsat.
```

where the countermodel in the `Sat` branch is an environment instead of a set of literals. The proof search strategies which we have described in Chapter 6 can be adapted very easily:

- when testing the status of a literal in the current assignment (*i.e.* environment), `query` must be used instead of set membership;
- when extending the current assignment with a literal, `assume` must be used and the case where this assumption returns `Inconsistent` must be treated properly.

We do not give more details on how the proof strategies, especially the efficient strategy presented in Section 6.3, are adapted to environments. Actual details can be quite tedious and verbose but are not particulary difficult. In the end, modules suitable for generating a reflexive tactic need to have the following `DPLL` signature:

```
Module Type DPLL (Import F : CNF)(E : ENV_INTERFACE F).
  Inductive Res : Type :=
  | Sat : E.t → Res
  | Unsat.

  Parameter dpll : formula → Res.
  Axiom dpll_correct :
    ∀f, dpll f = Unsat → Sem.incompatible ∅ (make f).
End DPLL.
```

which is really similar to the original DPLL interface. This emphasizes how little has to be changed to adapt the development of the tactic itself: reification and reflection theorems are unchanged, and the only modification is that when the tactic fails, the countermodel is now an environment and not simply a set of literals. The function assumed is used by the front-end to retrieve the literals which were explicitely assumed during the proof search and display them.

As a final remark, and a demonstration of the capabilities of the module sytem, note how a functor D with this DPLL signature can be instantiated with the basic environment functor ENV presented above in order to retrieve a functor with the old DPLL signature (as in Chapter 6):

```
Module NewDPLLasOld (F : CNF)(D : DPLL).
  Module OldE := ENV F.
  Include (D F OldE).
End NewDPLLasOld.
```

This functor takes the uninstantiated functor D and applies it to a F of signature CNF module and a basic environment for F. The result is included in the result module.

8.2 Beyond Literals: Terms and Reification

In the second part of this chapter, we detail how to adapt the reification process in order to go beyond simple propositional literals, which is a quite complex task. Indeed, in Chapters 9 and 10, we will build an environment for our DPLL reflexive tactic which will implement reasoning for the theory of equality modulo linear arithmetic. Consequently, we need to be able to reify formulae in the following grammar:

$$\begin{array}{lcl} F & := & p \mid T = T \mid \neg F \mid F \vee F \mid F \wedge F \mid F \rightarrow F \mid F \leftrightarrow F \\ T & := & f(T, \ldots, T) \end{array}$$

where p represents propositional variables and f function symbols. Reciprocally, we need to be able to interpret these reified objects back to their original counterparts, in a way similar to what we did with the interp function in Section 6.2.1. We already know how to reify propositional variables, the usual logical connectives, and define their interpretation. Unfortunately, the difficulty lies in the interpretation of terms and equalities: suppose we reify terms to a concrete datatype term, we need a function interp_term that interprets such an object back to the corresponding term, but what should its type be? There is no way to give such function a simple type since its type depends on the input: for example, if 0 is reified into `t0 : term` and *true* is reified into `ttrue : term`, `interp_term t0` shall have type `nat` and `interp_term ttrue` type `bool`. The only way to achieve this is to use dependent types and have a function with a type of the form:

```
interp_term : ∀(t : term), type_of t
```

where `type_of` returns the expected type of the object corresponding to a reified term. That being said, when interpreting a term of the form $f(T_1, \ldots, T_n)$, we need to interpret the T_i to concrete terms t_i of various types, and somehow apply the symbol f to these terms. The symbol f is itself reified (as were propositional variables) and must be interpreted to a concrete Coq entity which can be applied to the t_i. Even if we can program such functions using Coq's rich type system, there is no guarantee that a `term` corresponds to a well-typed concrete object and that its interpretation will succeed, in other words that the reified symbols represent symbols with the adequate types. To detect ill-formed reified terms, we need to be able to compare expected types and actual types during the interpretation of a term, and this is not possible if we use Coq's types directly. Hence, in order to be able to correctly reify terms with arbitrary types, we cannot use a *shallow embedding*, *i.e.* only reifying terms, but we will use a *deep embedding* of terms in the logic, *i.e.* reify both terms and their types.

8.2.1 Types

We will not reify all possible Coq types, in particular we only interpret non-dependent products ("arrows"), the type `Z` of relative integers, and consider all other types as atomic types, *i.e.* we reify them as variables, similarly to what we did with propositional variables. We define the following inductive datatype `type` for reified types:

```
Inductive type : Set :=
| typeCst (tidx : index)
| typeDefault
| typeArith
| typeArrow (_ _ : type).
```

The last two constructors correspond to the type of relative integers and to arrow types. The role of `typeDefault` is to serve as a default datatype used to make some functions total and which should not be used by the reification process. Finally, `typeCst` is used for a reified atomic type. Note that we again use an object of type `index` to denote a variable, which means that we use a `varmap` to interpret a `type` into a Coq type:

```
Definition type_env := varmap Type.
Inductive dummy : Set := mk_dummy.
Section TInterp.
```

```
  Variable vtypes : type_env.
  Fixpoint tinterp (t : type) : Type :=
    match t with
      | typeCst idx ⇒ varmap_find dummy idx vtypes
      | typeDefault ⇒ dummy
      | typeArith ⇒  Z
      | typeArrow t1 t2 ⇒ (tinterp t1) → (tinterp t2)
    end.
End TInterp.
```

We define `type_env` as the type of maps used to interpret reified types, *i.e.* as `varmap Type`, and we also define a new type `dummy` which is specific to the reification routine. The type interpretation function `tinterp` uses a map of type `type_env` and is defined in a section[2] where such a map is introduced. It is straightforward and simply proceeds by induction on the structure of the reified type, interprets arrows as arrows, integers as integers, and atomic types in the map using `varmap_find`. The special type `dummy` is used as a default, in particular when the lookup in the map fails; looking for `dummy` in a reified formula is then a way to easily spot problems in the reification process.

We also define an equality test for reified types, which would not have been possible with Coq's types:

```
Fixpoint tequal (t t' : type) : bool := ....
Property tequal_1 : ∀t t', tequal t t' = true → t = t'.
Property tequal_2 : ∀t t', t = t' → tequal t t' = true.
```

8.2.2 Symbols

In this subsection and in the following, we suppose we are in a Coq section where a variable `vtypes` of `type_env` is defined, as above, and our definitions will therefore be implicitely parameterized by `vtypes`. We use the notation ⟦`ty`⟧ to denote `tinterp ty vtypes`, the interpretation of a reified type `ty`.

We do not interpret any symbols except arithmetic constants and arithmetic operations. For other symbols, we need to proceed as with propositional variables and uninterpreted types, *i.e.* we need to store them in some kind of map and use indices in the map to represent these symbols. The problem is that symbols may have arbitrary types and therefore we cannot store them in one particular `varmap` (these are homogeneous); instead, we use, for each reified type, one varmap to store all symbols with that type, and we store all these varmaps in a single "varmap of varmaps". This leads to a double indirection, and each symbol must be represented with two indices: one to identify which varmap should be used, and the other to locate

[2] Coq's sectioning mechanism allow one to introduce *variables* which are generalized at the end of the section.

the symbol in that particular varmap. The type of reified symbols is defined as follows:

```
Inductive arithop : Set :=
| Plus | Minus | Opp | Mult.
Inductive symbol : Set :=
| Unint (ty_idx t_idx : index)
| Cst (z : Z)
| Op (op : arithop).
```

where uninterpreted symbols are encoded with a pair of indices, as explained above. We now need to formally define the "varmap of varmaps" used to interpret such symbols. Because the outer varmap needs to be homogeneous, each of the inner varmaps must have the same type and therefore we use the following dependent pair to denote the type of the inner varmaps:

```
Definition depvarmap := {ty : type & (⟦ty⟧ × varmap ⟦ty⟧)}.
Definition defvm : depvarmap :=
  existT _ typeDefault (mk_dummy, Empty_vm).
```

Such a `depvarmap` is a dependent pair whose first element is a reified type `ty`, and whose second element is a varmap containing values of type ⟦ty⟧ and an extra value of the same type which will be used as a default. An example of a `depvarmap` is given with the definition of `defvm`, a default varmap for the default type. The environment used to interpret symbols, called a `term_env` is then simply defined as:

```
Definition term_env := varmap depvarmap.
Variable v : term_env.
```

Note that `term_env` is a dependent type itself, since it implicitely depends on `vtypes` in this context. We now also introduce a variable `v : term_env` in the context and we can define the function which returns the (reified) type of a symbol:

```
Definition lookup_type (f : symbol) : type :=
  match f with
    | Unint ty_idx _ ⇒
      π1(varmap_find defvm ty_idx v)
    | Cst _ ⇒ typeArith
    | Op (Plus | Minus | Mult) ⇒
      typeArrow typeArith (typeArrow typeArith typeArith)
    | Op Opp ⇒ typeArrow typeArith typeArith
  end.
```

The types of arithmetic constants and operations do not require explanations, and the type of an uninterpreted symbol is found using its first index: we find the corresponding `depvarmap` in `v` using `varmap_find`, and use its first projection, *i.e.* the reified type. Now that we have this function, we can define the interpretation of a symbol, which we call `lookup`:

```
Definition lookup (f : symbol) : ⟦lookup_type f⟧ :=
  match f with
    | Unint ty_idx s_idx ⇒
      let (d, vs) :=
        π2(varmap_find defvm ty_idx v) in
        varmap_find d s_idx vs
    | Cst z ⇒ z
    | Op Plus ⇒ Zplus | Op Minus ⇒ Zminus ...
  end.
```

This function is dependently-typed and for all symbol `f`, returns an object whose type is the interpretation of the reified type `lookup_type f`. The interpretation of arithmetic symbols is straightforward, and as for uninterpreted symbols, the `depvarmap` containing the symbol is retrieved using the first index, its second component is retrieved with the projection π_2 and the second index is used to find the Coq value corresponding to the symbol in the inner varmap. Note that the default passed to that second `varmap_find` is the value stored along the inner varmap in the `depvarmap`. Coq is able to verify that this function indeed returns an object of type ⟦`lookup_type` f⟧ for all f.

8.2.3 Terms

Once symbols are defined, the type of reified terms is simply:

```
Inductive term : Set :=
| app (f : symbol) (lt : list term).
```

The expected type of a reified term can be defined by the following recursive[3] function:

```
Nested Fixpoint type_of (t : term) : type :=
  match t with
    | app f l ⇒ types_of l (lookup_type f)
  end
with types_of (l : terms) (ret : type) : type :=
  match l with
    | nil ⇒ ret
    | cons _ l ⇒
      match ret with
        | typeArrow _ t2 ⇒ types_of l t2
```

[3] This syntax for recursive functions is not standard: Coq normally does not allow fixpoints through a nested inductive (in this case, list) to be written in the usual way; we wrote an extension to allow this.

```
      | _ ⇒ typeDefault (* absurd *)
    end
  end.
```

This function finds the type of the term's head symbol and removes the types of its direct subterms in order to retrieve the type of the result. We now want to define the interpretation of reified terms, but we can only compute this interpretation if the reified terms are "well-typed". Therefore, we will compute the interpretation using an hypothesis that the reified term is well-typed. We thus start by writing a function `has_type t ty` which returns `true` if the reified term `t` has reified type `ty`:

```
Nested Fixpoint has_type (t : term) (ty : type) : bool :=
  match t with
    | app f l ⇒ have_type l (lookup_type f) ty
  end
with have_type (l : terms) (ty res : type) : bool :=
  match l with
    | nil ⇒ tequal ty res
    | cons t l ⇒
      match ty with
        | typeArrow t1 t2 ⇒ has_type t t1 &&& have_type l t2 res
        | _ ⇒ false
      end
  end.
```

The intuitive meaning of the mutually recursive function `have_type l ty res` is that it checks if a term of type `ty` can be applied to a list of arguments `l` to yield type `res`. We can now implement the interpretation of a reified term; this function takes a reified term `t`, a reified type `ty` and a proof that `has_type t ty = true`, and returns an object of type ⟦`ty`⟧. The implementation of this function `interp` is quite complex and is presented in Fig. 8.1. It uses a mutually recursive function `interps` which interprets a list of terms `l` such that `have_type l ty res = true` for some types `ty` and `res`, and passes them to an object of type ⟦`ty`⟧ to return an object of type ⟦`res`⟧.

We will not explain the implementation of `interp` in detail, but a couple of points are worth noticing. Most importantly, it uses an advanced feature of the CIC, dependent elimination: it allows one to write case analysis where the type in each branch depends on the value matched in the branch. We use dependent elimination here in order to take advantage of the hypothesis that the term is well-typed. For instance, in the body of `interp`, the term `t` is matched against `app f l` and in that branch, the hypothesis `H` of type `has_type t ty = true` becomes `has_type (app f l) ty = true`, which by definition of `has_type` is also a hypothesis of `have_type l (lookup_type f) ty`: this allows us to pass this hypothesis to `interps`

```
Nested Fixpoint interp (t : term) (ty : type)
  (H : has_type t ty = true) : ⟦ty⟧ :=
  (match t as a
   return has_type a ty = true → ⟦ty⟧ with
     | app f l ⇒
       fun H ⇒ interps l _ ty (lookup f) H
   end) H
with interps (l : terms) (ty res : type) (f : ⟦ty⟧)
  (H : have_type l ty res = true) : ⟦res⟧ :=
  (match l as a
   return have_type a ty res = true → ⟦res⟧ with
     | nil ⇒
       fun H ⇒ eq_rect _ (fun x ⇒ ⟦x⟧) f _ (tequal_1 _ _ H)
     | cons t lt ⇒
       (match ty as b
        return ⟦b⟧ → have_type (t::lt) b res = true → ⟦res⟧ with
          | typeArrow t1 t2 ⇒
            fun f H ⇒
              let H1 := proj1 (andb_prop _ _ H) in
              let H2 := proj2 (andb_prop _ _ H) in
              interps lt t2 res (f (interp t t1 H1)) H2
          | _ ⇒
            fun f ⇒ bdiscr ⟦res⟧
        end) f
   end) H.
```

Figure 8.1: Interpretation of reified terms

along with `lookup f`, which has type ⟦`lookup_type f`⟧. Similar dependent elimination in the body of `interps` allows us to dispatch absurd cases, for instance.

The interpretation of terms with an arbitrary signature is not easy to implement, and it can also be tedious to reason about. In particular, one must be very cautious in order to avoid using the proof-irrelevance axiom when reasoning about dependent constructs. We can prove a variety of useful results on term interpretation, like the fact that a term can only have one type (and if it exists, it is given by `type_of`), or the fact that the relation `interp_eq` between terms defined by the equality of their interpretations is a congruence relation. This relation is presented in Fig. 8.2. Note that only well-typed terms with the same type can be compared with equality, therefore in order to define an equality on all reified terms, we use the special `dom` structure as the result of interpretation. In particular, we consider all ill-

```
Inductive dom :=
| IllTyped
| Interpreted (ty : type) (v : ⟦ty⟧).
Definition int (t : term) : dom :=
  let ty := type_of t in
  (match has_type t ty as a
   return has_type t ty = a → dom with
     | true ⇒ fun H ⇒ Interpreted ty (interp t ty H)
     | false ⇒ fun _ ⇒ IllTyped
   end) (refl_equal _).
Definition interp_eq (t t' : term) := int t = int t'.
```

Figure 8.2: An equality relation on reified terms

typed terms to be equal (this is required by reflexivity), and an ill-typed term is never equal to a well-typed term. This relation is clearly an equivalence relation but the fact that it is congruent is a fundamental result since this allows us to use interpretation of literals as a model for equalities on terms, as we will see in Section 8.3. This is similar to the fact that we used the interpretation of propositional variables as a model for propositional literals.

Note that this interpretation of equalities is the source of another difference with respect to pure propositional reification: consider two concrete Coq terms `T` and `U` of the same type, and suppose `t` and `u` of type `term` are their reified counterparts; then `interp_eq t u` is not convertible to `T = U` because it reduces to an equality between elements of type `dom`. Nevertheless, `interp_eq t u` is equivalent to `T = U`, and more generally, when reifying formulae with equalities, the interpretation of the reified formula is equivalent, but not convertible, to the original.

8.2.4 Implementation

We implemented this reification mechanism with $\mathcal{L}$tac as we did for propositional literals in Section 6.2.1, but it is genuinely more complex. More precisely, reifying the formulae and equalities between terms required four different passes (two for the reification of types, two for the symbols and terms) and involved a great deal of advanced term matching. The resulting tactic was much too slow to be used in practice: it is not reasonable to have the reification process take longer than the proof search itself.

Therefore, we implemented an alternative version of the reification mechanism in OCaml, using Coq sources as an API. This requires manipulation of the internal constructs of Coq but allows for a fast reification process. The result is a tactic `ergo_reify f reif v` which takes a formula `f` in the context, whose type must be a proposition, reifies it and introduces new objects in the context: the reified formula `reif` and the varmaps `v` for interpreting the reified formula. For instance, reifying the formula `final` in the following context:

```
1 subgoal

A : Prop
x : Z
f : Z → Z
final : A ∧ f x = 3
============================
 False
```

```
ergo_reify final reif v.
```

will give the following reified formula, and the corresponding maps `v`:

```
1 subgoal

A : Prop
x : Z
f : Z → Z
final : A ∧ f x = 3
v := ...
reif := FAnd (FVar (Left_idx End_idx))
          (FEq
             (app
                (Unint (Left_idx End_idx) End_idx)
                ((app (Unint End_idx End_idx) nil)::nil)
             (app (Cst 3) nil))
============================
 False
```

8.3 New Literals, New Semantics

The use of literals extended with equalities on terms prevents us from using the modules for propositional literals `LPROP` and `CNFPROP` defined in Section 6.2.1 and Fig. 6.4 page 142, which were only designed for propositional variables. Thus we define a `LITERAL` module which embeds propositional variables and equalities, and the corresponding semantics module.

```
Module LITINDEX <: LITERAL.
  Inductive atom : Set :=
  | Atom (a : index)
  | Equation (u v : term).
  Definition t := atom × bool.
  Definition mk_not (l : t) := (fst l, negb (snd l)).
   ...
End LITINDEX.
```

In this module called LITINDEX, literals are propositional variables (*atoms*) or equalities between terms, along with their polarity. All the required properties are straightforward. Note that given this module of literals, we can obtain a module of expandable literals suitable for our lazy CNF conversion mechanism by using the LLAZIFY functor presented in Section 7.3.2. In order to interpret these literals, we need a varmap for propositional literals, and an environment for interpreting terms, *i.e.* a type_env and a corresponding term_env, as we explained above. We pack everything in a varmaps structure:

```
Record varmaps := {
  varmaps_lits : varmap Prop;
  varmaps_vty : type_env;
  varmaps_vsy : term_env varmaps_vty
}.
```

and we can define the interpretation of a literal parameterized by such a varmaps object:

```
Definition interp_atom (v : varmaps) (a : atom) : Prop :=
  match a with
    | Atom a ⇒ varmap_find True (varmaps_lits v) a
    | Equation s1 s2 ⇒
      interp_eq (varmaps_vty v) (varmaps_vsy v) s1 s2
  end.
Definition interp (v : varmaps) (l : t) : Prop :=
  match l with
    | (a, true) ⇒ interp_atom v a
    | (a, false) ⇒ ~ interp_atom v a
  end.
```

Note the use of interp_eq to interpret equalities. To define a semantics module for these literals, we naturally simply take varmaps as our type of models, and interpret a literal in a model using the interp function above:

```
Module SEM_INDEX <: SEM_INTERFACE LITINDEX.
  Definition model := varmaps.
  Definition model_as_fun : model → LITINDEX.t → Prop :=
    LITINDEX.interp.
  ...
End SEM_INDEX.
```

The properties required on these models are quite straightforward, but the interesting thing is that we can isolate the notion of model for equalities and prove that our notion of models indeed represent congruence relations:

```
Definition models_eq (M : model) a b := M (Equation a b, true).
Notation "M ⊨ a = b" := (models_eq M a b).
Property models_eq_refl : ∀M a, M ⊨ a = a.
Property models_eq_sym : ∀M a b, M ⊨ a = b → M ⊨ b = a.
...
```

These proofs rely on the fact that `interp_eq` is a congruence relation. We can define a similar notion for disequalities, noted `M ⊨ a ≠ b` and prove symmetry and antireflexivity for this relation. Finally, we add a notation for semantic entailment of equalities, namely `E ⊢ a = b` to denote that all models which satisfy the equations in `E` also model `a = b`:

```
Definition entails (E : list (term × term)) (a b : term) :=
  ∀M, (∀u v, In (a, b) E → M ⊨ u = v) → M ⊨ a = b.
Notation "E ⊢ a = b" := (entails E a b).
```

8.4 Conclusion

We have presented how to extend the formalized DPLL architecture seen in the previous chapters to an SMT architecture by replacing the environment. Thanks to the modular approach, the modifications are quite constrained and do not interfere with the top-level tactic or the low-level notions of literals and CNF formulae. We have also shown how to extend the reification to terms of arbitrary types, which is significantly harder than reifying expressions with constant types (propositional values for instance). The only similar reification development which we are aware of is presented by P. Corbineau in his thesis [Cor05] as an attempt at a semi-reflexive version of the `congruence` tactic, but this version has not been retained in Coq. We have also defined an interpretation for equalities of reified terms, which represents a congruence relation on terms and can be used as a notion of model in our semantics.

CHAPTER 9

Adding Equality Reasoning

Car s'il [les bestes] eüssent parleüre
Et reson pour eux s'antr'antandre
Qu'il s'antre peüssent aprandre
Mal fust aus homes avenu.

Jehan de Meung, *Le Roman de la Rose*
(XIIIe siècle)

Contents

The previous chapter was devoted to extending our DPLL architecture from a SAT architecture to an SMT architecture, by replacing partial assignments with a more general notion of environments. In this chapter, we implement an environment for the theory of equality modulo a theory X by following our system CC(X) presented in Chapter 3. Section 9.1 presents the formalization of solvable theories and we present the actual implementation of the CC(X) procedure in Section 9.2. We then show how to embed this implementation in an environment suitable for DPLL in Section 9.3. Finally, we conclude in Section 9.4 by showing a concrete application to the theory of equality with uninterpreted functions.

9.1 Theories

We start our Coq formalization of CC(X) by its basis component: theories. In Section 3.2.1, we described a theory as a collection of functions and later added the properties that these functions had to verify. We proceed in a similar manner here, and we start with the definition of the class of theories:

```
Class Theory := {
  R : Type;
  R_OT :> OrderedType R;
  (* Operations *)
  make : term → R;
  one : R;
  leaves : R → list R;
  subst : R → R → R → R;
  solve : R → R → Solution R
}.
```

In accordance with Definition 3.2.1 page 58, the class of theories provides an (ordered) type `R` of semantic values and the expected operations. The returned type of `solve` is a three-branch inductive:

```
Inductive Solution (R : Type) : Type :=
| Solved
| Unsolvable
| Subst (p : R) (P : R).
```

denoting the possible results of solving an equation. In order to describe the specifications of a theory, we now suppose we work in a context with some theory:

```
Section TheorySpecs.
  Context `{Th: Theory}.
  ...
```

and any later reference to `R`, `make`, ... will refer to that theory. We start by defining the `iter` function, which corresponds to the “iterated solving function” of Definition 3.2.2:

```
  Fixpoint iter (E : list (term × term)) : option (R → R) :=
    match E with
      | nil ⇒ Some (fun r ⇒ r)
    | ($t_1$, $t_2$)::E' ⇒
      match iter E' with
        | Some $f$ ⇒
          let $r_1$ := $f$ (make $t_1$) in
          let $r_2$ := $f$ (make $t_2$) in
            match solve $r_1$ $r_2$ with
              | Solved ⇒ Some $f$
              | Unsolvable ⇒ None
              | Subst $p$ P ⇒ Some (fun $r$ ⇒ subst $p$ P ($f$ $r$))
            end
        | None ⇒ None
      end
  end.
```

The definition follows exactly the one we made on page 59, with the exception that `iter` iterates on equations between terms instead of semantic values, but this choice is superficial and for convenience only. Note also that `iter` return's type is `option (R -> R)` and not `R -> option R`, thus emphasizing that `iter E r` succeeds either for all `r`, or for none, and this only depends on `E`. We can now formalize how a list of equations entails an equation between semantic values:

```
Definition implyX (E : list (term × term)) ($r_1$ $r_2$ : R) :=
  match iter E with
    | Some uf ⇒ uf $r_1$ === uf $r_2$
    | None ⇒ True
  end.
```

which corresponds to Definition 3.2.3. With these definitions, we can prove interesting properties, like the fact that `implyX E` is an equivalence relation which contains at least `E`, and is actually monotonic with respect to `E`. In order to specify the properties that `solve` must verify, we define an inductive relation `solve_specs` which links `solve`'s possible results and its arguments[1]:

```
Inductive solve_specs ($u$ $v$ : R) : Solution R → Prop :=
| solve_specs_Solved :
  $u$ === $v$ → solve_specs $u$ $v$ (Solved R)
| solve_specs_Unsolvable :
  $u$ =/= $v$ → (∀E, implyX E $u$ $v$ → iter E = None) →
  solve_specs $u$ $v$ (Unsolvable R)
| solve_specs_Subst :
  ∀$p$ P, $u$ =/= $v$ → subst $p$ P $u$ === subst $p$ P $v$ →
  $p$ ∉ (leaves P) → solve_specs $u$ $v$ (Subst $p$ P).
```

[1] It is similar to the way we specified the comparison function with the `compare_spec` relation in Chapter 5 page 112.

The various properties follow the Axioms 3.2.4 presented in Chapter 3. In particular, `solve u v` should return `Solved` if and only if `u === v` holds. Also, it should return `Unsolvable` if and only if the equation `u = v` can only be entailed by an inconsistent set of equations.

We are now ready to formally write the specifications that a theory must verify, we pack these specifications in the following class:

```
Class TheorySpecs := {
  (* leaves is never empty *)
  leaves_not_empty : ∀r, leaves r ≠ nil;
  (* properties of implyX *)
  ...
  implyX_entails :
    ∀E u v, implyX E (make u) (make v) → E ⊢ u = v;
  (* properties of solve *)
  solve_dec : ∀u v, solve_specs u v (solve u v);
  (* morphisms *)
  ...
}.
```

We see that `leaves` should never return an empty list, as in Definition 3.2.1, and that `solve` should verify the specification defined in `solve_specs`. The central property `implyX_entails` that we require on `implyX` is the one which makes it consistent with the semantic notion of equality entailment and corresponds exactly to Axiom 3.2.5 page 59. By itself, it guarantees that `make`, `subst` and `solve` are correctly describing the theory. Finally, there are a few other requirements in `TheorySpecs` which we do not detail, in particular proofs that `subst`, `leaves` and `solve` are all morphisms for equality on semantic values. We require less properties on theories than what we presented in Section 3.2.1, simply because we are only interested in soundness for the reflexive tactic, and some properties were only useful to prove the completeness of CC(X).

To conclude this presentation, we show how to define a module signature for theories by packing together an instance of a theory implementation and an instance of the corresponding specifications:

```
Module Type THEORY.
  Instance Th : Theory.
  Instance ThSpecs : TheorySpecs Th.
End THEORY.
```

Equality on Uninterpreted Functions. A simple basic case of a theory is equality modulo the “empty” theory, *i.e.* the theory $\mathcal{E}$ of equality with uninterpreted functions as defined in Definition 3.1.1 on page 52. We now define instances of `Theory` and `TheorySpecs` which implement this simple theory.

```
Instance Empty_theory : Theory := {
  R := term;
  R_OT := term_OT;
  make := fun t ⇒ t;
  one := app (Op Plus) nil;
  leaves := fun t ⇒ t::nil;
  subst := fun p P r ⇒ if p == r then P else r;
  solve := fun r1 r2 ⇒ if r1 == r2 then Solved _ else Subst r1 r2
}.
```

In this theory, the set of semantic values is simply `term` itself, and the `make` function is just the identity. Any term will do for `one` (and we use an ill-typed one) since it won't be used with this theory, and the leaves of a term `t` are simply `t` itself. The substitution `subst p P` simply maps `p` to `P`, and leaves any other term unchanged. Finally, an equation `u = v` is solved if `u` and `v` are equal, and yields the substitution pair `Subst u v` otherwise.

With these definitions, proving an instance of `TheorySpecs` is completely straightforward and we obtain a module `EmptyTheory` for the theory of equality on uninterpreted functions. Chapter 10 will be devoted to the implementation of a more interesting theory, the theory of linear arithmetic.

9.2 Implementing Congruence Closure

The implementation of our CC(X) decision procedure for congruence closure modulo a theory is derived from the formal presentation we made in Section 3.2.2. We do not describe a faithful formalization of the rules in Figures 3.3 and 3.4 (cf. pages 63 and 79) with soundness and completeness proofs, like we did for the SAT solver in Chapter 6, but an algorithm which is derived from these rules[2]. This algorithm is parameterized by a `Theory` as described in the previous section, and uses a variety of auxiliary data structures which correspond to the different elements of CC(X) configurations: the union-find Δ, the "use" relation Γ and a "diff" structure N to hold disequalities. We start by describing how we implement these data structures and their properties.

[2] We have actually implemented such a faithful Coq formalization of CC(X) as presented in Chapter 3 with soundness and completeness proofs, thus the system has been formally verified in Coq. However, that formalization is not adapted to usage in an efficient algorithm: for instance, it uses the set of all terms of the problem to ensure termination, which is not practical; another issue is the union-find Δ and mapping Γ which are described mathematically as total functions, but must be implemented as finite efficient structures.

9.2.1 Uf

The module Uf is a Coq file which implements a union-find structure of mapping terms to semantic values. It is parameterized by an instance of Theory, and uses the finite dictionaries of our containers library detailed in Chapter 5:

```
Section WithTheory.
  Context `{Th : Theory}.
  Structure t := mk_t {
    this :> Map[term, R];
    canon : term → R
  }.
  ...
End WithTheory.
```

The union-find structure is a record which contains a finite map from terms to values of type R, *i.e.* to semantic values. The coercion in the declaration of member this means that an object of type t can be used as if it was of type Map[term, R] and the projection will be added implicitely. The extra member canon is a function from terms to semantic values: it ensures that the structure can be used incrementally. In order to understand this point, recall that CC(X) configurations in Chapter 3 defined the union-find Δ as a function $\mathcal{T} \to \mathcal{R}$ from maps to terms; this ensured that when solving an equation and applying a substitution in Δ in the Congr rule, the substitution was applied to all terms in $\mathcal{T}$. Then, when "adding" a new term in the Add rule, its representative in Δ was already up-to-date with all the equations already merged. In practice, we want to calculate representatives progressively, storing them in the finite map this. We also store the full function in canon so that we can correctly calculate its representative when a new term is added to the structure. In other words, this can be seen as a memoization of canon on all the terms already treated by the algorithm. At the start of the algorithm, the canon function is simply make (as $\Delta_0(t)$ was defined as $[t]$ in CC(X)), and it is straightforward to define the following basic operations in Uf:

```
  Definition empty : t := mk_t [] make.

  Definition mem (m : t) (t : term) : R :=
    match m[t] with | Some _ ⇒ true | None ⇒ false end.

  Definition find (m : t) (t : term) : R :=
    match m[t] with
      | Some r ⇒ r
      | None ⇒ canon m t
    end.
```

```
Coercion find : t ↣ Funclass.
Definition congruent (m : t) (a b : term) : bool :=
  m a == m b.

Definition add (m : t) (a : term) : t :=
  match m[a] with
    | Some _ ⇒ m
    | None ⇒ mk_t (m[a ← canon m a]) (canon m)
  end.
```

In particular, `find m t` returns the representative of term `t` in the structure `m` (using `canon` if the term is not in the map), and the coercion lets us write `m t` directly: `congruent m a b` tests whether two terms have the same representative. Adding a term `t` to `m` leaves the structure unchanged if `t` was already known, and adds it using `canon` otherwise.

The last operation is the most important and allows one to `merge` two elements in the map:

```
Definition merge (m : t) (p P : R) : t :=
  mk_t (map (subst p P) m) (fun t ⇒ subst p P (canon m t)).
Definition merge' (m : t) (p P : R) : t × list term := ...
```

It merges `p` and `P` by applying the substitution `subst p P` to all terms in the map, and to `canon` as well. We also provide an alternate version `merge'` which also returns the list of terms in the map which were "touched" by the merge, *i.e.* the terms whose representative changed. We will see in Section 9.2.5 that this can be used to check for congruence equations faster in the algorithm.

Invariants and properties. We make the functions above *opaque* in `Uf`, which ensures encapsulation: outer modules using this structure cannot depend on the particular implementation. Instead, we prove all the necessary properties to reason on the functions above in the module `Uf`. The main thing is that a union-find map is only well-formed if `canon` is consistent with the map `this`; we express this invariant using the following class `Wf`:

```
Class Wf (m : t) := {
  is_wf : ∀t, mem m t = true → m t === canon m t
}.
Instance Wf_empty : Wf empty.
Instance Wf_add `{Wf m} (a : term) : Wf (add m a).
Instance Wf_merge `{Wf m} (p P : R) : Wf (merge m p P).
```

The instances ensure that all the operations provided above preserve the invariant `Wf`. We then specify the operations `empty`, `add` and `merge` in terms of `find`. Some properties will only be true for a union-find which verifies `Wf`; for instance the fact that `add` does not change the representatives:

```
   Property empty_find : ∀t, empty t = make t.
   Property add_find : ∀m `{Wf m} a a',
     find (add m a) a' === find m a'.
   Property merge_find : ∀m p P t,
     (merge m p P) t = subst p P (m t).
```

9.2.2 Use

The Use module implements a structure of finite maps from semantic values to sets of terms, representing a mapping from semantic values to terms who "use" this value (see the description of Γ in Section 3.2.2 page 62). Again it uses our containers library:

```
Section WithTheory.
  Context `{Th : Theory}.

  Definition t := Map[R, set term].
  Definition empty : t := [].

  Definition find (m : t) (r : R) : set term :=
    match m[r] with
      | Some r ⇒ r
      | None ⇒ ∅
    end.
  Coercion find : t ↣ Funclass.
  ...
End WithTheory.
```

The structure is simply a finite map from R to finite sets of terms, and find returns the set of terms associated to a value if any, and the empty set otherwise. There are two operations to modify such a structure, one where two values p and P are merged, and the other to add a new term fa to a list of values la:

```
  Definition merge (m : t) (p P : R) : t :=
    match m[p] with
      | None ⇒ m
      | Some usedp ⇒
        let lP := leaves P in
        List.fold_left (fun m l ⇒
          insert l usedp (union usedp) m) lP (remove p m)
    end.

  Definition add (m : t) (fa : term) (la : list R) : t :=
    List.fold_left (fun m l ⇒ insert l {fa} (add fa) m) la m.
```

Notice how `merge m p P` adds the terms in `find m p` to every value in `leaves P`, which corresponds exactly to the update of Γ in the Congr rule in Fig. 3.3. Both `merge` and `add` use the `insert` function provided by containers: `insert x d f m` updates the value bound to `x` in `m` by applying `f`, or uses `d` if `x` is unbound. It improves on the basic find-then-update by only using a single traversal of the map.

We define two extra operations in `Use` which will be used later in the algorithm:

```
Definition using_all (m : t) (lvs : list R) : set term :=
  match lvs with
    | nil ⇒ ∅
    | x::ls ⇒
      List.fold_left (fun acc y ⇒ acc ∩ (m y)) ls (m x)
  end.

Definition terms_of (m : t) : set term :=
  fold (fun _ v acc ⇒ v ∪ acc) m ∅.
```

`using_all m lvs` returns the set of terms which are used by all values in `lvs` and is used to compute new congruence equations, while `terms_of m` returns all terms appearing in the bindings in the map, and is used to ensure the termination of the algorithm in Section 9.2.5.

Properties. Contrary to `Uf.t`, the objects of `Use.t` do not require a well-formedness invariant. We again make the definitions in this module opaque to ensure encapsulation, and prove several properties which are enough to reason about the structure. It turns out that this structure has little effect on the soundness of CC(X) (even though it could threaten its completeness), so strictly speaking almost no properties are required. We prove basic invariants nonetheless, because the structure is reusable:

```
Property empty_find : ∀r, empty r = ∅.
Property terms_of_iff : ∀m t, t ∈ terms_of m ↔ ∃r, t ∈ m r.
...
```

9.2.3 `Diff`

The last structure which we present is implemented in the `Diff` module and represents a set of disequalities. It does not depend on a theory but only on the fact that terms are an ordered type, and uses a dictionary from terms to set of terms to represent, for every term `t`, the set of terms which are known to be different from `t`:

```
Section AnyOrderedType.
  Context ‘{term_OT : OrderedType term}.
  Definition t := Map[term, set term].

  Definition empty : t := [].

  Definition neqs (d : t) (a : term) : set term :=
    match d[a] with
      | Some s ⇒ s
      | None ⇒ ∅
    end.
  Definition are_diff (d : t) (a b : term) : bool :=
    mem a (neqs d b).
  Coercion are_diff : t ↣ Funclass.
  ...
End AnyOrderedType.
```

The definition of `empty` is straightforward, and so is the function `neqs` which simply returns the set of terms different from a given term. A diff structure can be used as a boolean relation between terms using `are_diff` and the corresponding coercion: if `N` as type `t` and `a`, `b` are two terms, `N a b` is true if `a` and `b` are different in `N`. The final operation which we define allows one to add a new disequality to the structure:

```
  Definition separate (d : t) (a b : term) : t :=
    insert a {b} (add b) (insert b {a} (add a) d).
```

When adding a disequality `a` $\neq$ `b`, it simply adds each term to the other's bindings.

Invariants and properties. A diff structure is only well-formed if it is symmetric, and we describe this invariant as a class, along with instances that this invariant is preserved by `empty` and `separate`:

```
  Class Wf (d : t) := {
    is_wf : ∀a b, d a b = d b a
  }.
  Instance Wf_empty : Wf empty.
  Instance Wf_separate ‘{Wf d} (a b : term) : Wf (separate d a b).
```

We then prove the specifications of the various operations using `are_diff`, some of which will require a well-formedness invariant:

```
  Remark are_diff_sym : ∀‘{Wf d} a b, d a b = d b a.
  Property are_diff_empty : ∀a b, empty a b = false.
  Corollary separate_monotonic : ∀d a b x y,
    d x y = true → (separate d a b) x y = true.
  Corollary are_diff_separate_1 : ∀d a b,
    (separate d a b) a b = true.
  ...
```

9.2.4 Raw Implementation of CC(X)

Our congruence closure procedure can be seen as a blackbox which implements two types of operations: it must process equalities and disequalities which it receives in input, and it must answer queries about whether some equality or disequality is true or not. This is similar but not exactly the same as environments as defined in Section 8.1.1, because environments must process all kinds of literals while CC(X) only deals with (dis)equalities. We can summarize the interface of the procedure we describe in this section in the following way:

```
  Parameter t : Type.

  Parameter empty : t.
  Parameter assume : input → t → Exception t.
  Parameter query : input → t → bool.
```

where the type `input` is simply defined as:

```
Inductive input : Set :=
| Equation (a b : term)
| Disequation (a b : term).
```

We define the procedure as a functor parameterized by a `THEORY` module, *i.e.* a module which brings a `Theory` instance and its specifications (see Section 9.1 above).

```
Module RAWCCX (Import T : THEORY).
  Structure t : Type := mk_env {
    G : Use.t;
    D : Uf.t;
    N : Diff.t;
    F : list (term × term);
    I : list input
  }.
  Definition empty : t :=
    mk_env Use.empty Uf.empty Diff.empty nil nil.
  ...
```

A configuration of our algorithm is defined as a record with five fields, which for the most part correspond to CC(X) configurations as defined in Chapter 3. The first three fields correspond to Γ, Δ and N and use the modules of data structures presented above: `G`, the "use" structure, has type `Use.t`, `D` has type `Uf.t` and `N` has type `Diff.t`. The field `F` partly corresponds to the

Φ part of CC(X) configurations: it contains pending equalities which must be processed, but it does not contain queries; indeed, contrarily to CC(X) configurations, our procedure here will receive and process the queries one at a time. Finally, field `I` has no counterpart in CC(X) configurations and is needed here for the specifications: it stores the list of inputs which have been assumed in the environment already. Note that CC(X) configurations also contained the set of terms Θ which had been added, our configurations do not and we will use the union-find `D` and the function `Uf.mem` to know whether a term is new or not.

In order to define `assume` and `query`, we start by defining many auxiliary functions; they precisely correspond to the inference rules for CC(X) and can be read in parallel with Fig. 3.4 page 79.

Assuming disequalities. We start with assuming disequalities since it is the easiest task. It is performed by the following function `diff a b e` which assumes a disequality between `a` and `b` in configuration `e`:

```
Definition diff (a b : term) (e : t) : Exception t :=
  if D e a == D e b then Inconsistent
  else
    let N' := Diff.separate (N e) a b in
    Normal (mk_env (G e) (D e) N' (F e) (I e)).
```

When the two terms are equal in `D e`, it implements rule IncohDiff and returns `Inconsistent`; otherwise, it implements the Diff rule and uses `Diff.separate` to update the `N` field in `e`.

Assuming equalities. There are four rules which deal with adding equalities: Congr, IncohEq, Unsolv and Remove. In particular, `Congr` is the most complex rule and requires the computation of the new equations obtained by congruence. This function is quite tedious and follows the mathematical definition, we just show its prototype:

```
Definition find_congr_equations
  (D' : Uf.t) (G : Use.t) (F : list (term × term))
  (p : R) (touched : list term) : list (term × term) := ...
```

This function calculates the new congruence equations in `D'`, knowing that the value which was substituted is `p` and that the terms whose representative changed are in `touched`. It prepends the resulting equations to `F`. After applying a substitution in the union-find, we also need to check if it has become incoherent with the diff structure or not, this is done by the `incoherent` function:

```
Definition incoherent
  (D : Uf.t) (N : Diff.t) (touched : list term) : bool :=
lexists (fun t ⇒
  exists_ (fun u ⇒ D t == D u) (Diff.neqs N t)) touched.
```

For every term whose representative changed, this checks if there exists a term in its class which also belongs to its bindings in `N`. We can now write the `merge` function, which assumes an equality `a = b` in a configuration:

```
Definition merge (a b : term) (e : t) : Exception t :=
  let 'mk_env G D N F I := e in
  if N a b then Inconsistent
  else
    let ra := D a in let rb := D b in
    if ra == rb then Normal e
    else
      match solve ra rb with
        | Unsolvable ⇒ Inconsistent
        | Solved ⇒ Normal e
        | Subst p P ⇒
          let '(D', touched) := Uf.merge' D p P in
          if incoherent D' N touched then Inconsistent
          else
            let G' := Use.merge G p P in
            let F' := find_congr_equations D' G  F p touched in
            Normal (mk_env G' D' N F' I)
      end.
```

If the two terms `a` and `b` are known to be different, the function returns `Inconsistent` (rule IncohEq). Otherwise, it retrieves their representatives `ra` and `rb` in `D`. If these are equal or the solver returns `Solved`, the configuration is unchanged (rule Remove). If the equation is unsolvable, rule Unsolv is applied. Otherwise, the substitution is applied to `D` using `Uf.merge'`, and if the new union-find becomes incoherent, `Inconsistent` is returned (rule IncohEq again). Otherwise, the new equations and the new use structure are computed and the updated environment is returned.

Adding terms. In accordance with the rules' side conditions, all our previous functions were implicitely assuming that the terms they were passed as arguments had already been added to the environment. Adding new terms is the task of the Add rule in the inference system, and must be done in a bottom-top manner (*i.e.* subterms first). Also, adding terms may yield new equalities by congruence, and similarly to above, we define a function to compute these new equations:

```
Definition find_add_equations
  (G : Use.t) (D : Uf.t) (F : list (term × term))
  (fa : term) (lvs : list R) : list (term × term) := ...
```

It computes the new equations as described in rule ADD, using `Use.using_all` in particular, and prepends them to `F`. We can now write the function `add_term fa e` which recursively adds `fa` and all its subterms, as required, to the configuration `e`.

```
Nested Fixpoint add_term (fa : term) (e : t) : t :=
  if Uf.mem (D e) fa then e else
    let 'app f la := fa in
      match add_terms la e with
        | mk_env G D N F I ⇒
          let D' := Uf.add D fa in
          let lvs := lleaves D la in
          let G' := Use.add G fa lvs in
          let F' := find_add_equations G D F fa lvs in
            mk_env G' D' N F' I
      end
  with add_terms (la : list term) (e : t) : t :=
    match la with
      | nil ⇒ e
      | a::qa ⇒ add_term a (add_terms qa e)
    end.
```

where `lleaves D la` corresponds to what we wrote $\mathcal{L}_{\Delta}(\vec{a})$ in the ADD rule. It simply proceeds by adding subterms recursively, stops as soon as a term has already been added, updates the different fields and the new equations and returns the updated configuration.

Cleaning up pending equations. Before a query can be correctly addressed, the pending equations which were added before that query must be processed. Until now, we have only added new equations to the field `F` of the configuration, but we now explain how to process these equations. We would like to simply iterate the `merge` function on these equations, until `F` becomes empty. The issue with that is that `merge` can add new equations itself, and therefore we have to ensure that this terminates. Because this recursion is not structural, and because we don't want to calculate an extra integer bound as we did for the DPLL procedure in Chapter 6, we use yet another method for defining non-structural recursive functions. Forest *et al.* [BFPR06] provide a facility for defining a recursion with respect to a *well-founded relation* on some of the arguments. The function is introduced with the `Function` vernacular: it must be accompanied with a proof that the chosen relation is well-founded, and that it decreases on recursive calls.

```
Function clean_up (e : t) {wf t_lt e} : Exception t :=
  let 'mk_env G D N F I := e in
  match F with
```

```
    | nil ⇒ Normal e
    | (u,v)::F' ⇒
      match merge u v (mk_env G D N F' I) with
       | Normal e' ⇒ clean_up e'
       | Inconsistent ⇒ Inconsistent
      end
  end.
Proof.
  ...
Defined.
```

The function is called `clean_up` and applies `merge` repeatedly until there are no more equations or until the configuration becomes inconsistent. It is declared as a recursion by well-foundedness for a relation `t_lt` on configurations, which we explain below. The proof that the recursive call is performed on a smaller configuration for that relation has the following form:

```
Theorem merge_decreases :
  ∀e u v e', merge u v e = Normal e' →
    t_lt e' (mk_env (G e) (D e) (N e) ((u,v)::(F e)) (I e)).
```

and the proof that `t_lt` is well-founded:

```
Theorem t_lt_wf : well_founded t_lt.
```

where `well_founded` is part of the Coq standard library. Now there are two thing to consider at this point, which are essential for the efficiency of the procedure:

1. The proof `merge_decreases` must be provided at the definition of the function, therefore it is important that it does not rely on implicit invariants on the structure. Otherwise, this would compel us to add these invariants to the type `t` of configuration (similarly to how we added invariants to expandable literals using dependent types in Section 7.3.2) and we would not be able to separate these functions and some of their specifications, which would add a considerable overhead. We want to keep this procedure purely computational and therefore it is very important that whatever invariant required in `merge_decreases` be true *by construction.* We show how `t_lt` is defined below, and why it decreases by construction during `clean_up`.

2. The second point is that functions defined by well-founded recursion are actually normal structural inductions, albeit on a special inductive type `Acc` which represents the fact that an element `x` of type `A` is *accessible* for a relation `R`:

```
Inductive Acc {A} {R : A → A → Prop} (x: A) : Prop :=
| Acc_intro : (∀y:A, R y x → Acc y) → Acc x.
```

> Accessibility represents the absence of infinite descending chains for relation `R`, in other words, any element `y` such that `R y x` holds is also accessible and is structurally smaller than `x`. Well-founded recursions are therefore just inductions on a proof of accessibility, and `well_founded R` is actually just defined as $\forall a$:`A, Acc` a. Therefore, for the computation to be effective, the proof of well-foundedness of the relation, in our case `t_lt`, must be reduced[3] until a constructor `Acc_intro` appears, *i.e.* until it is in head normal form. This computation of proofs can be costly as well, and therefore we use another trick[4] with consists in "guarding" the proof of well-foundedness `t_lt_wf` with a large number of `Acc_intro` so that the proof is never reduced in practice:

```
Fixpoint guard (n : nat) (wfR : well_founded t_lt)
  {struct n} : well_founded t_lt :=
  match n with
    | 0 ⇒ wfR
    | S n' ⇒ fun x ⇒
      Acc_intro x (fun y _ ⇒ guard n' (guard n' wfR) y)
  end.
Definition guarded_wf_lt := guard 100 t_lt_wf.
```

> This definition takes a proof of well-foundedness and guards it with 2^{100} constructors, which will be unveiled in a lazy manner during the computation. We then use `guarded_wf_lt` to define `clean_up` instead of `t_lt_wf`.

We finish this presentation of `clean_up` by detailing the relation `t_lt`. Since `clean_up` only applies `merge` repeatedly, the number of terms added to the union-find remains constant throughout the function. Therefore, one interesting measure is the number of different equivalent classes in the union-find: it strictly decreases as soon as the solver in `merge` returns a substitution. In other cases, the number of classes is unchanged, but no equation is added to `F`, therefore the number of pending equations strictly decreases in those cases. Altogether, we define `t_lt` as the lexicographic product of the number of classes and the number of pending equations and we are able to prove the `t_lt_wf` and `merge_decreases` lemmas.

Assumptions and queries. We can now define the top-level functions `assume` and `query` of our congruence closure procedure.

[3] Hence the fact that our proof following the definition ends with `Defined` instead of `Qed`.

[4] This method was introduced on the Coq-Club mailing list by B. Barras and G. Gonthier as a way to use well-founded recursion even if the well-foundedness proof is not constructive.

```
Definition assume (i : input) (e : t) :=
  match i with
    | Equation a b ⇒
      match merge a b (add_terms (a::b::nil) e) with
        | Normal (mk_env G' D' N' F' I') ⇒
          clean_up (mk_env G' D' N' F' (i::I'))
        | Inconsistent ⇒ Inconsistent
      end
    | Disequation a b ⇒
      match clean_up (add_terms (a::b::nil) e) with
        | Normal e' ⇒
          match diff a b e' with
            | Normal (mk_env G' D' N' F' I') ⇒
              Normal (mk_env G' D' N' F' (i::I'))
            | Inconsistent ⇒ Inconsistent
          end
        | Inconsistent ⇒ Inconsistent
      end
  end.
```

For both equalities and disequalities, `assume` starts by adding the terms of the input; it then dispatches the result either to `merge` or `diff` depending on the nature of the input. Note the required use of `clean_up` after `merge` and before `diff`.

The `query` function uses two auxiliary functions `are_equal` and `are_diff` which respectively follow the rules QUERY and QUERYDIFF:

```
Definition are_equal (a b : term) (e : t) : bool :=
  match clean_up (add_term b (add_term a e)) with
    | Normal e' ⇒ D e' a == D e' b
    | Inconsistent ⇒ true
  end.
Definition are_diff (a b : term) (e : t) : bool :=
  match assume (Equation a b) e with
    | Normal e' ⇒ false
    | Inconsistent ⇒ true
  end.
Definition query (q : input) (e : t) : bool :=
  match q with
    | Equation a b ⇒ are_equal a b e
    | Disequation a b ⇒ are_diff a b e
  end.
```

Note that before testing the status of an equation between terms, the terms must be added and the possible pending equations processed with `clean_up`. Note also how `assume` is used to resolve a disequality query.

9.2.5 Designing Invariants and Proofs

In order to establish the necessary proofs on the functions that we described in the previous section, we first define an invariant which must hold on all configurations during the proof search. Suppose we have a configuration `mk_env G D N F I`, the following six conditions must be verified:

1. `D` must be well-formed, *i.e.* `Uf.Wf D` must hold (see Section 9.2.1).

2. `N` must be well-formed, *i.e.* `Diff.Wf N` must hold (see Section 9.2.3).

3. The representatives associated to terms in the union-find `D` must be justified by the inputs, or more precisely by the equations in the inputs. To express this, we write a function:

   ```
   eqns_of : list input → list (term × term)
   ```

   which filters the equations in a list of inputs. The property we want to express corresponds to Proposition 3.3.2 on page 68 and states that the representative of a term `t` is obtained by applying `iter` to all equations already merged. We call this property `coincides` and define it as follows:

   ```
   Inductive coincides (D : Uf.t) (I : list input) : Prop :=
   | mk_coincides : ∀eqns,
     (∀p, In p (eqns_of I) → In p eqns) →  (* i *)
     (∀M, M ⊨ (eqns_of I) → M ⊨ eqns) →  (* ii *)
     (match iter eqns with                (* iii *)
        | Some f ⇒ ∀t, D t === f (make t)
        | None ⇒ False
      end) →
     coincides D I.
   ```

   When `coincides D I` holds, there exists a list of equations `eqns` which (i) contain at least the equations in the inputs `I`, (ii) are semantically justified by these input equations, (iii) verify Proposition 3.3.2. Note that `eqns` plays the role of the set of equations merged O in Proposition 3.3.2, it exists and is used in the proof, but doesn't need to be computed by the procedure.

4. The pending equations in `F` must be consequences of the inputs, which corresponds to the second part of Theorem 3.3.5. We call this property `justify` and define it as follows:

   ```
   Inductive justify (D : Uf.t) : list (term × term) → Prop :=
   | justify_nil : justify D nil
   | justify_cons : ∀f lu lv F,
   ```

```
  congruent D lu lv = true → justify D F →
  justify D ((app f lu, app f lv)::F).
```

When `justify D F` holds, it simply expresses that each equation in `F` must be of the form $f(\vec{t}) = f(\vec{u})$ where $\vec{t}$ and $\vec{u}$ are congruent in `D`.

5. The differences stored in `N` must correspond exactly to the disequalities in the inputs:

```
Definition Ncoincides (N : Diff.t) (I : list input) :=
  ∀a b, a ∈ Diff.neqs N b ↔
  (In (Disequation a b) I ∨ In (Disequation b a) I).
```

6. The diff structure `N` must not be incoherent with the union-find `D`, which we write `coherent D N`:

```
Definition coherent (D : Uf.t) (N : Diff.t) : Prop :=
  ∀a b, N a b = true → (Uf.mem D b = true ∧ D a =/= D b).
```

Note that `coherent D N` requires an extra property, which is that all terms that appear in `N` must have been added to `D`. This justifies that our algorithm only checks (in)coherence for the terms which have been touched in the union-find (see definitions of `merge` and `incoherent` above).

We can then define the class of well-formed configurations:

```
Class Wf (e : t) := {
  Dwf :> Uf.Wf (D e);
  Nwf :> Diff.Wf (N e);
  Dcorrect : coincides (D e) (I e);
  Fcorrect : justify (D e) (F e);
  Ncorrect : Ncoincides (N e) (I e);
  coherence : coherent (D e) (N e)
}.
```

which simply packs together all the properties described above. We then proceed to prove all sorts of properties on the various functions of the procedure, including the fact that the functions which modify a configuration preserve well-formedness under certain conditions. We use a very systematic and generic way to write the specifications of the different functions: we write a logical view for each function, *i.e.* a logical relation which contains the graph of the function and is precise enough to describe all the properties we need on the function. For instance, the view associated to `clean_up` is the following:

```
Inductive clean_up_spec (e : t) : Exception t → Prop :=
| clean_up_Inconsistent :
  (Wf e → ∀M, M ⊨ (I e) → False) →
  clean_up_spec e Inconsistent
| clean_up_Normal : ∀G0 D0
  (Hclean_wf : Wf e → Wf (mk_env G0 D0 (N e) nil (I e)))
  (Hclean_uf : ∀a b, D e a === D e b → D0 a === D0 b)
  (Hclean_range : ∀t, Uf.mem (D e) t → Uf.mem D0 t)
  (Hclean_congr : ∀a b, In (a, b) (F e) → D0 a === D0 b),
  clean_up_spec e (Normal (mk_env G0 D0 (N e) nil (I e))).
Theorem clean_up_dec : ∀e, clean_up_spec e (clean_up e).
Proof. .... Qed.
```

In the `Inconsistent` case, we require that the inputs `I e` are unsatisfiable. In the `Normal` case, the return type of the constructor shows that `clean_up` only modifies the `G` and `D` fields, and clears the `F` field. Other properties are named for clarity; for instance, `Hclean_wf` expresses the preservation of well-formedness, `Hclean_congr` the fact that all formerly pending equations are now merged in the union-find, etc. Subsequent reasoning on `clean_up` can then be done by simply eliminating `clean_up_dec`[5], and all relevant properties are automatically added. We believe that when writing an algorithm which involves many functions and sum types (like `Exception` or `bool`), this approach has many benefits and scales well in comparison to writing many ad-hoc lemmas for each function, in particular it avoids many practical problems.

9.3 A CC(X) Environment for DPLL

The algorithm which we have presented in Section 9.2 does not qualify as an environment for DPLL, as we defined in Chapter 8. Indeed, it only deals with equalities or disequalities but not propositional atoms. Moreover, the properties that we were able to established on the `assume` and `query` functions in the previous section depended on the fact that the configuration was well-formed. In this section, we start by deriving the functor from the previous section in order to hide the well-formedness constraints, and we then extend it into an environment by taking propositional atoms into account.

9.3.1 `CCX` with Invariants

We use the same method which we presented on expandable literals in Chapter 7: we use dependent pairs and the functor `RAWCCX` described in the pre-

[5] It is even possible to make `clean_up` opaque once `clean_up_dec` has been established, since there is no need to use the definition if the specification is sufficient for one's purpose. Our experience is that it is good practice to always make these functions opaque when possible, since this can speed up typechecking of subsequent proofs significantly.

vious section to implement a CCX procedure which works on well-formed configurations:

```
Module CCX (Import T : THEORY).
  Module RAW := RAWCCX T.
  Definition t := {e : RAW.t | RAW.Wf e}.
  ...
End CCX.
```

The definition of the various operations relies on the proofs of preservation established in RAWCCX, and there is nothing complex about it. Similarly, the specifications are lifted from the proofs established in RAWCCX. In the end, this functor has the signature CCX_SIG, which is given for reference in Fig. 9.1.

```
Module Type CCX_SIG.
  Parameter t : Type.

  Parameter empty : t.
  Parameter assume : input → t → Exception t.
  Parameter query : input → t → bool.
  Parameter assumed : t → list input.

  Module Specs.
    Axiom assumed_empty : assumed empty = nil.
    Axiom assumed_assume :
      ∀c i c′, assume i c = Normal c′ → assumed c′ = i::(assumed c).
    Axiom assumed_inconsistent :
      ∀c i, assume i c = Inconsistent →
        query (match i with
                 | Equation a b ⇒ Disequation a b
                 | Disequation a b ⇒ Equation a b
               end) c = true.

    Axiom query_true : ∀c i, query i c = true →
      ∀M, M ⊨ assumed c → M (input_to_lit i).
    Axiom query_assumed :
      ∀c i, In i (assumed c) → query i c = true.
  End Specs.
End CCX_SIG.
```

Figure 9.1: The signature of the CCX blackbox.

9.3.2 A CCX-based Environment

We are now interested in developing an environment relying on our CCX implementation, in order to use in our DPLL procedure. In Section 8.1.1, we presented the signature ENV_INTERFACE that such an environment must verify. It is actually parameterized by a module of formulae, *i.e.* of signature CNF. The type of literals which we are interested in has been described in Section 8.3 and is provided by the module LITINDEX which embeds both propositional atoms and (dis)equalities. We now suppose we have a module CNFLAZY of signature CNF which represents formulae whose literals are expandable literals based on LITINDEX. We want to build an environment for these formulae, in other words a module of signature ENV_INTERFACE CNFLAZY.

```
Module ENVLAZY (CC : CCX_SIG) <: ENV_INTERFACE CNFLAZY.
  Import CNFLAZY.
  Record t := mk_t {
    (* - uninterpreted atoms which have been assumed *)
    env : set L.t;
    (* - the CC environment *)
    cc : CC.t
  }.

  Definition empty := mk_t ∅ CC.empty.
  ...
End ENVLAZY.
```

Our functor ENVLAZY is parameterized by a CC(X) procedure using the signature CCX_SIG seen above (in particular it does not know anything about the notion of theory, this is internal to the CC module parameter). The type of environments is a record which contains a CCX configuration CC.t on one side, and a set of propositional atoms on the other side. The definitions of the operations assume and query is fairly straightforward: depending on whether a literal is an atom or an equality, it is added to the env part of the environment or passed to the cc configuration using CC.assume; similarly, the query of an atom is just a lookup in the set env, while the query of an equality is performed using CC.query. Using the specification in the CCX_SIG signature, we are able to prove all the required properties and give our functor the expected interface.

9.4 Results

9.4.1 Example

As an example, let us consider the module EmptyTheory for the theory of equality on uninterpreted functions which we described at the beginning of this chapter. We can instantiate our CCX functor on this module to get a procedure for simple congruence closure:

```
Module CCE := CCX EmptyTheory.
```

We can then use this module to instantiate the `ENVLAZY` functor:

```
Module E := ENVLAZY CCE.
```

which can in turn be used to instantiate our SAT functor described in Section 8.1.3:

```
Module DPLLE := SAT CNFLAZY E.
```

The resulting module is a DPLL procedure for expandable literals with equalities and an environment that performs congruence closure: in other words, it is a decision procedure for satisfiability modulo equality. Since `DPLLE` has signature `DPLL CNFLAZY E`, it can be passed to a functor of signature `LoadTactic` similar to the one presented in Section 6.3 which contains the reflection lemma and generates the `unsat` tactic for formulae in `CNFLAZY`:

```
Module TacE := LoadTactic DPLLE.
Ltac cc := TacE.unsat.
```

We can now prove a goal by reflection using the tactic `cc`:

```
1 subgoal

A : Type
f : A → A
x : A
H : f (f (f (f (f x)))) = x
H0 : f (f (f x)) = x
H1 : f x ≠ x
============================
 False
```

This goal is valid because it is well-known in mathematics that if x is a fixpoint of the iterates f^n and f^m of some function f, it is also a fixpoint of $f^{|n-m|}$, and therefore of $f^{gcd(m,n)}$. The goal is discharged automatically in about 10ms by the tactic:

```
 cc.
```

```
 Proof completed.
```

The reader may notice that this example does not really use propositional reasoning, here is a similar goal which involves some propositional reasoning, and succeeds in a hundredth of a second as well:

```
1 subgoal

A : Type
f : A → A
x : A
y : A
H : f (f x) = y ∧ f (f (f y)) = x
H0 : f y = x ∨ f x = x
H1 : f x ≠ x
============================
 False
```

```
cc.
```

```
 Proof completed.
```

9.4.2 Conclusion

To conclude this chapter, we have presented a Coq implementation of the CC(X) combination method which we presented earlier in Chapter 3. We have formalized the notion of solvable theories and implemented the procedure as Coq functions. We then showed how to include this procedure in a DPLL environment which handles propositional atoms other than equalities. So doing, and using functors presented in previous chapters, we can derive a tactic which decides the satisfiability of a formula modulo the theory of equality on uninterpreted symbols. This tactic is similar to a combination of the tactics `tauto/intuition` and `congruence`; however, it is fully reflexive, implements the same algorithm as our SMT solver Alt-Ergo and is entirely programmed and proved in Coq, with the exception of the reification mechanism which is performed in OCaml for the sake of efficiency.

CHAPTER 10

A Theory of Linear Arithmetic

> Mentir à sa façon à soi, c'est presque mieux que de dire la vérité à la façon des autres.
>
> Fiodor M. Dostoievski, *Crime et Châtiment*

Contents

In Section 3.2.3, we presented the theory of linear rational arithmetic $\mathcal{A}$ as an interesting example of a solvable theory which we could plug into our CC(X) system. We will now implement this theory for the CC(X) implementation described in Chapter 9, and see how it can be used to decide satisfiability modulo linear integer arithmetic in the Coq proof assistant. In Section 10.1, we present an implementation of polynomials with rational coefficients at the base of our theory, before describing the theory itself in Section 10.2. We conclude in Section 10.3 by giving examples of use of the tactic obtained with this theory.

10.1 Rational Polynomials

As explained in Section 3.2.3, a convenient representation of terms for the theory of arithmetic is as a sum of monomials with rational coefficients. Because we are only dealing with linear arithmetic, we are only interested in rational polynomials of degree one, *i.e.* the monomials are actually reduced to simple terms. The theory of arithmetic is actually independent of the actual representation of polynomials and we now present a Coq implementation of rational polynomials.

10.1.1 Raw Polynomials

We define rational polynomials in a general manner, as parameterized by an ordered type `vars` representing variables. We require that `vars` be an ordered type in order to be able to use an efficient data structure to represent polynomials: we use a finite map from our containers library to map variables to rational coefficients. This leads to the following `poly` definition:

```
Section WithVars.
  Variable vars : Type.
  Context `{vars_OT : OrderedType vars}.

  Definition poly := Q × Map[vars, Q].

  Definition P0 := (0, []).
  Definition P1 := (1, []).
  Definition embed (v : vars) : poly := (0, [][v ← 1]).
  ...
End WithVars.
```

A rational polynom is a pair of a constant rational coefficient and a finite mapping from variables to rationals. We also give the definitions of the constant zero polynom `P0` and the monom `embed v` for any variable `v`. As other basic examples, we can define various basic constructors for polynomials:

```
  Definition mult (c : Q) (p : poly) : poly :=
    if (c == 0) then P0
    else (c × fst p, map (fun q ⇒ c × q) (snd p)).
  Definition add_const (c : Q) (p : poly) : poly :=
    (fst p + c, snd p).
```

to multiply a polynomial by a constant or add a constant to a polynomial. A more interesting operation is `addk` P_1 k P_2 which returns the polynom $P_1 + kP_2$:

```
  Definition addk_m (p1 : Map[vars,Q])
    (k : Q) (p2 : Map[vars,Q]) : Map[vars,Q]:=
    map2 (fun oc1 oc2 ⇒
      match oc1, oc2 with
        | None, None ⇒ None
        | Some q1, None ⇒ Some q1
        | None, Some q2 ⇒ Some (k × q2)
        | Some q1, Some q2 ⇒
          let q := q1 + k × q2 in
          if q == 0 then None else Some q
      end) p1 p2.
  Definition addk (p1 : poly) (k : Q) (p2 : poly) : poly :=
    if k==0 then p1
    else (fst p1 + k × fst p2, addk_m (snd p1) k (snd p2)).
```

The auxiliary function addk_m uses the map2 function provided by containers: map2 f m1 m2 merges two maps m1 and m2 by iterating f over all keys which appear in either map. The function f expects two options, representing the values bound to some key in each map or None if no such binding is present, and returns an option: the value bound to that key in the resulting map, or None if no binding should be added for that key. For instance, in addk_m m p1 p2, if some variable is bound to q1 in p1 and q2 in p2, we compute q := q1 + k * q2 and add it to the resulting map if it is non-zero. addk simply uses addk_m to sum the maps of monomials, and calculates the constant coefficient as well. Using the functions seen so far, we can define the usual operations:

```
  Definition add (p1 p2 : poly) : poly := addk p1 1 p2.
  Definition sub (p1 p2 : poly) : poly := addk p1 (-1) p2.
  Definition div (c : Q) (p : poly) : poly := mult (Qinv c) p.
```

In order to write the specifications of these rational polynoms, we define the function which returns the coefficient associated to a variable in a polynom:

```
  Definition coef_of (p : poly) (v : option vars) : Q :=
    match v with
      | Some v ⇒
        match (snd p)[v] with Some q ⇒ q | None ⇒ 0 end
      | None ⇒ fst p
    end.
  Coercion coef_of : poly ↣ Funclass.
```

coef_of (Some v) returns the coefficient associated to variable v in the polynom, while coef_of None returns its constant coefficient. We can see polynoms as functions from option vars to rationals, and this gives us a natural way to express the equality of polynoms:

```
  Definition equiv (p1 p2 : poly) := ∀t, p1 t === p2 t.
```

This relation, which binds two polynoms which have the same coefficients, is weaker than Leibniz equality but we can prove that it is an equivalence relation on polynoms, and more importantly we can prove that all operations on polynoms (addition, multiplication, etc) are morphisms for this operation. For instance, the following declaration:

```
Instance add_equiv : Morphism (equiv ⟹ equiv ⟹ equiv) add.
Proof. ... Qed.
```

proves that `add` is a morphism for `equiv`. The morphism and rewriting mechanism (whose implementation was revamped by M. Sozeau and is described in [Soz09]) then typically allows one to rewrite `add p1 p3` into `add p2 p3` using the "equality" `equiv p1 p2`.

We can write the specifications of the different operations using the characterization of polynomials as functions from variables to rationals:

```
Property P0_co : ∀t, P0 t === 0.
Property embed_co :
  ∀v t, embed v t === if t == Some v then 1 else 0.
Property mult_co : ∀k a t, mult k a t === k × a t.
Property addk_co : ∀a k b t, addk a k b t === a t + k × b t.
...
```

10.1.2 Polynoms as `OrderedType`

In order to use the type `poly` as the type of semantic values in our theory, we need to give an instance of `OrderedType poly`. Moreover, we do not want any instance of `OrderedType`, we want an instance where the equality relation is `equiv`. Indeed, if our implementation of the theory of arithmetic is going to verify the specifications of `TheorySpecs` described in Section 9.1, `solve p q` will have to return `Solved` if and only if `p` and `q` are equal for the provided `OrderedType` instance (*i.e.* `p === q`). On the other hand, if `equiv p q` holds, the two polynoms are strictly equivalent and therefore `solve p q` will return `Solved`. In other words, the equality for which the `OrderedType` instance is provided must coincide with the `equiv` relation that we introduced above.

It turns out that our containers library provides a generic instance of `OrderedType Map[key,elt]` as long as `elt` is itself an ordered type. Since both `vars` and `Q` are ordered types, an instance can be automatically inferred by the system for `Map[vars, Q]`, and therefore for `Q * Map[vars, Q]`, *i.e.* `poly`. This instance is not for Leibniz equality, and it is not for `equiv` either: the maps are considered equal if and only if both have the same keys with equal bindings. This is a stronger equality than `equiv` because two equivalent polynoms can differ by monomials with null coefficients: any binding of a variable to 0 in the map has no effect on the relation `equiv` but has effect on the equality inferred by the typeclass mechanism. However, for polynoms without null coefficients, the two notions of equalities coincide. Therefore, in order to use the available instance with `equiv`, we will equip the type of raw polynomials `poly` with the invariant that all variables in the map have non-zero coefficients. We define the class of polynomials which verify these invariants:

```
Class Wf (p : poly) := {
  Wf_p : ∀v q, MapsTo v q (snd p) → q =/= 0
}.
```

We took care when implementing the operations on polynomials above to avoid adding monomials with null coefficients, and as a consequence we are able to prove that the basic constructors and operations preserve this invariant:

```
Instance Wf_P0 : Wf P0 := ...
Instance Wf_embed (v : vars) : Wf (embed v) := ...
Instance Wf_mult (c : Q) `{Wf p} : Wf (mult c p) := ...
Instance Wf_addk `{Wf p1} k `{Wf p2} : Wf (addk p1 k p2) := ...
...
```

We now proceed as we did several times already, for expandable literals in Section 7.3.2 and for the CC(X) configurations in Section 9.3.2: we pack all the development above in a module `RAW` and define the type of polynomials as a dependent pair of a raw polynomial and the `Wf` invariant:

```
Definition poly := {p : RAW.poly | RAW.Wf p}.
```

and we lift all operations on `RAW.poly` which preserve the invariant to this type `poly`. We are now able to use the instance of `OrderedType` automatically inferred to compare polynomials and prove that this is an instance for `equiv`, we call this instance `poly_OT`.

To conclude the implementation of polynomials, we add extra operations which are useful for the implementation of the theory in the next section. We add these operations now because we want to make the definitions of polynomials opaque in order to ensure both encapsulation and the fact that the theory implementation is independent of the actual representation of polynomials.

```
Definition is_const (p : poly) : option Q := ...
Definition leaves (p : poly) : list poly := ...
Definition extract (p : poly) : option vars := ...
Definition choose (p : poly) : option (vars × Q) := ...
```

`is_const` checks if a polynomial is constant, in which case it returns its constant coefficient or not; `leaves` returns the list of variables in the polynom; `extract` is the reverse operation of `embed`: if its argument is simply a polynomial of the form `embed t`, it returns `Some t`, and `None` otherwise; finally, `choose p` returns an unspecified monomial in `p` if possible.

10.2 Theory of Integer Arithmetic

10.2.1 Implementation

We give the implementation of an instance of the class `Theory` for the theory of linear integer arithmetic. This implementation is based on the rational polynomials we just described, and we show below in Section 10.2.2 that this implementation verifies the expected specifications.

```
Definition R := poly (vars := term).
Definition R0 : R := P0.
Definition R1 : R := P1.
```

After the definition of the type of semantic values `R`, we define the `make` function used to convert a term into a semantic value. We show the full definition in Fig. 10.1 for reference. It uses an auxiliary function `mk_term` such that `m_term coef p t` constructs the polynomial for the term `t` multiplied by coefficient `coef` and adds it to polynomial `p`. It recursively builds a polynomial by analyzing the head symbol of its argument. When the symbol is uninterpreted (or when the symbol is an arithmetic symbol but it is not used with the correct arity), we simply add the term `t` as a monomial with coefficient `coef`. Most cases are self-explanatory, for instance when `t` has the form `app (Op Plus) (t1::t2::nil)`, we construct and add the polynomials for `t1` and `t2` by calling `mk_term coef (mk_term coef p t1) t2`. The most interesting case is the case of the multiplication symbol, which is particular since it is only partially interpreted: since we are only dealing with linear arithmetic, we only interpret multiplications by constants. Therefore, the two arguments `t1` and `t2` are recursively transformed in polynomials and we test whether the results are constant or not in order to adequately build the resulting polynomial.

We define the special value `one`, arbitrarily, as `R1`, and implementing the `leaves` function is simply done by retrieving the variables in the polynomial, and returning `one` if there is none:[1]

```
  Definition one : R := R1.
  Definition leaves (p : R) :=
    match leaves p with
      | nil ⇒ one::nil
      | l ⇒ l
    end.
```

The last two functions required by a theory are the substitution application and the solver. Using mathematical notations, the substitution of t_j by P in $\Sigma_i q_i t_i$ is simply $q_j P + \Sigma_{i \neq j} q_i t_i$, the implementation of `subst` follows this:

[1] Remember that CC(X) requires that the leaves of a semantic value always be non-empty.

```
Fixpoint mk_term (coef : Q) (p : R) (t : term) : R :=
  match t with
    | app (Unint _ _) _ ⇒
      add_monome t coef p
    | app (Cst z) nil ⇒
      add_const (coef × z) p
    | app (Op Plus) (cons t1 (cons t2 nil)) ⇒
      mk_term coef (mk_term coef p t1) t2
    | app (Op Mult) (cons t1 (cons t2 nil)) ⇒
      let p1 := mk_term 1 R0 t1 in
      let p2 := mk_term 1 R0 t2 in
        match is_const p1, is_const p2 with
          | Some c1, Some c2 ⇒
            add_const (coef × (c1 × c2)) p
          | Some c1, None ⇒
            addk p  (c1 × coef) p2
          | None, Some c2 ⇒
            addk p (c2 × coef) p1
          | None, None ⇒
            add_monome t coef p
        end
    | app (Op Minus) (cons t1 (cons t2 nil)) ⇒
      mk_term (Qopp coef) (mk_term coef p t1) t2
    | app (Op Opp) (cons t1 nil) ⇒
      mk_term (Qopp coef) p t1
    | _ ⇒ add_monome t coef p
  end.
Definition make (t : term) : R := mk_term 1 R0 t.
```

Figure 10.1: Construction of polynomials from terms

```
Definition subst (p P r : R) : R :=
  match extract p with
    | Some t ⇒ addk (cancel t r) (r (Some t)) P
    | None ⇒ r
  end.
```

In particular `subst p P r` only changes `r` if the pivot `p` is of the form `embed t` for some term `t`. Now, in order to solve an equation between two polynomials `t` and `u`, we proceed by case analysis on the form of `t - u`:

```
Definition solve (t u : R) : Solution R :=
  let diff := sub t u in
  match choose diff with
    | None ⇒ (* constant *)
      if diff None == 0 then Solved else Unsolvable
    | Some (t, k) ⇒
```

```
        let P := mult (Qinv (Qopp k)) (cancel t diff) in
          Subst (embed t) P
    end.
```

If `t - u` is the constant polynom `0`, the equation is `Solved`, and if it is any non-zero constant, we return `Unsolvable`. Otherwise, it has the form $\Sigma_{i=0}^{n} q_i t_i$ and we return the substitution $t_0 \mapsto -\frac{1}{q_0}\Sigma_{i=1}^{n} q_i t_i$. Notice that we use the fact that our polynomials do not contain monomials with null coefficient as this guarantees us that q_0 is not zero. This allows us to pick any monomial to isolate and perform the Gauss elimination.

With all the above definitions implemented, we finish by declaring the corresponding `Theory` instance:

```
Instance Arith_theory : Theory := {
  R := R;
  R_OT := poly_OT;
  make := make;
  one := one;
  leaves := leaves;
  subst := subst;
  solve := solve
}.
```

10.2.2 Specifications

Once the instance `Arith_theory` is defined, we are left with proving that this theory meets the required specifications described in the class `TheorySpecs` (see page 188). Some properties are straightforward to establish: notably, the fact that all operations are compatible with the equivalence relation on polynomials, or that the `leaves` of a semantic value are never empty. The others require considerable work: we first look at the specifications of the `solve` function, and later at the property `implyX_entails` which links theory reasoning and semantic entailment.

Specifications of `solve`. The solver which we have implemented above must verify the specifications described in the inductive `solve_specs` presented on page 187. The case for `Solved` does not raise any issue; neither does the case of a substitution `Subst p P`, where it is rather straightforward to establish the required properties by a simple analysis of the definitions of `solve`, `subst` and `leaves`. The only case which requires extra work is `Unsolvable`, where we need to establish that if `solve u v` is unsolvable, then any lists of equations `E` such that `implyX E u v` holds must be inconsistent (*i.e.* `iter E = None`). We prove this by noticing interesting invariants on `iter E`: we characterize all functions `f : R → R` for which there exists `E` such that `iter E = Some f`, and we can prove that such functions `f` have interesting properties. In particular, they are compatible with equivalence of polynomials, and are *linear* with respect to the operations on polynomials:

```
Section Iter.
  Variable E : list (term × term).
  Variable f : R → R.
  Variable Hf : iter E = Some f.
  ...
  Instance iter_m : Morphism (equiv ⟹ equiv) f.
  Property iter_R0 : f R0 === R0.
  Property iter_linear :
    ∀a k b, f (addk a k b) === addk (f a) k (f b).
  ...
  Corollary iter_sub : ∀a b, f (sub a b) === sub (f a) (f b).
  ...
End Iter.
```

This linearity property is very interesting since it allows to express the interaction between a function returned by `iter E` and all the various polynomial operations (*e.g.* substraction in `iter_sub` above). Using this, we can prove the fundamental property that `iter E` is idempotent:

```
Corollary iter_idem :
  ∀E f, iter E = Some f → ∀r, f (f r) === f r.
```

We can also establish the soundness of `solve` when the equation is unsolvable: suppose `implyX E u v` and `solve u v` is unsolvable, we suppose `iter E` is equal to some function `f` and establish a contradiction. By definition, we know that `f u === f v`, which by linearity means that `f (sub u v) === R0`. Now using the definition of `solve` and the fact that it returned `Unsolvable`, we know that `sub u v` is equal to some constant, non-zero, polynomial `q`. By linearity again, this means that `f (sub u v)` is equal to the constant polynom `q`, which cannot be `R0` since `q` is non-zero. This way, we can prove the following theorem of soundness for `solve`:

```
Theorem solve_dec : ∀u v, solve_specs u v (solve u v).
```

Specification of `make`. Let us recall the property which must be proved if we want `implyX` to be semantically correct:

```
  implyX_entails :
    ∀E u v, implyX E (make u) (make v) → E ⊢ u = v;
```

This is by far the most complex proof we have to establish on our theory, because it requires linking the manipulations of polynomials performed by our theory to the actual semantic notion of equality and entailment on terms.

First of all, note that when the list of equations `E` is empty, this property reduces to:

```
∀(u v : term), make u === make v → ∀M, M ⊨ u = v.
```

which can be read as the semantic correctness of make. We start by proving this lemma on make, and in order to do so, we need to justify the manipulations performed in make: intuitively, the construction of the polynomials is justified by properties such as the associativity and commutativity of the symbol Op Plus on terms, its distributivity over Op Mult, etc. In other words, we can justify the representation of terms as polynomials by using the fact that terms and arithmetic symbols form a *commutative ring structure*. More precisely, they do not form a ring structure for Leibniz equality (the terms app (Op Plus) (u::v::nil) and app (Op Plus) (v::u::nil) are not equal for instance), but for semantic equality in a certain model M. This means that every model M defines a ring structure over terms.

In order to avoid performing a variety of ring-specific reasoning and manipulations manually, we can use the ring reflexive tactic available in Coq (see [GM05]). Indeed, ring not only works on some predefined ring structures like booleans, integers or rationals, but it allows one to declare a new ring structure. Therefore, let us assume a fixed model M and define a new adequate ring structure. For the sake of clarity, we also introduce some notations for arithmetic operations on terms: u [+] v will stand for app (Op Plus) (u::v::nil), u [-] v for app (Op Minus) (u::v::nil), etc. For any integer z, we also write [z] for the term app (Cst z) nil which corresponds to this integer. We then prove that the type term with these operations form a commutative ring structure with additive and multiplicative neutrals being respectively [0] and [1]:

```
Theorem models_ring : Ring_theory.ring_theory
  term [0] [1] [+] [×] [-] [opp] (models_eq M).
Add Ring models_eq_Ring : models_ring (abstract).
```

The last parameter to Ring_theory.ring_theory is the equivalence relation for which the structure is a ring, and we chose semantic equality in the model M[2]. We then registered this *ad hoc* ring structure to the system with Add Ring. The tactic ring can then be used to discharge semantic equalities which are consequences of ring roperties:

```
Goal ∀t t′, M ⊨ t [+] t′ [-] [0] = t′ [+] t.
Proof. intros; ring. Qed.
```

Note that proving such a lemma directly would be very tedious, because we would have to delve into the definition of models_eq, *i.e.* into the interpretation of terms as presented in Chapter 8. As a side remark, it is not surprising that proving our theory for linear integer arithmetic involves a

[2] Remember from Chapter 8 page 8.3 that M ⊨ u = v is just a shortcut notation for models_eq M u v.

large part of common reasoning with the implementation of the ring tactic. However, we cannot use ring's implementation directly in our reflexive procedure because its concrete objects are less refined than our tactic's concrete objects, *i.e.* terms. This prevents us from directly reusing their development. Fortunately, we are able to use the fact that ring allows the declaration of customized ring structures in order to avoid reimplementing our own formalization of the theory of rings.

Now that we have ensured that ring reasoning about semantic equality will be automated, we focus again on proving that if make u === make v for some terms u and v, then M ⊨ u = v. The intuition behind this is that if u and v yield the same polynomial, then there exists some "canonical" term t which can be obtained from u and v using ring properties. To that purpose we define the "inverse" of make, *i.e.* a function term_of_R which takes a polynomial R and returns a term corresponding to this polynomial.

```
Definition term_of_R (r : R) : term :=
  [Qfloor (r None)] [+]
  fold (fun v qv acc ⇒ [Qfloor qv] [×] v [+] acc)
    (snd (π₁ r)) [0].
```

In essence, when applied to a polynomial $c + \Sigma_{i=0}^{n} q_i t_i$, term_of_R returns the term:

$$\lfloor c\rfloor[+]\lfloor q_0\rfloor[*]t_0[+]\ldots[+]\lfloor q_n\rfloor[*]t_n$$

where $\lfloor c\rfloor$ is the closest integer less or equal to rational c, and where the t_i are in increasing order. It is clear that the point of writing such a function is that when combined with make, it acts as a term canonizer for our theory of linear arithmetic. In particular, it constructs the same term for two equivalent polynoms. One difficulty, which is visible in the definition of term_of_R, is that our terms can only embed integer constants, whereas our polynomials have rational coefficients. Therefore we use Qfloor to convert rationals to integers in the definition, but many properties of term_of_R will only hold when applied to polynomials with integer coefficients. To that end, we define predicates isZ : Q -> Prop and isZpoly : R -> Prop:

```
Definition isZ (q : Q) : Prop := ...
Definition isZpoly (r : R) : Prop := ∀t, isZ (r t).
```

which identify rationals which are actually integers, and polynomials whose coefficients are all integers. We can prove the main properties of term_of_R:

```
Property term_of_R_embed : ∀t, M ⊨ term_of_R (embed t) = t.
Property term_of_R_addk : ∀p₁ k p₂,
  isZpoly p₁ → isZpoly p₂ → isZ k →
  M ⊨ term_of_R (addk p₁ k p₂) =
       term_of_R p₁ [+] [Qfloor k] [×] term_of_R p₂.
```

The first property specifies `term_of_R` on uninterpreted terms, whereas the second shows how it can be "distributed" over a linear combination of polynomials with integer coefficients. By induction on a structure of a term and definitions of `make` and `mk_term`, we can prove the fundamental result:

Property `make_correct` : $\forall t$, `M` $\models$ `term_of_R (make` t`)` = t.

whose meaning is that the combination of `term_of_R` and `make` canonize a term in one which is semantically equal. From this theorem and the transitivity of equality follows the correctness of `make`:

Corollary `make_entails` : $\forall t\ u$, `make` t `===` `make` u $\to$ `M` $\models$ t = u.

Specification of `implyX`. We prove `implyX_entails` by induction on the list of equations `E` on which `iter` is applied. The theorem `make_entails` which we just established represents the initialization step of this induction. In order to prove the induction step, we consider a set of equations `(a,b)::E` and suppose the property holds for `E`. If `iter E = None` the result is straightforward, but the interesting case arrives when `iter E` is some function `f`. In that case, by definition of `iter`, the result of `iter ((a,b)::E)` depends on the result of `solve (f (make a)) (f (make b))`. The main issue in this part of the proof is that our induction hypothesis has the form:

`IH` : $\forall u\ v$, f `(make` u`)` `===` f `(make` v`)` $\to$ `E` $\models$ u = v

and therefore only characterizes `f` on polynomials of the form `make t` for some term `t`, *i.e.* on polynomials with integer coefficients. In practice, we have polynomials like `f (make a)` and `f (make b)` which are not necessarily of that form because applying `f` yields rational coefficients in general. The main difficulty of the proof is then to try and construct adequate terms on which to apply the induction hypothesis. In that regard, the linearity and idempotency of `f` play an important role. Another fundamental property is the fact that for any polynomial `P`, there exists a "multiple" of `P` which has integer coefficients, hence the following theorem:

Theorem `find_multiple_poly` : $\forall$`(P : R)`,
  $\exists m$: `Z`, m `=/= 0` $\wedge$ `isZpoly (mult` m `P)` $\wedge$
    $\exists$`mP : term, make mP === mult` m `P`.

This theorem gives the existence of a term `mP` such that `make mP` has integer coefficients and is a multiple of `P`.

In order to give a better understanding of how we proceed in practice, we detail the induction step when `solve (f (make a)) (f (make b))` returns `Unsolvable` (the case of a substitution `Subst p P` would be somewhat similar, albeit much too complex to be detailed here). In this unsolvable case, we need to prove `(a,b)::E` $\vdash$ `u = v` for any terms `u` and `v`. By definition of `solve`, we know that there exists a constant polynomial `q` different from zero such that sub (f (make a)) (f (make b)) === q. Let m be an integer such that m * q is an integer c. We define the following terms u' and v':

```
u' := [m] [×] (a [-] b)
v' := [c]
```

and we want to apply the induction hypothesis to these terms. To that end, we need to prove that f (make u') === f (make v'). By applying properties of make and the linearity of f, we can prove that f (make u') is equal to mult m (sub (f (make a)) (f (make b))), hence equal to mult m q and to make [c]. We can thus apply the induction hypothesis, and we obtain that the semantic entailment E ⊢ u' = v' holds. Let us now consider the set of equations (a,b)::E, since it includes E it also entails u' = v'. But because it contains (a,b), we can replace a [-] b with [0] in u', and we finally obtain that (a,b)::E ⊢ [0] = [c]. This means that c is null, which contradicts our hypothesis that q was different from zero and ends the proof in that case. Proceeding similarly for the other cases, we prove the correctness of implyX:

```
Theorem implyX_entails : ∀E u v,
  implyX E (make u) (make v) → E ⊢ u = v.
```

Definition of ArithTheory. We finish this implementation by declaring the module of signature THEORY for our theory of arithmetic, using the above theorems in order to provide the instance for the specifications of the theory.

```
Module ArithTheory <: THEORY.
  Definition Th := Arith_theory.

  Instance ThSpecs : TheorySpecs Th := {
    solve_dec := solve_dec;
    implyX_entails := implyX_entails;
    ...
  }.
End ArithTheory.
```

10.3 Results

10.3.1 Example

Proceeding exactly like we did in Section 9.4.1, we can instantiate our CCX functor on the theory of linear integer arithmetic, obtain an environment for DPLL and *in fine* a tactic which we call ergo and which combines propositional reasoning and the theory of equality on uninterpreted symbols modulo linear integer arithmetic.

```
Module CCA := CCX ArithTheory.
Module E := ENVLAZY CCA.
Module DPLLA := SAT CNFLAZY E.
Module TacA := LoadTactic DPLLA.
Ltac ergo := TacA.unsat.
```

We can try a simple example of a goal which mixes arithmetic reasoning and equality:

```
1 subgoal

f : Z → Z
x : Z
y : Z
H : f (x - 1) - 1 = x + 1
H0 : f y + 1 = y - 1
H1 : y + 1 = x
============================
 False
```

```
 ergo.
```

```
 Proof completed.
```

The goal is discharged by the `ergo` tactic in 5 hundredth of a second. We can try a more complex example which requires some propositional reasoning as well:

```
1 subgoal

f : Z → Z
t0 : Z
t1 : Z
t2 : Z
t3 : Z
a : Z
H : t1 = t0 + a
H0 : t2 = t1 + a
H1 : ~ (t3 ≠ t2 + a ∨ 2 × a - 1 ≠ a)
H2 : f (f (f (t3 - 3))) = t0 ∨ f (t1 - 1) = t2 - 2
H3 : f (f t0) = t3 - 3
H4 : f t0 ≠ t0
============================
 False
```

In this goal, the `ti` are such that `ti - i` is a constant, and the remaining uses the same kind of equality and propositional reasoning than the second example we presented in Section 9.4.1. It is a valid goal and can be discharged in a tenth of a second by our tactic:

```
ergo.
```

```
Proof completed.
```

10.3.2 Conclusion

In this chapter, we have presented a non trivial solvable theory implemented in Coq and which can be used with the `CCX` framework presented in Chapter 9: the theory of linear integer arithmetic. It represents terms as polynomial with rational coefficients and solves equations using simple Gauss elimination. The development of this theory, and in particular the semantic proofs, are quite complex: it takes more than 4000 lines of definitions and specifications as a whole. In comparison, the whole development of `CCX`, including the proofs and the data structures `Uf`, `Diff` and `Use`, requires “only” 3500 lines of Coq. When instantiating our framework with this theory, we obtain a tactic which can automatically prove goals mixing arithmetic, equality and propositional reasoning, which no other tactic (or simple combination of tactics) could achieve so far.

Part III

Results, Conclusions and Perspectives

CHAPTER 11

Results and Analysis

> La perfection, ce n'est pas de faire quelque chose de grand et de beau, mais de faire ce que l'on fait avec grandeur et beauté.
>
> Swami Prajnanpad

Contents

In this chapter, we summarize the results we have obtained in the second part of this dissertation, and in particular the status of the reflexive tactics we provide. In Section 11.1, we give an overview of the final state of our implementation, and how the reflexive tactics can be used by Coq users. In Section 11.2, we present benchmarks to compare our reflexive implementation with existing Coq tactics. We address some of the limitations of our implementation and how it could be extended in Section 11.3. We finally conclude in Section 11.4 by presenting other integrations of automated provers, in particular SMT solvers, in interactive provers.

11.1 Overview of the tactic

11.1.1 Implementation

In Chapters 4 through 10, we have presented the implementation of our reflexive tactic in an iterative manner, we now give an overview of the complete development and how it is available to a user of the Coq proof assistant. The different files and their dependencies are represented in Fig. 11.1. This figure is just given for reference, and we have grouped files in clusters depending on their role; there are eight clusters in the figure corresponding to the DPLL implementation, formulae and literals, the reification, the lazy CNF conversion, CC(X), the theory of arithmetic, the toplevel tactic and miscellaneous helper files used in our development. The total size of this developement represents around 17000 lines of Coq and 1000 lines of extra OCaml code. Note that we did not include the Containers extension, which by itself represents 20000 lines of Coq and 1500 lines of OCaml. It does not comprises either the extra DPLL strategies and CNF conversions which we have presented earlier in this document, and which were implemented mainly for comparison purposes and are not included with the final tactic; they amount to more than 10000 lines of Coq.

This represents a quite substantial implementation, and to the best of our knowledge, this effort represents the largest reflexive decision procedure formalized and proved in the Coq proof assistant. It can be compared with the sizes of implementations of other Coq reflexive or semi-reflexive tactics which we have already presented: `ring` (3000 lines of Coq, 2000 lines of OCaml), `field` (800 lines of Coq, 200 lines of OCaml), `romega` (2500 lines of Coq, 1500 lines of OCaml). Théry and Letouzey [LT00] also formalized and proved an alternative decision procedure for SAT solving, Stålmarck's algorithm (10000 lines of Coq, 1000 lines of OCaml). Largest Coq developments are almost all libraries of generic definitions and results dedicated to some domain: C-CoRn [C-C](constructive mathematics, 85kloc), PFF [PFF](floating-point programs, 50kloc), CoLoR [BK09](termination proofs, 32kloc), SSReflect [GM08](Coq extension and group theory, 22kloc), Coccinelle [coc](term algebras and unification, 20kloc). One notable exception is the certified C compiler CompCert [Ler09b, Ler09a], which is the implementation and verification of a compiler for a substantial subset of the C language. It is the largest program verified in Coq (45kloc), but is only used after extraction and cannot be executed in Coq. In contrast to CompCert, our work has two extra requirements which come from the fact that we are implementing a reflexive tactic: it must be efficiently computable in the proof assistant, and it cannot rely on any axiom, whereas formalizations of external results or programs commonly use axioms like the excluded-middle, proof irrelevance or functional extensionality.

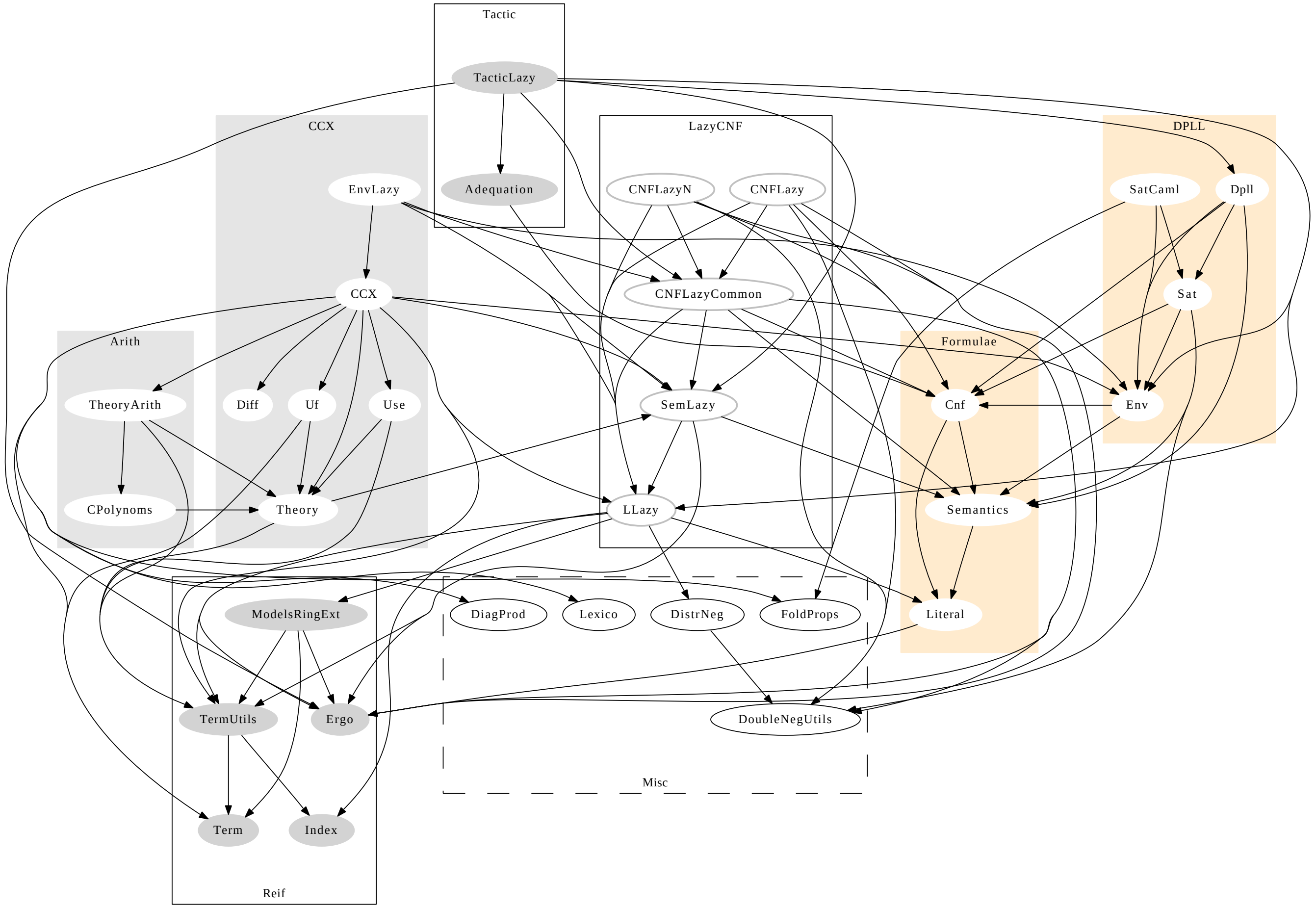


Figure 11.1: Dependency graph of our tactic implementation

11.1.2 Usage

There is no need to understand how the tactic is built in order to use it. To this end, we added an extra “top-level” module called `AltErgo`, not represented in Fig. 11.1, which contains all the necessary definitions and functor applications. It introduces a special functor parameterized by a CNF representation and an environment for DPLL, and which constructs everything up to the two reflexive tactics `unsat` and `valid` based on the given environment. While `unsat` proceeds by refuting the context in intuitionistic logic, `valid` proves that the current goal is valid and thus relies on classical logic. In practice, `valid` first checks if the module `Classical`, which is in the standard library and contains the excluded-middle and its classical consequences, has been imported by the user. If the excluded-middle is not available, the tactic fails with a comprehensive error message, inviting the user to use `unsat` instead or to explicitely require `Classical`. This way, we ensure that the user does not use `valid` and rely on classical logic inadvertently.

The `AltErgo` module uses the above functor to build tactics for the three following environments:

- the purely propositional environment `ENV` presented in Section 8.1.2, yielding a procedure solely based on DPLL;
- our `CCX` implementation instantiated on the empty theory `Empty_theory` presented in Section 9.1, yielding a procedure for the satisfiability of formulae modulo the theory $\mathcal{E}$ of equality with uninterpreted functions;
- an instantiation of `CCX` on the theory of linear integer arithmetic which we presented in Chapter 10, yielding a procedure for the satisfiability of formulae modulo the theory of equality and linear integer arithmetic.

The module introduces short names for all the possible tactics, so that the user does not have to access the modules generated in `AltErgo` explicitely:

`dpll/vdpll`: pure propositional tactics based on DPLL

`cc/vcc`: DPLL modulo equality with uninterpreted functions

`ergo/vergo`: DPLL modulo equality and linear integer arithmetic

When prefixed with `v`, the tactic uses the classical version, and the intuitionistic otherwise. We also provide alternate versions of all these tactics which use the lazy CNF conversion with n-ary connectives instead of binary connectives (see Section 7.3.3). These tactics have the same names as above, suffixed with an extra `n`; for instance, `dplln` is the pure propositional solver using n-ary proxies. Altogether, `AltErgo` provides four tactics for each of the three different different environments.

In order to use any of the tactics described above, it is therefore sufficient to import the module `AltErgo`:

```
Require Import AltErgo.

Theorem peirce : ∀A B, ((A → B) → A) → ~A → False.
Proof. dpll. Qed.
```

```
  peirce is defined.
```

The tactics we provide introduce all possible hypotheses from the goal, but they do not perform any reduction or unfolding of constants automatically, therefore it may be a good idea to simplify a goal with tactics like `simpl` or `cbv -[not]` beforehand, in order to unveil the propositional structure hidden under some terms. For instance:

```
Require Import Classical List.
Lemma In_list : ∀a b c, a = b → In b ((b+1)::c::a::nil).
Proof.
  vcc.
```

```
The formula in not valid.
The following countermodel has been found :
(In b (b + 1 :: c :: a :: nil))  :  false
(a = b)
```

But if we simplify the formula in order to unfold the definition of the list membership predicate `In`:

```
  simpl.
```

```
1 subgoal

============================
 ∀a b c : Z, a = b → b + 1 = b ∨ c = b ∨ a = b ∨ False
```

```
  vcc.
```

```
  Proof completed.
```

Of course, in these last two goals, we could have used respectively `ergo` and `vergo`, even though there was no equality reasoning in `peirce` and no arithmetic involved in `In_list`. We only provide the special tactics `dpll`/`vdpll` and `cc`/`vcc` because they can be slightly faster if the full power of `ergo` is not required, especially `dpll` since it does not perform congruence closure.

We finish this section by describing the typical proof term generated by our tactic; we detail the full proof term of the lemma `peirce` above:

```
1    peirce =
2  fun (A B : Prop) (H : (A → B) → A) (H0 : ~ A) ⇒
3  False_rec False
4    (let final := conj H0 H in
5     let finalt := ~ A ∧ ((A → B) → A) in
6     let _varmap__v :=
7       Node_vm True
8         (Node_vm A Empty_vm Empty_vm)
9         (Node_vm B Empty_vm Empty_vm) in
10    let _vtypes__v := Empty_vm in
11    let _vsymbols__v := Empty_vm in
12    let vm := mk_varmaps _varmap__v _vtypes__v _vsymbols__v in
13    let reif :=
14      FAnd (FNot (FVar (Left_idx End_idx)))
15        (FImp (FImp (FVar (Left_idx End_idx))
16                    (FVar (Right_idx End_idx)))
17           (FVar (Left_idx End_idx))) in
18    validity' reif finalt vm
19      (refl_equal Unsat<:dpll' vm reif = Unsat)
20      (refl_equal (Some finalt)<:binterp vm reif = Some finalt)
21      final)
22      : ∀A B : Prop, ((A → B) → A) → ~ A → False
```

After the introduction of hypotheses in line 2, the proof starts with `False_rec` in line 3, which denotes that it is a proof by contradiction. Lines 4 and 5 aggregates the hypotheses in a proof `final` of the formula `finalt` which must be refutated. Lines 6 to 12 introduce the varmaps used in the reification, in this case only the varmap for propositional variables is non-empty: it is defined on lines 6-9 and contains the variables `A` and `B`. The reified version `reif` of the formula is defined in lines 13-17, it simply follows the structure of `finalt` and replaces the variables by their paths in the varmap. Finally, the term ends with an application of the reflection lemma `validity'` and two of its arguments are actually equalities obtained by conversion. They are the two computations required by the reflexive tactic: line 19 checks that the call to the proof search `dpll'` on the reified formula returns `Unsat`, while line 20 checks that `reif` interprets to the formula `finalt`.

It is interesting to see how such a proof term scales with a larger goal: the formula to prove is introduced once in `finalt`, the varmaps have a size which is linear in the size of the formula, and so does the reified formula `reif` (with a factor at most 2). Finally, the application lemma has constant size (this is the reason why we introduce `finalt` as a local let-in), therefore the whole proof term is linear in the size of the formula to prove, with a reasonably small factor. Although the overhead due to reification can seem quite heavy on simple formulae like `peirce`, it is actually neglectible on larger goals in comparison to a non-reflexive, step-by-step, proof.

11.2 Benchmarks

We now proceed to a quantitative analysis of the performance of our tactics in comparison to the existing tactics. This analysis focuses on formulae that can be proved by the considered tactics, *i.e.* we do not compare tactics in terms of how (in)complete they are. For a more qualitative comparison of the capabilities of the existing tactics, see Section 11.3 below. In order to quantitatively compare different Coq tactics in an adequate manner, we claim that there are three different measures which must be taken into account:

- the time the tactic requires to prove the goal;
- the size of the proof generated by the tactic;
- the size of the tactic incantation.

Tactics are often compared only in regard of how fast they can prove a goal, but we believe that all three of these measures are equally important. A very slow tactic is almost useless of course, but a fast tactic which creates enormous proof terms is barely more useful since it will slow down the typechecking at `Qed`'s so much, and yield large compiled files which take longer time to be imported. The last measure is here to emphasize that we only consider completely automated tactics, *i.e.* a tactic which requires user input constant in the size of the goal. For instance, a tactic requiring manual reification, or CNF conversion, of the goal would not fulfil this condition.

11.2.1 Propositional Logic

We start by comparing our reflexive tactic for the fragment of pure propositional logic with `tauto`. We have already shown benchmarks in Chapter 7, more precisely in Fig. 7.3 page 164, but here we also focus on the size of generated proofs. In order to measure the size of proof terms, we wrote a small extension which counts the number of nodes in the AST internal representation of a proof term.

The results for several formulae are summarized in Table 11.1. We use two different families of propositional tautologies in order these tests: the first is the pigeon-hole formulae H_n defined in Chapter 2 page 23, and the second is due to de Bruijn and states that among $2n+1$ boolean variables set in a circular list, at least two adjacent variables are equal. More precisely, the de Bruijn formula with parameter n is defined as:

$$deb_n \equiv \forall x_0 \ldots x_{2n},\ \bigvee_{i=0}^{2n} (x_i \leftrightarrow x_{i+1 \texttt{ mod } 2n})$$

		`dplln`		`tauto`	
	V/C	time (s)	size	time (s)	size
H_3	6/9	0.01	330	0.75	12110
H_4	12/22	0.04	814	–	–
H_5	20/45	0.19	1684	–	–
H_6	30/81	1.27	3042	–	–
deb_5	11/22	0.02	372	20	67301
deb_{10}	21/42	0.03	686	–	–
deb_{15}	31/62	0.04	1008	–	–
deb_{20}	41/82	0.06	1348	–	–
deb_{100}	201/402	0.50	7216	–	–

Table 11.1: Comparison of `tauto` and `dplln` on propositional tautologies. The V/C column gives the number of variables and clauses in each formula.

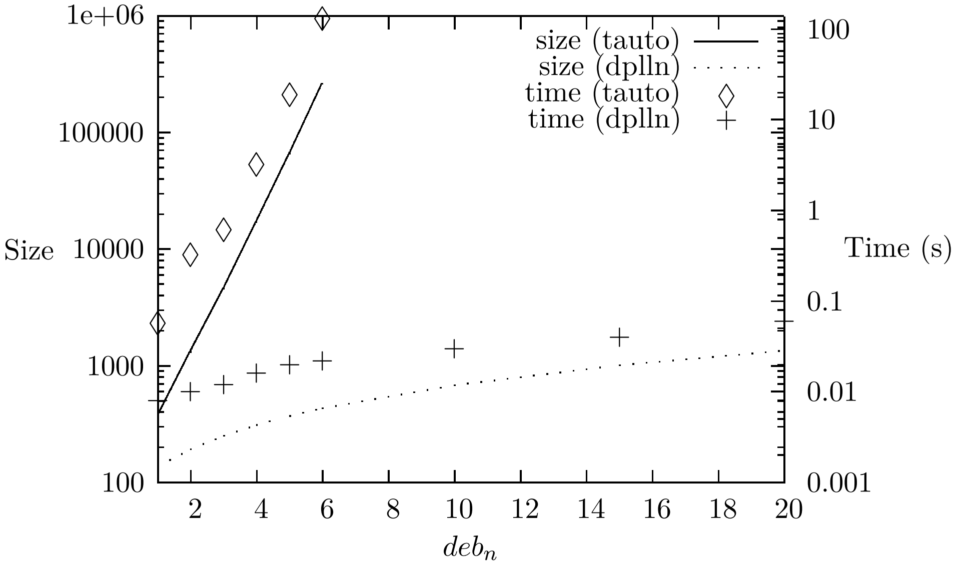


Figure 11.2: Size and speed comparison of `tauto` and `dplln` on De Bruijn formulae with varying sizes. The vertical scales for time and size are in logarithmic scale.

The figures in Table 11.1 confirm that our DPLL-based reflexive tactic is orders of magnitude faster than `tauto` because the latter times out (or runs out of available memory) for all but the easier formulae. An interesting thing to notice is that the size of proof terms generated by `dplln` are indeed linear in the size of the goal, and much smaller than proof terms generated by `tauto`. Figure 11.2, which shows the variation of proof sizes for the deb_n formulae as n increases, actually reveals that the proof size generated by `tauto` grows exponentially in n although deb_n itself is linear in n.

11.2.2 Adding Equality

Our implementation provides the tactic `cc` and its variants which combine propositional logic and equality with uninterpreted functions. The built-in tactic `congruence` can prove goals which are true by pure congruence closure; therefore we can start by comparing `cc` and `congruence` on goals which do not require any propositional reasoning. To that end, we use the formula $FP(n, m, k)$ defined as follows:

$$FP(n, m, k) \equiv \forall f x,\ f^n(x) = x \to f^m(x) = x \to f^k(x) = x$$

which is valid if and only if k is a multiple of $gcd(n, m)$. The figures are summarized in Table 11.2, where we tested both tactics for various values of n, m and k.

	`cc`		`congruence`	
	time (s)	size	time (s)	size
$FP(5,3,1)$	0.02	324	0.008	242
$FP(9,4,1)$	0.08	385	0.012	390
$FP(13,5,1)$	0.16	446	0.016	1475
$FP(13,12,1)$	0.39	537	0.004	192
$FP(25,2,1)$	1.0	551	0.016	2280
$FP(25,11,1)$	1.7	668	0.1	14848
$FP(25,13,1)$	0.36	694	0.02	1722
$FP(25,15,5)$	0.13	772	0.02	1944
$FP(25,24,24)$	5.0	1136	0.004	121

Table 11.2: Comparison of `cc` and `congruence`.

The results of this comparison show that `congruence` is significantly faster than `cc`, while generating proof terms with reasonable size in most cases. We can explain that by the fact that `congruence` uses a very efficient congruence closure analysis and also reconstructs proof terms using ad-hoc lemmas which keep the proof as short as possible. In contrast, our implementation of congruence closure is not specifically designed for this kind of goals but is designed to be used as an environment by a DPLL procedure.

In particular, it performs the whole congruence closure of the equalities it is being passed: this explains why it does not behave well on special easy cases of the form $FP(n, n-1, 1)$ or, even worse, $FP(n, m, m)$. Consider for instance what happens when dealing with $FP(25, 24, 24)$: CC(X) is being passed two equalities $f^{25}(x) = x$ and $f^{24}(x) = x$, and this leads to many new equalities, which are merged until all terms of the form $f^m(x)$ with $m \leq 25$ are equal to x in the union-find. Only then is CC(X) passed the literal $f^{24}(x) \neq x$ which leads to a contradiction, and therefore it ends up taking much longer than `congruence` which simply finds the obvious proof without performing the whole congruence closure. One consequence of this is that `cc` proves $FP(n, m, k)$ in a time which does not depend on k; conversely, we see that the proof found by `congruence` can vary a lot depending on the parameters, e.g. the proof for $FP(25, 11, 1)$ is much larger than other proofs for $n = 25$.

Note that the issue which the above discussion raises about our implementation of CC(X) is specific to the Coq implementation: a standard implementation would only merge all the pending equations during queries in a lazy manner, as long as the query isn't obviously true. This also emphasizes that one should prefer the dedicated built-in tactic `congruence` over our tactics to deal with purely equational goals. It is more relevant to look at how `cc` behaves on goals which require mixed propositional and equational reasoning.

By using the fact that `intuition` can be "chained" with a tactic to be applied to remaining branches in the proof search, we can use the tactic `intuition congruence` to solve the same fragment as `cc` and its variants. We use the two following generic formulae which depend on an integer parameter n:

$$\begin{aligned} D_n &\equiv \left(\bigwedge_{i=0}^{n-1} (x_i = y_i \wedge y_i = x_{i+1}) \vee (x_i = z_i \wedge z_i = x_{i+1})\right) \to x_0 = x_n \\ D_n^f &\equiv \left(\bigwedge_{i=0}^{n-1} (x_i = y_i \wedge y_i = f(x_{i+1})) \vee (x_i = z_i \wedge z_i = f(x_{i+1}))\right) \to x_0 = f^n(x_n) \end{aligned}$$

Intuitively, D_n links every x_i to x_{i+1} by transitivity using either y_i or z_i at each step, while D_n^f is similar but adds an application of some symbol f at each step (therefore requires congruence reasoning, whereas strictly speaking D_n only requires equivalence).

The figures are summarized in Table 11.3 and show that the two tactics are rather similar. The combination of `intuition` and `congruence` is slightly faster than `cc`, but it is especially faster on smaller goals (*i.e.* where it doesn't make much of a difference), whereas for bigger goals it suffers from the fact that it creates very large proof terms, while proof terms of `cc` remain linear in the input formula. For values of n larger than 9, `intuition congruence` actually generates proofs that are several megabytes large and

	cc		intuition congruence	
	time (s)	size	time (s)	size
D_3	0.08	1225	0.03	1986
D_4	0.18	1772	0.07	5220
D_5	0.40	2447	0.20	13134
D_8	4.0	5400	3.9	181868
D_3^f	0.12	1398	0.07	3001
D_4^f	0.29	1997	0.16	8474
D_5^f	0.66	2728	0.42	22467
D_8^f	7.2	5873	6.0	342622

Table 11.3: Comparison of `cc` and `intuition congruence`.

where the time taken by the `Qed` (not included in our timings) is much larger than the tactic application itself.

Therefore, even if our tactic is not fast in comparison to `congruence`, when it involves propositional reasoning, combination of the existing tactics suffer from the combined weaknesses of `intuition` and `congruence`: relatively slow propositional reasoning and large proof terms. Because it is fully reflexive, `cc` does not have such shortcomings.

11.2.3 Adding Arithmetic

We finally focus our quantitative comparison on goals which require reasoning with linear arithmetic, using the built-in tactic `omega`.

	ergo		omega	
	time (s)	size	time (s)	size
$\mathcal{F}_2$	0.03	436	0.02	1037
$\mathcal{F}_5$	0.12	1073	0.19	6177
$\mathcal{F}_{10}$	0.48	2976	0.56	21727
$\mathcal{F}_{15}$	1.35	6259	1.68	41780
$\mathcal{F}_{20}$	3.14	11323	4.0	68895

Table 11.4: Comparison of `ergo` and `omega`.

As with equality, we start with formulae which do not involve any propositional reasoning to better isolate the behavior of `omega` in regard to the `ergo` tactic. The results of these first tests are displayed in Table 11.4. They use a family of formulae defined as:

$$\mathcal{F}_n \equiv \left(\bigwedge_{i=2}^{n} x_i = x_{i-1} + x_{i-2}\right) \to x_0 = 1 \to x_1 = 1 \to x_n = \mathit{fibo}(n)$$

where $\mathit{fibo}(n)$ returns the n-th Fibonacci number. It is clear that the $\mathcal{F}_n$

are valid since they simply "unfold" the definition of the n-th Fibonacci number is a conjunction of equalities. The results show that the two tactics are really similar, with a slight edge of about 15% in favor of `ergo`, but it does not show any significant difference between the tactics, although they are based on completely different methods. Note that even if `omega` creates proof terms which are much bigger than `ergo`, this is not as big an issue as with `tauto`, since the proof terms created by `omega` still grow linearly (there is an approximate factor of 7 with `ergo`).

	`ergo`		`intuition omega`	
	time (s)	size	time (s)	size
D_2^+	0.11	949	0.07	6428
D_3^+	0.28	1458	0.18	18648
D_4^+	0.68	2094	0.56	49509
D_5^+	1.6	2856	1.6	123589
D_6^+	3.7	3768	4.2	297549
D_7^+	8.3	4846	11.9	698397

Table 11.5: Comparison of `ergo` and `intuition omega`.

We now combine propositional and arithmetical reasoning and compare the behaviour of `ergo` and of the combination of tactics `intuition omega`. To that end, we use a variation of the formulae D_n and D_n^f which we used to test equality reasoning earlier:

$$\begin{aligned} D_n^+ \quad \equiv \quad & \left(\bigwedge_{i=0}^{n-1} (x_i = y_i \land y_i = 1 + x_{i+1})) \lor (x_i = z_i \land z_i = 1 + x_{i+1})\right) \\ & \rightarrow x_0 - n = x_n \end{aligned}$$

The results are summarized in Table 11.5 and show that although the combination of `intuition` and `omega` is slightly faster for small values of n, the situation is inversed as n increases, and this is also linked to the fact that the size of the proof terms generated by the built-in tactic are growing very large when n increases. In comparison, the performance of `ergo` remains reasonable and its proof terms quite small.

11.3 Limits and Extensions

In this section, we investigate the limits of our implementation in regard to existing tactics which deal with the same fragments. We also discuss whether it would be possible to extend our development in order to go beyond these limits, and how much work this would represent.

11.3.1 Interpreted Predicate Symbols

The decision procedure which we implemented in Chapter 10 is used in our tactic to deal with the theory of linear integer arithmetic. It is closely related to the theory which is decided by the `omega` tactic (introduced Section 4.2.2), with the important difference that our tactic is unable to treat inequalities. This is a serious limitation in practice since many interesting goals use inequalities; for instance, proof obligations coming from static program verification with the Why platform often involve inequalities to avoid out-of-bounds accesses or to express loop invariants.

Because of its practical significance, an extension of our implementation to inequalities is certainly the first extension we would consider. We are confident that this would be possible because Alt-Ergo uses the same core algorithm and also manages inequalities. More generally, the implementation of CC(X) in Alt-Ergo is ahead of the formalization we presented in Chapter 3: theories can interpret not only terms, but also atoms. In practice, theories have some interpreted predicate symbols, are being passed relevant atoms during the proof search, and are queried for the status of interpreted literals. There is some boilerplate which keeps these theory literals in synchronization with the union-find in CC(X).

In the case of linear integer arithmetic, interpreted literals would be inequalities between polynomials and the theory can treat these literals by implementing an incremental Fourier-Motzkin procedure or a simplex algorithm. A Coq implementation would not be too difficult, but the proofs of the whole system would have to be changed considerably. In particular, the formalization of CC(X) (both on paper and in Coq) must first be extended to account for interpreted literals.

11.3.2 Propositional Simplification

We have emphasized several times already that the `intuition` tactic, Coq's built-in intuitionistic propositional solver, can also be used as a propositional simplifier to explore the propositional structure of the formula and clear as many branches as possible, yielding only the remaining cases. Our tactic does not allow this, and simply either succeeds or fails with the first countermodel found.

We could actually change our system so that it does not stop at the first countermodel, but instead finds a complete set of satisfying assignments by traversing the whole proof tree. It is similar to well-known variations of the SAT problem such as MAX-SAT, the Maximum Satisfiability problem, or #SAT, the problem of counting the number of satisfying assignments. We give a possible adaptation of our basic DPLL inference system in Fig. 11.3. Sequents are of the form $M | \Gamma \vdash \Delta$ where M is the set of countermodels found in the derivation of that sequent. There is an extra rule Sat which

$$\textsc{Red}\ \frac{M|\,\Gamma, l \vdash \Delta, C}{M|\,\Gamma, l \vdash \Delta, \bar{l} \vee C} \qquad \textsc{Elim}\ \frac{M|\,\Gamma, l \vdash \Delta}{M|\,\Gamma, l \vdash \Delta, l \vee C}$$

$$\textsc{Assume}\ \frac{M|\,\Gamma, l \vdash \Delta}{M|\,\Gamma \vdash \Delta, \{l\}} \qquad \textsc{Conflict}\ \frac{}{\emptyset|\,\Gamma \vdash \Delta, \emptyset}$$

$$\textsc{Split}\ \frac{M|\,\Gamma, l \vdash \Delta \qquad M'|\,\Gamma, \bar{l} \vdash \Delta}{M \cup M'|\,\Gamma \vdash \Delta} \qquad \textsc{Sat}\ \frac{}{\{\Gamma\}|\,\Gamma \vdash}$$

Figure 11.3: Finding all satisfying assignments with DPLL

is used when the problem becomes empty and a countermodel is found, and the unsatisfiability of a formula Δ would be equivalent to the derivability of $\emptyset|\,\emptyset \vdash \Delta$.

With a proof search returning all possible assignments, it is possible to write a tactic based on this procedure which solves a goal if possible, and otherwise replaces the goal by one subgoal per countermodel, each subgoal representing the refutation of such countermodel. This would become even more interesting with an extension to SMT, since the countermodel could comprise the union-find computed by CC(X). Therefore, we can imagine that in each subgoal, the terms of the problem would be canonized according to the union-find corresponding to this subgoal. This would be a feature similar to the command `simplify` available in PVS and which simplifies a goal using decision procedures, and it would be entirely formalized in an LCF-style prover like Coq. An interesting issue would arise because of the way CC(X) uses semantical values to canonize terms: when we use the theory of arithmetic, we do not actually have a term canonizer but a union-find on polynomials, and the representative of a term can be a polynomial which doesn't correspond to a term. It is therefore not straightforward to forward all relevant equalities from the union-find to the user's formula.

11.3.3 Non-Linear Integer Arithmetic

We have shown in Section 10.2.2 how we used, in our proofs of the theory of linear integer arithmetic, the reflexive tactic `ring` which is dedicated to ring structures. One limitation of our theory is that it is restricted to linear arithmetic and only interprets multiplication by constants. Although `ring` does not solve Peano arithmetic, it does a bit more than our tactic as far as non-linear arithmetic is concerned, because it deals with the full commutative ring structure of the relative integers. In other words, in comparison with our tactic, `ring` also uses the associativity and commutativity of the multiplication symbol.

We know that we cannot apply CC(X) with non-linear arithmetic since it is an undecidable theory. Nonethess, a question which arises naturally is whether it would be possible to add at least the associativity and commutativity (AC) of multiplication so that the procedure in our tactic subsumes `ring`. To that end, the type of polynomials which we used in our theory of arithmetic would have to be changed slightly: monomials shall not be reduced to a single variable anymore, but shall be extended to ordered products of variables. For instance, `5 * x * y * z` would be a monomial with three variables. Solving equalities between such polynoms yields substitutions where the left-hand side is not reduced to a single variable, but to an ordered product of variables. Such substitutions cannot simply be "applied" to other polynomials and that CC(X) is not because they be applied easily and CC(X) is not adapted to such theories.

More generally, this issue is about the ability to define AC symbols in CC(X) and have the procedure correctly compute the congruence closure modulo AC. This is a complex modification to Alt-Ergo's core procedure which has been investigated and implemented by M. Iguernelala [CCI10]. The treatment of AC symbols is complex and typically requires dealing with critical pairs, it cannot be implemented as a theory but must be intertwined with CC(X). This extension of CC(X) is too complex to be easily applicable to our Coq implementation.

11.3.4 Theory of Constructors

Our tactic performs congruence closure with uninterpreted functions and therefore does not distinguish whether these function symbols are constructors of inductive types or not. The `congruence` tactic, on the contrary, does more than its name suggests: it not only performs congruence closure but also reasons modulo the theory of *constructors*.

This theory is defined by the fact that constructors of an inductive type are symbols which have two special properties: discriminability and injectivity. Discriminability means that terms starting with different constructors of the same type are necessarily different, while injectivity means that each constructor is injective with respect to its arguments. These properties are not "meta" properties, in the sense that they are not explicitly added or specifically checked by the system for each constructor; instead, they are a consequence of the elimination principle which is provided at the definition of the inductive. For example, let us define this simple two-branch inductive:

Inductive *t* := A (*x* : nat) | B.

We can write a discrimination lemma A 0 $\neq$ B explicitly:

Definition discrAB (H : A 0 = B) : False :=
  let P := **fun** *t* ⇒

```
    match t with | A _ ⇒ True | B ⇒ False end in
  @eq_ind t (A 0) P I B H.
```

```
discrAB is defined
```

It works by taking advantage of the elimination principle to define a predicate `P` which is true for terms starting with constructor `A`, and false otherwise. It is then easy to give a proof of `P (A 0)`, and by using the equality `A 0 = B`, it becomes a proof of `P B`, *i.e.* `False` by conversion. Injectivity can be obtained in a similar manner, and the built-in tactics `discriminate` and `injection` automatically build the adequate discrimination and injection terms when applied.

Since inductives are used extensively in Coq, in particular both for types and predicates, these properties are used a lot in a typical development. We would therefore be pleased if we could adapt our tactic to reason modulo the theory of constructors. The adaptation of CC(X) to deal with this theory is straightforward[1]: it suffices to consider a new kind of function symbol C_i^I, the i-th constructor of inductive I, and to add the discriminability and injectivity rules to CC(X).

$$\textsc{Inject}\ \frac{\langle\ \Theta \mid \Gamma \mid \Delta \mid N \mid C_i^I(t_1,\ldots,t_n) = C_i^I(u_1,\ldots,u_n)\ ;\ \Phi\ \rangle}{\langle\ \Theta \mid \Gamma \mid \Delta \mid N \mid t_1 = u_1\ ;\ \ldots\ ;\ t_n = u_n\ ;\ \Phi\ \rangle}$$

$$\textsc{Discr}\ \frac{\langle\ \Theta \mid \Gamma \mid \Delta \mid N \mid C_i^I(t_1,\ldots,t_n) = C_j^I(u_1,\ldots,u_m)\ ;\ \Phi\ \rangle}{\langle\ \ \bot\ \mid\ \Phi\ \ \rangle}\ i \neq j$$

Similar rules should be added to deal with constructors adequately when assuming a disequation or treating a query. These modifications show that it is easy to adapt CC(X) in order to deal with the theory of constructors.

It would be similarly straightforward to adapt our CC(X) implementation in Coq to use the theory of constructors. Unfortunately, we would be unable to justify these modifications semantically, *i.e.* to prove the reflection lemma. The reason for this is that in order to justify, on the Coq side, the manipulations we do with constructors of inductive types on the reified side, we must be able to prove discrimination and injection lemmas. We cannot prove these lemmas unless we know these reified constructors correspond to an inductive (or coinductive) definition, but being an inductive is not a property of the logic, it is a meta property which comes from the ability to write elimination principle. In other words, there is no Coq predicate `IsInductive : Type → Prop` which identifies inductive types.

[1] Note that this is a fundamental difference between the theory of constructors and the – stronger – theory of algebraic datatypes. In the latter, we also have the fact a term of an inductive type has the form $C_i(\bar{x})$ for one of the constructors C_i of that type. In contrat to the theory of constructors, this theory is not convex and cannot be dealt with by CC(X).

Therefore, even if we could reify constructors in a special way and treat the theory of constructors in our implementation, we cannot semantically justify this theory. In the best case, we could do that for a finite, predefined, number of concrete inductive types (`option`, `bool`, etc) which we could justify individually, but there is no way to justify the theory of all constructors in general. This can be seen as a limitation to the reflexive approach.

11.3.5 First-Order Logic

The last limitation which we investigate in the fact that we only deal with quantifier-free formulae, and this is maybe what separates our tactic the most from the actual Alt-Ergo theorem prover. In order to deal with first-order logic, Alt-Ergo's SAT solver alternates phases of proof search and of generation of new formulae (called a *matching* phase). The proof search simply follows the DPLL procedure which we have described in Chapter 2, but when a countermodel is found a matching phase is launched: it uses all the lemmas (*i.e.* the universally quantified literals) in the partial assignment and instantiates them on terms of the problem. This adds new clauses to the problem and the proof search is started again. Of course, this procedure does not terminate, and the prover could keep on searching and adding new instantiations forever so only a finite number of matching phases is allowed in practice. Knowing this, there are two ways we could extend our tactic to first-order logic:

1. The first way would be to use Alt-Ergo as an external oracle to guess the ground instantiations which are sufficient to establish the unsatisfiability of the original first-order formula. In practice, this means instrumenting Alt-Ergo so that, when it successfully proves a formula, it also returns a trace of all lemma instantiations performed on the way. The Coq tactic could start by calling Alt-Ergo on this formula and retrieves the list of sufficient ground instantiations, apply these instantiations in the Coq goal and then call the ground reflexive tactic. The main advantage of this method is that we avoid having to extend our Coq implementation and formalization to first-order logic, in particular we avoid having to reason about binders, which is notoriously difficult [ACP$^+$08, Cha09]. The main inconvenient is that we need an external tool and therefore the tactic cannot simply be a Coq plugin. Also, modifying Alt-Ergo in order to retrieve the ground instances must not be done naively and this requires a dependency analysis as complex as the one necessary to use backjumping with SMT and discussed in Section 2.3.

2. The second way would simply be to continue using the fully reflexive approach and extend our implementation presented in this dissertation to first-order logic. Note that the modifications are restrained to the

literals and the SAT solver, as the CC(X) environment will continue to deal with ground atoms. The main difficulty lies in extending the formalization of semantics and the reification process to embed quantifiers and bound variables. The main advantage is that it would lead to a much tighter integration than using the external oracle's method.

11.4 Automation by Proof Reconstruction

We have explained in Chapter 4 that a common method to implement a tactic for automated deduction without using full reflection is to use an external tool generating traces and reconstruct a Coq proof from those traces. To conclude this chapter, we now recount existing integrations of automated provers in interactive provers.

Until recently, examples of integrations of SMT solvers into interactive provers were rather sparse and the most successful integrations were actually related to automated theorem provers of the TPTP family. The most popular integration is Sledgehammer [MQP06b] by Paulson et al., an integration of three different ATPs in Isabelle/HOL [Isa]: Sledgehammer sends goals to the E prover [Sch02], SPASS [WDF$^+$09] and Vampire [PSS02], and let these three tools work in the background. When a proof is found, it is reconstructed into an Isabelle proof, via the Metis [Met] theorem prover. Provers can run for a long time since the user can continue its proof while they run in the background. Metis is used as an intermediate because it is able to output proof traces based on six simple inference rules. This feature was developed as part of an integration of Metis into HOL4 [GM93].

There have been also a few integrations of decision procedures inspired by SMT solvers in interactive provers of the HOL family. Mclaughlin, Barrett and Ge [MBG06] have described an integration of CVC Lite in HOL Light for the fragment of quantifier-free formulae with equality on uninterpreted functions, linear real arithmetic and arrays. This work was later extended [GB08] to quantifiers and linear integer arithmetic for CVC3 [BT07]. Fontaine et al. [FyMM$^+$06] describe an integration of haRVey, an earlier version of the VeriT SMT solver [BdODF09], in Isabelle/HOL. Their integration encompasses quantifier-free first-order logic with equality on uninterpreted functions, *i.e.* the exact scope of our `cc`/`vcc` tactics. This work was later extended to quantified formulae in [HCF$^+$07]. The most recent and most complete integration was proposed by Weber and Böhme in [BW10]: they present a reconstruction mechanism for proof traces of the Z3 SMT solver into HOL. They put the emphasis on efficiency and their reconstruction phase takes less time than the proof search in Z3 for most benchmarks in the SMT-LIB, which is very encouraging. This work builds on earlier work limited to SAT traces by Weber and Amjad [WA09b].

All the integrations presented so far have been made in provers of the

HOL family; this can be explained by the fact that HOL and Isabelle are better suited for proof reconstruction from an external solver than a proof assistant like Coq. The main reason for that is that these HOL provers build proofs using a small number of basic inference steps, encoded as ML functions manipulating an abstract type of theorem `thm`. The soundness of the system is ensured by typechecking the applications of these basic combinators and by the fact that the type `thm` is abstract, thus can only be manipulated using these basic steps. Therefore, a proof in these systems corresponds more or less to proving that a formula has the type of a theorem, and the proof itself, *i.e.* the chain of steps which led to the theorem object, is verified on-the-fly and is not kept in memory. In contrast, we have seen that in Coq one proceeds by constructing a full proof term for a theorem, and the proof term is typechecked at the end, and is kept in memory to be eventually reduced or re-checked (see [KW10] for a traduction of HOL proofs to Coq proofs). The consequence of these observations is that Coq is much more sensible to large proofs: first because its typechecking mechanism is for a very rich logic and is much slower than HOL's, second because the whole proof is kept in a compiled Coq file. Therefore, proof reconstruction of large traces in Coq is less efficient than in provers of the HOL family.

One way to counter this proof term issue is to rely on one of Coq's strongest asset, its ability to compute efficiently using the virtual machine. In [AGST10], Armand et al. present a reconstruction of SAT traces in Coq, which they have optimized to take advantage of faster reductions. They indeed remark that Coq's performance while typechecking and manipulating large proof terms is unsatisfactory, and they compensate by modifying the virtual machine in order to use imperative arrays to execute algorithms described, in Coq, using functional arrays. This granted them a large speed-up and they end up with timings similar to Weber and Amjad [WA09b].

Our work could also benefit from these imperative features introduced in [AGST10]. In choosing to follow the fully reflexive approach in our work, we have of course taken a radical path which bases everything on efficient internal computations in the logic, and require neither proof reconstruction nor large proof terms. Of course, the downside was that implementing the kernel of an SMT solver in the proof assistant required a large complex development, but our strategy also aimed at formalizing our algorithms since they are used in an external SMT solver. When formalizing a fully reflexive procedure, we actually certify that the *procedure* is correct, whereas when verifying proof traces from an external tool, one is only formally checking that one particular run of the solver is correct. This is therefore natural that the fully reflexive approach be more complex to implement than proof reconstruction, since it achieves a much stronger verification result.

Conclusion

In this thesis, we have presented a formalization of the kernel of Alt-Ergo in the Coq proof assistant and an implementation of a reflexive tactic based on this formalization. This work uses SMT technology to bring more automation to an interactive prover, without sacrificing the correctness of the prover or augmenting the trusted kernel. Our contribution is thus twofold:

1. We have comforted the trust that we have in our SMT solver by formally proving the soundness of the components which are at Alt-Ergo's heart. In order to do this, we took advantage of the fact that Alt-Ergo's implementation is modular and that the propositional solver, the congruence closure mechanism and the decision procedures for the various background theories are implemented as separate components. Although Alt-Ergo's propositional solver relies on a well-known DPLL procedure, our presentation based on inference rules is as general as possible and at the same time remains very close to the actual implementation. Therefore it is reasonably easy to reason about, but the gap with the actual software is quite limited, even when adding backjumping and conflict-driven clause learning. We also presented our original algorithm CC(X) for combining the theory of equality on uninterpreted functions with a solvable theory X. This work is inspired by Shostak's algorithm but allows underlying decision procedures to use abstract data structures instead of terms. Under some conditions on the theory X, we have formally proved the soundness and completeness of this algorithm in Coq. The proof is quite involved and it is a real asset to have been able to formally verify such proof in an interactive prover.

2. We have not only formalized and proved the correctness of Alt-Ergo's core components in Coq, but we have used these formalizations in a reflexive manner to obtain a Coq tactic combining propositional logic,

congruence closure and linear integer arithmetic. The main specificity when formalizing an algorithm to use in a reflexive tactic is that it must also be an executable implementation. To that end, we contributed a library of first-class containers based on Coq's type classes mechanism and which allows easy intuitive use of basic data structures in Coq programs. Another difficulty which is specific to the use of reflection is the reification process; in particular, we showed how to reify terms in an arbitrary signature using dependently-typed interpretations. Following the modular structure of Alt-Ergo, we formalized the different components separately and made extensive use of Coq's own module system to isolate the different parts of the system: formulae, semantics, CNF conversion, strategies, theories, etc. This allowed us to define clear interfaces between the different components, to write the proofs in a modular manner, and thanks to encapsulation to implement and test different versions of some components (DPLL strategies, DPLL environments, CNF conversion methods) and easily plug them in the framework. On the front end, we use these different subcomponents to provide twelve different versions of the reflexive tactic, all obtained by different instantiations of the framework.

On a more general note, our work demonstrates that the fully reflexive approach can also be used for larger systems. In particular, Coq can be used as a full-blown, albeit purely functional, programming language and the virtual machine provides reasonably efficient computation of Coq programs. This approach does not require the use of an external tool or proof reconstruction and is therefore easier to maintain; moreover, the implemented procedure is formally proved and it can also be extracted to a standard programming language and become a certified solver.

Published Work

[1] Sylvain Conchon, Evelyne Contejean, Johannes Kanig, and Stéphane Lescuyer. Lightweight Integration of the Ergo Theorem Prover inside a Proof Assistant. In John Rushby and N. Shankar, editors, *AFM07 (Automated Formal Methods)*, 2007.

[2] Sylvain Conchon, Evelyne Contejean, Johannes Kanig, and Stéphane Lescuyer. CC(X): Semantic Combination of Congruence Closure with Solvable Theories. *Electr. Notes Theor. Comput. Sci.*, 198(2):51–69, 2008.

[3] François Bobot, Sylvain Conchon, Evelyne Contejean, and Stéphane Lescuyer. Implementing Polymorphism in SMT solvers. In Clark Barrett and Leonardo de Moura, editors, *SMT 2008: 6th International Workshop on Satisfiability Modulo*, 2008.

[4] Stéphane Lescuyer and Sylvain Conchon. A Reflexive Formalization of a SAT Solver in Coq. In *TPHOLs 2008: In Emerging Trends of the 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, 2008.

[5] Stéphane Lescuyer and Sylvain Conchon. Improving Coq Propositional Reasoning Using a Lazy CNF Conversion Scheme. In *FroCoS'09: Frontiers of Combining Systems*, 2009.

[6] Stéphane Lescuyer. Conteneurs de première classe en Coq. In *Vingt-et-unièmes Journées Francophones des Langages Applicatifs*, La Ciotat, January 2010. INRIA.

Bibliography

[Abr96] J.-R. Abrial. *The B-book: assigning programs to meanings.* Cambridge University Press, New York, NY, USA, 1996. 6

[ACL] Applicative common lisp 2. `http://www.cs.utexas.edu/users/moore/acl2/`. 6

[ACP$^+$08] Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages*, January 2008. 243

[ACTZ07] Andrea Asperti, Claudio Sacerdoti Coen, Enrico Tassi, and Stefano Zacchiroli. Crafting a proof assistant. In *TYPES'06: Proceedings of the 2006 international conference on Types for proofs and programs*, pages 18–32, Berlin, Heidelberg, 2007. Springer-Verlag. 5

[AGST10] Michaël Armand, Benjamin Grégoire, Arnaud Spiwack, and Laurent Théry. Extending coq with imperative features and its application to sat verication. In *Interactive Theorem Proving, international Conference, ITP 2010, Edinburgh, Scotland, July 11-14, 2010, Proceedings*, Lecture Notes in Computer Science. Springer, 2010. 245

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer Verlag, 2004. 139

[BCCL08] François Bobot, Sylvain Conchon, Evelyne Contejean, and Stéphane Lescuyer. Implementing Polymorphism in SMT solvers. In Clark Barrett and Leonardo de Moura, editors, *SMT 2008: 6th International Workshop on Satisfiability Modulo*, 2008. 19

[BDD07] Richard Bonichon, David Delahaye, and Damien Doligez. Zenon : An extensible automated theorem prover producing checkable proofs. In *LPAR*, pages 151–165, 2007. 98, 99

[BDN09] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of agda — a functional language with dependent types. In *TPHOLs '09: Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, pages 73–78, Berlin, Heidelberg, 2009. Springer-Verlag. 5

[BdODF09] Thomas Bouton, Diego Caminha B. de Oliveira, David Déharbe, and Pascal Fontaine. verit: an open, trustable and efficient smt-solver. In Renate A. Schmidt, editor, *Proc. Conference on Automated Deduction (CADE)*, Lecture Notes in Computer Science. Springer-Verlag, 2009. To appear. 244

[BFPR06] Gilles Barthe, Julien Forest, David Pichardie, and Vlad Rusu. Defining and reasoning about recursive functions: a practical tool for the coq proof assistant. In *In Functional and Logic Programming (FLOPS'06), LNCS 3945*, pages 114–129. Springer, 2006. 198

[BHdN02] Marc Bezem, Dimitri Hendriks, and Hans de Nivelle. Automated proof construction in type theory using resolution. *JAR*, 29(3):253–275, 2002. 98

[BK09] Frédéric Blanqui and Adam Koprowski. Automated Verification of Termination Certificates. Research Report RR-6949, INRIA, 2009. `http://color.inria.fr/`. 228

[BM88] R. S. Boyer and J. S. Moore. Integrating decision procedures into heuristic theorem provers: a case study of linear arithmetic. pages 83–124, 1988. 15

[BM90] Robert S. Boyer and J. Strother Moore. A theorem prover for a computational logic. In *Proceedings of the 10th International Conference on Automated Deduction*, pages 1–15, London, UK, 1990. Springer-Verlag. 6, 15

[Bou97] Samuel Boutin. Using reflection to build efficient and certified decision procedures. In *TACS*, pages 515–529, 1997. 99

[BRS05] Mike Barnett, Leino Rustan, and Wolfram Schulte. *The Spec# Programming System: An Overview*, volume 3362/2005 of *Lecture Notes in Computer Science*, chapter 3, pages 49–69–69. Springer, Berlin / Heidelberg, January 2005. 16

[BRVs95] Rolf Backofen, James Rogers, and K. Vijay-shanker. A first-order axiomatization of the theory of finite trees. *Journal of Logic, Language and Information*, 4:5–39, 1995. 14

[Bry92] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992. 22

[BSST09] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009. 15

[BST10] Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010. 15, 18, 52

[BT07] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany. 17, 55, 244

[BW10] Sascha Böhme and Tjark Weber. Fast LCF-style proof reconstruction for Z3. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6172 of *Lecture Notes in Computer Science*, pages 179–194. Springer, 2010. 244

[C-C] C-corn: Constructive coq repository at nijmegen. `http://c-corn.cs.ru.nl`. 228

[CC05] Evelyne Contejean and Pierre Corbineau. Reflecting Proofs in First-Order Logic with Equality. In Robert Nieuwenhuis, editor, *CADE-20*, volume 3632 of *LNAI*. Springer, 2005. 101

[CCF$^+$] Evelyne Contejean, Pierre Courtieu, Julien Forest, Olivier Pons, and Xavier Urbain. Certification of automated termination proofs. In *FroCos 07*, volume 4720 of *LNAI*. Springer. 101

[CCI10] S. Conchon, E. Contejean, and M. Iguernelala. Ground ac completion modulo shostak theories. In *Logic for Programming, Articifial Intelligence and Reasoning 17*, Yugyakarta, Indonesia, 2010. Easychair Volume. Short paper. 241

[CH88] Thierry Coquand and Gerard Huet. The calculus of constructions. *Inf. Comput.*, 76(2-3):95–120, 1988. 5, 86

[Cha] Arthur Charguéraud. Extended set of coq tactics. `http://www.chargueraud.org/arthur/projects/proofs/tactics/`. 96

[Cha09] Arthur Charguéraud. The locally nameless representation. To appear in Journal of Automated Reasoning. http://arthur.chargueraud.org/research/2009/ln/, 2009. 243

[Chl] Adam Chlipala. *Certified Programming with Dependent Types.* `http://adam.chlipala.net/cpdt/`. 96

[Chr03] Jacek Chrząszcz. Implementation of modules in the Coq system. In *TPHOLs*, volume 2758 of *Lecture Notes in Computer Science*, pages 270–286. Springer, 2003. 94, 106, 132

[CL07] Jean-François Couchot and Stéphane Lescuyer. Handling polymorphism in automated deduction. In *CADE-21: Proceedings of the 21st international conference on Automated Deduction*, pages 263–278, Berlin, Heidelberg, 2007. Springer-Verlag. 18

[CLS96] David Cyrluk, Patrick Lincoln, and Natarajan Shankar. On shostak's decision procedure for combinations of theories. pages 463–477. Springer-Verlag, 1996. 56

[coc] Coccinelle. `http://www.lri.fr/~contejea/Coccinelle/coccinelle.html`. 228

[Coe04] Claudio Sacerdoti Coen. A semi-reflexive tactic for (sub-)equational reasoning. In Jean-Christophe Filliâtre, Christine Paulin-Mohring, and Benjamin Werner, editors, *TYPES*, volume 3839 of *Lecture Notes in Computer Science*, pages 98–114. Springer, 2004. 94

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, New York, NY, USA, 1971. ACM. `http://dx.doi.org/10.1145/800157.805047`. 22

[Coq] The Coq Proof Assistant. `http://coq.inria.fr/`. 5

[Cor05] Pierre Corbineau. *Démonstration Automatique en Théorie des Types*. PhD thesis, 2005. 184

[Cor06] Pierre Corbineau. Deciding equality in the constructor theory. In Thorsten Altenkirch and Conor McBride, editors, *TYPES*, volume 4502 of *LNCS*, pages 78–92. Springer, 2006. 97

[Cou97] Judicael Courant. A module calculus for pure type systems. In R. Hindley, editor, *Proceedings fo the Third International Conference on Typed Lambda Calculus and Applications (TLCA'97)*, Nancy, France, 1997. Springer-Verlag LNCS. 94, 132

[CP90] T. Coquand and C. Paulin. Inductively defined types. In *COLOG-88: Proceedings of the international conference on Computer logic*, pages 50–66, New York, NY, USA, 1990. Springer-Verlag New York, Inc. 87

[dB94] N. de Bruijn. Highlighting the lambda-free fragment of automath. In Thomas Melham and Juanito Camilleri, editors, *Higher Order Logic Theorem Proving and Its Applications*, volume 859 of *Lecture Notes in Computer Science*, pages 81–96. Springer Berlin / Heidelberg, 1994. 5

[Del00] David Delahaye. A Tactic Language for the System Coq. In *Proceedings of Logic for Programming and Automated Reasoning (LPAR), Reunion Island (France)*, volume 1955 of *LNCS/LNAI*, pages 85–95. Springer-Verlag, November 2000. 95

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962. 3, 24, 25

[dlT90] Thierry Boy de la Tour. Minimizing the number of clauses by renaming. In *CADE-10*, pages 558–572. Springer-Verlag, 1990. 155

[DM01] David Delahaye and Micaela Mayero. `Field`: une procédure de décision pour les nombres réels en Coq. In *JFLA, Pontarlier (France)*. INRIA, Janvier 2001. 101

[dMB07] Leonardo Mendonça de Moura and Nikolaj Bjørner. Efficient E-matching for SMT solvers. In *CADE*, LNCS, pages 183–198, 2007. 166

[dMB08] Leonardo de Moura and Nikolaj Bjørner. *Z3: An Efficient SMT Solver*, volume 4963/2008 of *Lecture Notes in Computer Science*, chapter 24, pages 337–340. Springer Berlin, Berlin, Heidelberg, April 2008. 17, 55

[dMRS05] Leonardo de Moura, Harald Rueß, and Natarajan Shankar. Justifying equality. *Electron. Notes Theor. Comput. Sci.*, 125(3):69–85, 2005. 47

[DNS05] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005. 55, 154, 156, 157

[Dow08] Gilles Dowek. *Les Métamorphoses du Calcul: une étonnante histoire des mathématiques.* Éditions Le Pommier, Essais, 2008. Grand prix de philosophie de l'Académie Française. 100

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960. 3, 23

[DST80] Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. Variations on the common subexpression problem. *J. ACM*, 27(4):758–771, 1980. 53, 55, 62

[Dyc92] Roy Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *J. Symb. Log.*, 57(3):795–807, 1992. 97

[Dyc97] Roy Dyckhoff. Some benchmark formulae for intuitionistic propositional logic, 1997. 164

[ES04] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *Theory and Applications of Satisfiability Testing*, pages 502–518. 2004. 98

[FC06] Jean-Christophe Filliâtre and Sylvain Conchon. Type-safe modular hash-consing. In *ML '06: Proceedings of the 2006 workshop on ML*, pages 12–19, New York, NY, USA, 2006. ACM. 19, 166

[Fil03] J.-C. Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, March 2003. 16

[FL04] J.-C. Filliâtre and P. Letouzey. Functors for Proofs and Programs. In *Proceedings of The European Symposium on Programming*, pages 370–384, Barcelona, April 2004. 106

[FLL$^+$02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 2002. ACM. 16

[FM07] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *19th International Conference on Computer Aided Verification*, Lecture Notes in Computer Science, Berlin, Germany, July 2007. Springer-Verlag. 16

[Fre95] Jon William Freeman. *Improvements to propositional satisfiability search algorithms.* PhD thesis, Philadelphia, PA, USA, 1995. 31

[FS02] Jonathan Ford and Natarajan Shankar. Formal verification of a combination decision procedure. In *CADE-18: Proceedings of the 18th International Conference on Automated Deduction*, pages 347–362, London, UK, 2002. Springer-Verlag. 56

[FyMM$^+$06] Pascal Fontaine, Jean yves Marion, Stephan Merz, Leonor Prensa Nieto, Alwen Tiu, Loria Inria, and Lorraine Université Nancy. Expressiveness + automation + soundness: Towards combining smt solvers and interactive proof assistants. In *In Tools and Algorithms for Construction and Analysis of Systems (TACAS*, pages 167–181. Springer-Verlag, 2006. 244

[G. 62] G. M. Adel'son-Vel'skii, Y. M. Landis. An algorithm for the organization of information. *Doklady Akademii Nauk SSSR*, 1962. 122

[GB08] Yeting Ge and Clark Barrett. Proof translation and smt-lib benchmark certification: A preliminary report. In *In 6th International Workshop on Satisfiability Modulo Theories*, 2008. 244

[Gim96] Carlos Eduardo Giménez. *Un Calcul De Constructions Infinies Et Son Application A La Verification De Systemes Communicants.* PhD thesis, 1996. 87

[GL02] Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In *ICFP*, pages 235–246, 2002. 93

[GM93] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: a theorem proving environment for higher order logic.* Cambridge University Press, New York, NY, USA, 1993. 244

[GM05] Benjamin Grégoire and Assia Mahboubi. Proving equalities in a commutative ring done right in Coq. In *TPHOLs*, pages 98–113, 2005. 101, 218

[GM08] Georges Gonthier and Assia Mahboubi. A Small Scale Reflection Extension for the Coq system. Research Report RR-6455, INRIA, 2008. 113, 228

[Har95] John Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, Millers Yard, Cambridge, UK, 1995. Available on the Web as `http://www.cl.cam.ac.uk/~jrh13/papers/reflect.dvi.gz+`. 6

[HCF$^+$07] Clément Hurlin, Amine Chaib, Pascal Fontaine, Stephan Merz, and Tjark Weber. Practical Proof Reconstruction for First-order Logic and Set-Theoretical Constructions. In Moa Johansson Lucas Dixon, editor, *The Isabelle Workshop 2007 - Isabelle'07 The 21st Conference on Automated Deduction - CADE-21*, pages 2–13, Bremen Allemagne, 2007. URL : http://homepages.inf.ed.ac.uk/ldixon/events/isabelle-ws-07/isabelle-07.pdf. 244

[HM07] Thierry Hubert and Claude Marché. Separation analysis for deductive verification. In *Heap Analysis and Verification (HAV'07)*, pages 81–93, Braga, Portugal, March 2007. `http://www.lri.fr/~marche/hubert07hav.pdf`. 18

[hol] The HOL system. `http://www.cl.cam.ac.uk/research/hvg/HOL/`. 5

[Hua07] Jinbo Huang. The effect of restarts on the efficiency of clause learning. In *IJCAI'07: Proceedings of the 20th international joint conference on Artifical intelligence*, pages 2318–2323, San Francisco, CA, USA, 2007. Morgan Kaufmann Publishers Inc. 49

[Isa] Isabelle. `http://isabelle.in.tum.de/index.html`. 5, 244

[KW10] Chantal Keller and Benjamin Werner. Importing HOL Light into Coq. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK,* *July 11-14, 2010. Proceedings*, volume 6172 of *Lecture Notes in Computer Science*, pages 307–322, Edimbourg Royaume-Uni, 2010. Springer. 245

[Leg] The lego proof assistant. `http://www.dcs.ed.ac.uk/home/lego`. 5

[Ler09a] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009. 228

[Ler09b] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009. 228

[Let03] Pierre Letouzey. A New Extraction for Coq. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002*, volume 2646 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003. 94

[Let08] P. Letouzey. Coq Extraction, an Overview. In A. Beckmann, C. Dimitracopoulos, and B. Loewe, editors, *Logic and Theory of Algorithms, Fourth Conference on Computability in Europe, CiE 2008*, volume 5028 of *Lecture Notes in Computer Science*. Springer-Verlag, 2008. 94

[LGvH$^+$79] David C. Luckham, Steven M. German, Friedrich W. von Henke, Richard A. Karp, P. W. Milne, Derek C. Oppen, Wolfgang Polak, and William L. Scherlis. Stanford pascal verifier user manual. Technical report, Stanford, CA, USA, 1979. 55

[LT00] Pierre Letouzey and Laurent Théry. Formalizing Stålmarck's algorithm in Coq. In Mark Aagaard and John Harrison, editors, *TPHOLs*, volume 1869 of *Lecture Notes in Computer Science*, pages 387–404. Springer, 2000. 228

[MB07] Leonardo Moura and Nikolaj Bjørner. Efficient e-matching for smt solvers. In *CADE-21: Proceedings of the 21st international conference on Automated Deduction*, pages 183–198, Berlin, Heidelberg, 2007. Springer-Verlag. 19

[MBG06] Sean Mclaughlin, Clark Barrett, and Yeting Ge. Cooperating theorem provers: A case study combining HOL-Light and CVC Lite. In *In Proc. 3 rd Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR '05), volume 144(2) of Electronic Notes in Theoretical Computer Science*, pages 43–51. Elsevier, 2006. 244

[Met] The metis prover. `http://www.gilith.com/software/metis/`. 244

[ML75] Per Martin-Löf. An intuitionistic theory of types: Predicative part. In H.E. Rose and J.C. Shepherdson, editors, *Logic Colloquium '73, Proceedings of the Logic Colloquium*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 73 – 118. Elsevier, 1975. 5, 86

[MMZ$^+$01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Annual ACM IEEE Design Automation Conference*, pages 530–535. ACM, 2001. 25, 48, 98

[Mn94] Cesar Muñoz. Démonstration automatique dans la logique propositionnelle intuitionniste. Technical report, September 1994. Master's Thesis. 97

[MQP06a] Jia Meng, Claire Quigley, and Lawrence C. Paulson. Automation for interactive proof: first prototype. *Inf. Comput.*, 204(10):1575–1596, 2006. 98

[MQP06b] Jia Meng, Claire Quigley, and Lawrence C. Paulson. Automation for interactive proof: first prototype. *Inf. Comput.*, 204(10):1575–1596, 2006. 244

[NO79] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1:245–257, 1979. 15, 53

[NO80] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27:356–364, 1980. 15, 53, 55, 81

[NO05] Robert Nieuwenhuis and Albert Oliveras. Proof-producing congruence closure. In *16th International Conference on Rewriting Techniques and Applications*, pages 453–468. Springer, 2005. 47

[NO07] Robert Nieuwenhuis and Albert Oliveras. Fast congruence closure and extensions. *Inf. Comput.*, 205(4):557–580, 2007. 81

[NOT04] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Abstract DPLL and abstract DPLL modulo theories. In *LPAR*, volume 3452, pages 36–50. Springer-Verlag, 2004. 49, 50

[NRW98] Andreas Nonnengart, Georg Rock, and Christoph Weidenbach. On generating small clause normal forms. In *CADE-15*, pages 397–411, London, UK, 1998. Springer-Verlag. 155

[NSS57] A. Newell, J. C. Shaw, and H. A. Simon. Empirical explorations of the logic theory machine: a case study in heuristic. In *IRE-AIEE-ACM '57 (Western): Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability*, pages 218–230, New York, NY, USA, 1957. ACM. 3

[NuP] Nuprl. `http://nuprl.org`. 5

[Obj] The Objective Caml language. `http://caml.inria.fr/`. 19

[OG98] Chris Okasaki and Andy Gill. Fast mergeable integer maps. In *ACM SIGPLAN Workshop on ML*, pages 77–86, September 1998. 128

[PFF] Pff: Preuves formelles sur les flottants. `http://lipforge.ens-lyon.fr/projects/pff/`. 228

[PG86] David A. Plaisted and Steven Greenbaum. A structure-preserving clause form translation. *J. Symb. Comput.*, 2(3):293–304, 1986. 155

[Pie02] Benjamin C. Pierce. *Types and programming languages.* MIT Press, Cambridge, MA, USA, 2002. 17

[PM89a] Christine Paulin-Mohring. Extracting F_ω programs from proofs in the Calculus of Constructions. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, Austin, January 1989. ACM. 94

[PM89b] Christine Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions.* Thèse d'université, Paris 7, January 1989. 94

[PM93] Christine Paulin-Mohring. Inductive definitions in the system coq - rules and properties. In *TLCA '93: Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 328–345, London, UK, 1993. Springer-Verlag. 87

[Pre29] Mojzesz Presburger. *Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt.* Warsaw, Poland, 1929. 14

[PS99] Frank Pfenning and Carsten Schurmann. System description: Twelf — a meta-logical framework for deductive systems. In *Proceedings of the 16th International Conference on Automated Deduction (CADE-16*, pages 202–206. Springer-Verlag LNAI, 1999. 5

[PSS02] Francis Jeffry Pelletier, Geoff Sutcliffe, and Christian B. Suttner. The development of casc. *AI Commun.*, 15(2-3):79–90, 2002. 244

[Pug92] William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:4–13, 1992. 97

[PVS] Pvs specification and verification system. `http://pvs.csl.sri.com/`. 6, 56

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965. 3

[RRT04] Silvio Ranise, Christophe Ringeissen, and Duc K. Tran. Nelson-oppen, shostak and the extended canonizer: A family picture with a newborn. In *ICTAC*, pages 372–386, 2004. 82

[RRT07] Silvio Ranise, Christophe Ringeissen, and Duc-Khanh Tran. Combining proof-producing decision procedures. In *FroCoS '07: Proceedings of the 6th international symposium on Frontiers of Combining Systems*, pages 237–251, Berlin, Heidelberg, 2007. Springer-Verlag. 47

[RS01] Harald Rueß and Natarajan Shankar. Deconstructing shostak. In *LICS '01: Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, page 19, Washington, DC, USA, 2001. IEEE Computer Society. 56, 58

[RW69] J. A. Robinson and L. T. Wos. Paramodulation and first-order theorem proving. *Machine Intelligence 4*, pages 135 – 150, 1969. 3

[sat] The international SAT Competitions web page. `http://www.satcompetition.org/`. 47

[Sch02] S. Schulz. E – A Brainiac Theorem Prover. *Journal of AI Communications*, 15(2/3):111–126, 2002. 244

[Sho78] Robert E. Shostak. An algorithm for reasoning about equality. *Commun. ACM*, 21(7):583–585, 1978. 15, 53, 55

[Sho79] Robert E. Shostak. A practical decision procedure for arithmetic with function symbols. *J. ACM*, 26(2):351–360, 1979. 15

[Sho84] Robert E. Shostak. Deciding combinations of theories. *J. ACM*, 31(1):1–12, 1984. 15, 55, 82

[SL95] Alexander Stepanov and Meng Lee. The standard template library. Technical report, WG21/N0482, ISO Programming Language C++ Project, 1995. 106

[SMT] Satisfiability modulo theories competition (smt–comp). `http://www.smtcomp.org`. 15

[SO08] Matthieu Sozeau and Nicolas Oury. First-Class Type Classes. In César Muñoz Otmane Ait Mohamed and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics*, pages 278–293, 2008. 106

[Soz09] Matthieu Sozeau. A New Look at Generalized Rewriting in Type Theory. *Journal of Formalized Reasoning*, 2(1):41–62, December 2009. 94, 212

[SS96] Joao P. Marques Silva and Karem A. Sakallah. Grasp: a new search algorithm for satisfiability. In *ICCAD*, pages 220–227. IEEE Computer Society, 1996. 31, 49

[Tar46] Alfred Tarski. *Introduction to Logic and to the Methodology of Deductive Sciences.* Oxford University Press, second edition, 1946. 14

[TH96] Cesare Tinelli and Mehdi Harandi. A new correctness proof of the nelson-oppen combination procedure. In *Frontiers of Combining Systems, volume 3 of Applied Logic Series*, pages 103–120. Kluwer Academic Publishers, 1996. 54, 55

[Tin02] Cesare Tinelli. A DPLL-based calculus for ground satisfiability modulo theories. In Giovambattista Ianni and Sergio Flesca, editors, *Proceedings of the 8th European Conference on Logics in Artificial Intelligence (Cosenza, Italy)*, volume 2424 of *Lecture Notes in Artificial Intelligence*, pages 308–319. Springer, 2002. 26, 49

[TKN07] Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. *SIGPLAN Not.*, 42(1):97–108, 2007. 18

[Tse68] G. S. Tseitin. On the complexity of derivations in the propositional calculus. *Studies in Mathematics and Mathematical Logic*, Part II:115–125, 1968. 155

[Typ] The TypiCal project. `http://www.lix.polytechnique.fr/typical`. 86

[TZ03] Cesare Tinelli and Calogero G. Zarba. Combining non-stably infinite theories. *Journal of Automated Reasoning*, 34, 2003. 82

[WA09a] Tjark Weber, , and Hasan Amjad. Efficiently Checking Propositional Refutations in HOL Theorem Provers. In *Journal of Applied Logic*, volume 7, pages 26–40, 2009. 98

[WA09b] Tjark Weber and Hasan Amjad. Efficiently checking propositional refutations in HOL theorem provers. *Journal of Applied Logic*, 7(1):26–40, March 2009. 244, 245

[Wan60] Hao Wang. Toward mechanical mathematics. *IBM J. Res. Dev.*, 4(1):2–22, 1960. 3

[WB89] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL '89*, pages 60–76, New York, NY, USA, 1989. ACM. 106

[WDF$^+$09] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. Spass version 3.5. In Renate Schmidt, editor, *Automated Deduction – CADE-22*, volume 5663 of *Lecture Notes in Computer Science*, pages 140–145. Springer Berlin / Heidelberg, 2009. 244

[WM06] Stefan Wehr and Manuel. Ml modules and haskell type classes: A constructive comparison. *Journal for Functional Programming*, 2006. 106, 129

[Yic] The Yices SMT Solver. `http://yices.csl.sri.com`. 17, 55

[Zha97] Hantao Zhang. Sato: an efficient propositional prover. In *In Proceedings of the International Conference on Automated Deduction*, pages 272–275. Springer-Verlag, 1997. 48

[ZM02] Lintao Zhang and Sharad Malik. The quest for efficient boolean satisfiability solvers. In *CADE-18*, pages 295–313. Springer-Verlag, 2002. 31

[ZMM01] Lintao Zhang, Conor F. Madigan, and Matthew H. Moskewicz. Efficient conflict driven learning in a boolean satisfiability solver. In *In ICCAD*, pages 279–285, 2001. 49

APPENDIX A

Correctness of Conflict-Driven Clause Learning

In this appendix, we prove the correctness of the inference system from Figure 2.6 page 42, *i.e.* a DPLL procedure optimized with non-chronological bracktracking and conflict-driven clause learning. We have seen in Section 2.2.3 that the completeness is straightforward because that system “subsumes” the inference system with backjumping (see Figure 2.3 page 32), which has been prove complete in Theorem 2.2.9, and we are therefore left with proving the soundness.

Similarly to the soundness proof of the system with backjumping, we will require our sequents to be well-annotated, *i.e.* that all literals appearing in dependencies are decision literals.

Definition A.0.1 (Decision literals)**.** *Let* $\Gamma \vdash \Delta : \mathcal{A}, \mathbb{A}$ *a sequent. A literal* l *is a* decision literal *if and only if* $l[l] \in \Gamma$*.*

Definition A.0.2 (Well-annotated sequents)**.** *Let* $\Gamma \vdash \Delta : \mathcal{A}, \mathbb{A}$ *a sequent. It is said to be well-annotated if the two following condtions hold:*

(i) $\forall l[\mathcal{B}] \in \Gamma, \forall k \in \mathcal{B}$*,* k *is a decision literal;*

(ii) $\forall C[\mathcal{B}] \in \Delta, \forall k \in \mathcal{B}$*,* k *is a decision literal.*

For the correctness proofs of the basic DPLL procedure, we proceeded by proving a local invariant of soundness on the system: Theorem 2.1.5 page 2.1.5 tells that any time $\Gamma \vdash \Delta$ is derivable, then Γ and Δ is incompatible, and the soundness of the system *per se* is just the special case where Γ is empty. Similarly, in our soundness proof of the system with backjumping, we proceeded by proving a local *stability* lemma (see page 36) which tells that any derivation of $\Gamma \vdash \Delta : \mathcal{A}$ can be pruned into a derivation of $\Gamma_{|\mathcal{A}} \vdash \Delta_{|\mathcal{A}}$. We claim that these lemmas are local properties of the respective derivation systems because they do not depend on whether they are part of a larger derivation or not, and are essentially expressed without context. Unfortunately, the system with clause learning cannot be proved by such methods because when learnt clauses are explicitely added to the problem, this is only justified by the fact that these clauses are consequences of the original problem. Therefore, in order to convey their correctness, we need to refer to the problem at the root of the tree. To that end, we define a notion of subsumption between sequents and formulae.

Definition A.0.3 (Subsumption)**.** *Let* Φ *a formula in conjunctive normal form, and* $\Gamma \vdash \Delta : \mathcal{A}, \mathbb{A}$ *a sequent. We say that this sequent* subsumes *the formula* Φ *if the two following conditions hold:*

(i) $\forall l[\mathcal{B}] \in \Gamma, \forall \mathcal{M}, \mathcal{B} \subseteq \mathcal{M} \implies \mathcal{M} \models \Phi \implies \mathcal{M}(l) = \top$*;*

(ii) $\forall C[\mathcal{B}] \in \Delta, \forall \mathcal{M}, \mathcal{B} \subseteq \mathcal{M} \implies \mathcal{M} \models \Phi \implies \mathcal{M} \models C$*.*

In other words, subsumption means that if a literal (resp. a clause) is annotated with some dependencies $\mathcal{B}$ in the sequent, then any model of Φ which extends that set of dependencies $\mathcal{B}$ is a model of the literal (resp. the clause).

We are now ready to prove the main lemma of our soundness proof, it establishes an invariant of a derivable sequent: it states that if a well-annotated derivable sequent subsumes a formula Φ, the clauses stored in the right-hand side of the sequent (*i.e.* the conflict clause and the learnt clauses) are consequences of Φ.

Lemma A.0.4 (Soudness of learnt clauses)**.** *Let* Φ *a formula in conjunctive normal form, and* $\Gamma \vdash \Delta : \mathcal{A}, \mathbb{A}$ *a well-annotated, derivable sequent subsuming* Φ*. Then,*

(i) $\forall \mathcal{M}, \mathcal{A} \subseteq \mathcal{M} \implies \mathcal{M} \not\models \Phi$*;*

(i') $\forall l \in \mathcal{A}$*, l is a decision literal;*

(ii) $\forall C[\mathcal{B}] \in \mathbb{A}\ , \forall \mathcal{M}, \mathcal{B} \subseteq \mathcal{M} \implies \mathcal{M} \models \Phi \implies \mathcal{M} \models C$*;*

(ii') $\forall C[\mathcal{B}] \in \mathbb{A}\ , \forall l \in \mathcal{B}$*, l is a decision literal.*

Proof. We have four different assertions to prove, we are actually only interested in (i) and (i'), which express that the conflict set is well-annotated and is a consequence of Φ; (ii) and (ii') are required in the proof to ensure that all the sequents in induction hypotheses remain well-annotated and subsume Φ; thus they must be proven in parallel.

The proof proceeds by structural induction on the derivation of the sequent, and by case analysis on the first rule applied.

(Conflict)

$$\textsc{Conflict}\ \frac{}{\Gamma \vdash \Delta, \emptyset[\mathcal{A}] : \mathcal{A}, \emptyset}$$

(ii) and (ii') are obvious since $\mathbb{A}$ is empty. (i') is true because the sequent is well-annotated. We are left to prove (i), *i.e.* that no model of Φ extends $\mathcal{A}$. Since the sequent subsumes Φ, every model of Φ extending $\mathcal{A}$ is a model of the empty clause, and therefore there is no such model.

(Elim)

$$\textsc{Elim}\ \frac{\Gamma, l[\mathcal{B}] \vdash \Delta : \mathcal{A}, \mathbb{A}}{\Gamma, l[\mathcal{B}] \vdash \Delta, l \vee C[\mathcal{C}] : \mathcal{A}, \mathbb{A}}$$

First of all, we can apply the induction hypothesis (IH) to the premise. Indeed, it is straightforward to check that the premise is well-annotated (its dependencies are included in the dependencies of the conclusion) and also subsumes Φ (the partial model does not change, and one clause is removed). Now, because the conflict set and the learnt clauses are the same in the premise and the conclusion, all assertions are proved by IH.

(Red)

$$\textsc{Red}\ \frac{\Gamma, l[\mathcal{B}] \vdash \Delta, C[\mathcal{B} \cup \mathcal{C}] : \mathcal{A}, \mathbb{A}}{\Gamma, l[\mathcal{B}] \vdash \Delta, \bar{l} \vee C[\mathcal{C}] : \mathcal{A}, \mathbb{A}}$$

Again, we can apply the induction hypothesis (IH) to the premise. Indeed, it is straightforward to check that the premise is well-annotated (its dependencies are included in the dependencies of the conclusion) and also subsumes Φ: if a model $\mathcal{M}$ of Φ extends $\mathcal{B} \cup \mathcal{C}$, it extends both $\mathcal{B}$ and $\mathcal{C}$ and since the conclusion subsumes Φ, we know that $\mathcal{M}$ models l and $\bar{l} \vee C$, and thus $\mathcal{M} \models C$. Because the conflict set and the learnt clauses are the same in the premise and the conclusion, all assertions are proved by IH.

(Assume)

$$\textsc{Assume}\ \frac{\Gamma, l[\mathcal{B}] \vdash \Delta : \mathcal{A}, \mathbb{A}}{\Gamma \vdash \Delta, l[\mathcal{B}] : \mathcal{A}, \mathbb{A}}$$

First note that the decision literals in the conclusion and the premise are exactly the same, because the literal l added to the premise is itself annotated with decision literals. Therefore, the premise is well-annotated and we can apply the induction hypothesis (IH) since it also subsumes Φ: if $\mathcal{M}$ models Φ and extends $\mathcal{B}$, it is a model of the singleton clause $\{l\}$, therefore $\mathcal{M}(l) = \top$. Now, (i) and (ii) are true by IH-(i) and IH-(ii) since the right-hand side of the sequent does not change, and (i') and (ii') are given by IH-(i') and IH-(ii') because the decision literals are the same below and above the bar.

(BJ)

$$\text{BJ}\ \frac{\Gamma, l[l] \vdash \Delta : \mathcal{A}, \mathbb{A}}{\Gamma \vdash \Delta : \mathcal{A}, \mathit{Shift}_l\,(\mathbb{A})}\ l \notin \mathcal{A}$$

The set of decision literals in the premise is the set of decision literals in the conclusion augmented with literal l. In particular, the premise sequent is well-annotated since the only new literal is l itself. Let us prove that it subsumes Φ: the clauses in Δ are not a problem because they are in the conclusion; the only new literal in Γ is l and since l is annotated with itself, it is clear that any model extending $\{l\}$ makes l true. Therefore, we apply the induction hypothesis (IH) to the premise. Because the conflict set does not change, (i) is given by IH-(i), and (i') is also a consequence of IH-(i') because we know that $l \notin \mathcal{A}$ and l is the only decision literal missing in the conclusion. Similarly, (ii') follows from IH-(ii') because by definition of *Shift*, l does not appear in the dependencies in $\mathit{Shift}_l(\mathbb{A})$. We are left with proving (ii): consider a clause $C[\mathcal{B}]$ in $\mathit{Shift}_l(\mathbb{A})$, and a model $\mathcal{M}$ of Φ extending $\mathcal{B}$, there are two cases to consider:

- if $C[\mathcal{B}]$ was already in $\mathbb{A}$, then by IH-(ii), we have $\mathcal{M} \models C$;
- otherwise, by definition of *Shift*, $C = \bar{l} \vee D$ for some D such that $D[l, \mathcal{B}]$ belongs to $\mathbb{A}$. If $\mathcal{M}(l) = \bot$, then $\mathcal{M}(\bar{l}) = \top$ and $\mathcal{M} \models C$, but if $\mathcal{M}(l) = \top$, then $\mathcal{M}$ extends $l, \mathcal{B}$ and by IH-(ii), $\mathcal{M} \models D$, hence $\mathcal{M} \models C$.

(Split)

$$\textsc{Split}\ \frac{\Gamma, l[l] \vdash \Delta : \mathcal{A}, \mathbb{A} \qquad \Gamma, \bar{l}[\mathcal{A} \setminus l] \vdash \Delta, \mathit{Shift}_l\,(\mathbb{A}) : \mathcal{B}, \mathbb{B}}{\Gamma \vdash \Delta : \mathcal{B}, \mathit{Shift}_l\,(\mathbb{A}) \cup \{\bar{l}[\mathcal{A} \setminus l]\} \cup \mathbb{B}}\ l \in \mathcal{A}$$

With the same arguments than in the BJ rule, we can apply the induction hypothesis (IH1) to the left premise of this rule. We prove that we can apply it to the right branch as well.

First, the decision literals in the right premise are the same as the conclusion, which are the same as in the left premise except for l. By IH1-(i'), the literals in $\mathcal{A}$ are decision literals in the left premise, therefore the literals in $\mathcal{A} \setminus l$ are decision literals in the right premise, and the partial model in the right premise is well-annotated. The clauses in Δ are also well-annotated by hypothesis because they are in the conclusion, and we are left with clauses in $\mathit{Shift}_l(\mathbb{A})$: by IH1-(ii'), the dependencies of clauses in $\mathbb{A}$ are decision literals in the left premise, and by definition of *Shift*, the dependencies in $\mathit{Shift}_l(\mathbb{A})$ are just the same, with l removed. Therefore, the right sequent is well-annotated.

In order to prove that the right premise subsumes Φ, the clauses in Δ are not a issue and are consequences of Φ by hypothesis because they are in the conclusion. For clauses in $\mathit{Shift}_l(\mathbb{A})$, we can reproduce the reasoning we made in the BJ rule: let $C[\mathcal{B}]$ a clause in $\mathit{Shift}_l(\mathbb{A})$, and $\mathcal{M}$ a model of Φ extending $\mathcal{B}$, there are two cases to consider:

- if $C[\mathcal{B}]$ was already in $\mathbb{A}$, then by IH1-(ii), we have $\mathcal{M} \models C$;
- otherwise, by definition of *Shift*, $C = \bar{l} \vee D$ for some D such that $D[l, \mathcal{B}]$ belongs to $\mathbb{A}$. If $\mathcal{M}(l) = \bot$, then $\mathcal{M}(\bar{l}) = \top$ and $\mathcal{M} \models C$, but

if $\mathcal{M}(l) = \top$, then $\mathcal{M}$ extends $l, \mathcal{B}$ and by IH1-(ii), $\mathcal{M} \models D$, hence $\mathcal{M} \models C$.

Therefore, we can apply the induction hypothesis (IH2) to the right premise and we can now prove the four assertions we need. Since the decision literals in the conclusion and the right premise are the same, (i') is given by IH2-(i'). Similarly, (ii') is a consequence of IH2-(ii') and of the fact that the right premise is well-annotated. By IH2-(i), there is no model of Φ extending $\mathcal{B}$, which proves (i). Finally, let $C[\mathcal{B}]$ be a clause stored in the right-hand side of the conclusion and $\mathcal{M}$ a model of Φ extending $\mathcal{B}$, there are three cases to consider:

- $C[\mathcal{B}] \in \mathit{Shift}_l(\mathbb{A})$: the clause appears in the right premise and because the right premise subsumes Φ, we know that $\mathcal{M}$ models C;
- $C[\mathcal{B}] = \bar{l}[\mathcal{A} \backslash l]$: by IH1-(i), there is no model of Φ extending $\mathcal{A}$ therefore if $\mathcal{M} \models \mathcal{A} \setminus l$, $\mathcal{M}(\bar{l})$ is necessarily true;
- $C[\mathcal{B}] \in \mathbb{B}$: by IH2-(ii), we know that $\mathcal{M}$ models C.

This concludes the proof. □

That lemma was all we needed in order to prove the soundness of our inference system.

Theorem A.0.5 (Soudness)**.** *Let Δ a formula in conjunctive normal form, let us annotate all clauses in Δ with an empty set of dependencies. Then, if there exists $\mathcal{A}$ and $\mathbb{A}$ such that $\emptyset \vdash \Delta : \mathcal{A}, \mathbb{A}$ is derivable, Δ is unsatisfiable.*

Proof. It is straightforward to check that $\emptyset \vdash \Delta : \mathcal{A}, \mathbb{A}$ is a well-annotated sequent which subsumes Δ. Therefore, we can apply Lemma A.0.4 with $\Phi = \Delta$ and by (i'), we know that $\mathcal{A} = \emptyset$ since it only contains decision literals and $\Gamma = \emptyset$. Hence, (i) states that there does not exist a model of Δ extending $\emptyset$, in other words that Δ is unsatisfiable. □

APPENDIX B

Comparison of DPLL Strategies in Coq

Table B.1: Benchmarks for DPLL Strategies Comparison

	V/C	**An**	**As**	**Al**	**Bn**	**Bs**	**Br**	**Cn**	**Cr**
aim-1	50/80	312	96	43	0.56	0.34	0.13	0.41	**0.10**
aim-2	50/80	31	10	3.5	0.13	0.07	0.02	0.10	**0.02**
aim-3	50/80	394	161	61	0.93	0.63	0.26	0.47	**0.13**
aim-4	50/80	103	40	11	0.08	0.05	**0.02**	0.07	**0.02**
aim-1	50/100	110	30	12	29	12	5.5	2.45	**0.66**
aim-2	50/100	42	9.4	5.1	3.6	0.67	0.28	0.58	**0.14**
aim-3	50/100	62	15	9.0	23	11	4.8	**2.6**	**2.6**
aim-4	50/100	26	6.3	3.0	0.4	0.16	0.07	0.4	**0.06**
phole 5	30/81	0.15	0.05	0.03	0.05	0.03	**0.02**	0.06	0.04
phole 6	42/133	1.8	0.51	0.20	0.37	0.20	**0.09**	0.56	0.32
phole 7	56/204	24.4	5.5	3.0	3.2	1.6	**0.73**	10	1.6
phole 8	72/297	353	74	39	27	13	**5.7**	NA	18.9
phole 9	90/415	5600	1080	710	257	125	**52**	NA	410
uuf50[1]	50/218	NA	NA	NA	5280	775	398	4500	**339**
uuf75[2]	75/325	NA	NA	NA	732	86	41.3	629	**31.9**

[1] total time for 1000 instances

[2] total time for 9 instances

In this appendix, we give a comparative test of different DPLL strategies which we have implemented and proved in Coq, using the framework described in Chapter 6. We benchmarked the strategies on several problems from the SATLIB library in DIMACS format[1].

In order to be able to test large examples in reasonable time, so that we can better compare the efficiency of different strategies, we did not test these procedures using the Coq VM but, instead, we extracted the strategies to OCaml and compiled the generated sources to native code. The figures are summarized in Table B.1: each line represent a problem or a set of problems, the V/C column gives the number of variables and clauses in the corresponding formulae, and the next columns show the timings in seconds for the various strategies.

The description of the various strategies represented in Table B.1 follows:

An: Basic DPLL with only one rule applied at each iteration. Rules are tried in the following order: Conflict, Elim, Red, Assume puis Split. This corresponds exactly to the first `proof_search` function described in Section 6.1.4.

As: Basic DPLL with an eager strategy where all the possible BCP is performed in one traversal of the problem. At each step, all the clauses which become true are eliminated, others are reduced as much as possible, and unitary clauses are Assumed on the fly, which extends the current partial assignment during BCP. When this process is finished, Split is applied.

Al: Exactly the same strategy as **As**, but using lists and lists of lists instead of sets and sets of sets to represent the right-hand side of sequents. This is exactly the strategy presented in Section 6.3.

Bn: Similar to **An** but using DPLL with backjumping. One rule is applied at each step and they are tried in the following order: Conflict, Elim, Red, Assume then BJ if possible and finally Split.

Bs: DPLL with backjumping where all the BCP is performed in one traversal of the clauses. The main difference with As and Al is that during one round of BCP, unitary clauses are not assumed on the fly but are returned separately. After each BCP, these unit literals are assumed and another round of BCP is started until there is no progress. Then, the rules BJ or Split are applied.

Br: Similar to **Bs**, but at each step, the BCP is performed only with respect to the literals assumed at the previous round, because the clauses are only reduced with respect to older literals. Similarly, when branching with Bj or Split, only the new assumed literal is used for the next round of BCP.

[1] The problems and their description/origin can be found at `http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html`

Cn: Similar to **An** and **Bn** but this time uses DPLL with backjumping and conflict-driven clause learning. At each step one rule is applied, they are tried in the following order: Conflict, Elim, Red, Assume then BJ if possible and finally Split.

Cr: Similar to **Br** insofar as it only performs BCP with respect to most recently assumed literals. Moreover, in the case of the Split rule, a special $Shift_l$ is used which filters all the clauses in which we should have added $\bar{l}$, since they would be eliminated in the right branch anyway.