Laboratoire d'Informatique de l'École Polytechnique
INRIA Saclay–Île-de-France

A Matter of Trust: Skeptical Communication Between Coq and External Provers

Chantal Keller

Thèse en vue de l'obtention du titre de
Docteur de l'École Polytechnique

soutenue le 19 juin 2013
devant le jury composé de:

Germain	Faure	Directeur
Benjamin	Grégoire	
John	Harrison	Rapporteur
Claude	Marché	
Stephan	Merz	Rapporteur
Leonardo	de Moura	Rapporteur
Benjamin	Werner	Directeur

Abstract

This thesis studies the **cooperation between the Coq proof assistant and external provers through proof witnesses**. We concentrate on two different kinds of provers that can return certificates: first, answers coming from **SAT and SMT solvers** can be checked in Coq to increase both the confidence in these solvers and Coq's automation; second, theorems established in **interactive provers based on Higher-Order Logic** can be exported to Coq and checked again, in order to offer the possibility to produce formal developments which mix these two different logical paradigms. It ended up in **two software**: SMTCoq, a bi-directional cooperation between Coq and SAT/SMT solvers, and HOLLIGHTCOQ, a tool importing HOL Light theorems into Coq.

For both tools, we took great care to define a **modular and efficient architecture**, based on three clearly separated ingredients: an **embedding** of the formalism of the external tool inside Coq which is **carefully translated** into Coq terms, a **certified checker** to establish the proofs using the certificates and a Ocaml **preprocessor** to transform proof witnesses coming from different provers into a generic certificate. This division allows that a change in the format of proof witnesses only affects the preprocessor, but no proved Coq code. Another fundamental component for efficiency and modularity is **computational reflection**, which exploits the computational power of Coq to establish generic and small proofs based on the certificates.

Résumé

Cette thèse présente une **coopération entre l'assistant de preuve Coq et certains prouveurs externes basée sur l'utilisation de traces de preuves**. Nous étudions plus particulièrement deux types de prouveurs pouvant renvoyer des certificats : d'une part, les réponses des **prouveurs SAT et SMT** peuvent être vérifiées en Coq afin d'augmenter à la fois la confiance qu'on peut leur porter et l'automatisation de Coq ; d'autre part, les théorèmes établis dans des **assistants de preuves basés sur la Logique d'Ordre Supérieur** peuvent être exportés en Coq et re-vérifiés, ce qui permet d'établir des preuves formelles mêlant ces deux paradigmes logiques. Cette étude a abouti à **deux logiciels** : SMTCoq, une coopération bi-directionnelle entre Coq et des prouveurs SAT/SMT, et HOLLIGHTCOQ, un outil important les théorèmes de HOL Light en Coq.

L'architecture de chacun de ces deux développements a été pensée de manière **modulaire et efficace**, en établissant une séparation claire entre trois composants : un **encodage** en Coq du formalisme de l'outil externe qui est ensuite **traduit avec soin** vers des termes Coq, un **vérificateur certifié** pour établir les preuves, et un **pré-processeur écrit en Ocaml** traduisant les traces venant de prouveurs différents dans le même format de certificat. Grâce à cette séparation, un changement dans le format de traces n'affecte que le pré-processeur, sans qu'il soit besoin de modifier du code ou des preuves Coq. Un autre composant essentiel pour l'efficacité et la modularité est la **réflexion calculatoire**, qui utilise les capacités de calcul de Coq pour établir des preuves à la fois courtes et génériques à partir des certificats.

Acknowledgments

I much enjoyed doing this PhD., not only for the pleasure I took to work on the subject presented in this document, but also for the numerous persons I had the chance to encounter.

I took joy in being a member of the INRIA TypiCal team hosted at LIX, which was a very nice working and discussing environment: thanks to Aloïs, Arnaud, Assia, Benjamin, Benoît, Bruno, Bruno, Clément, Cyril, Denis, Enrico, Eric, François, Germain, Gilles, Jean-Marc, Lisa, Mahfuza, Mathieu, Maxime, Pierre, Qian, Stéphane and Victor for successively propagating it and Cindy, James, Lydie, Marie-Jeanne and Valérie for making everything possible.

I wish also to thank lunch, coffee, cards, drawing and many other occasions friends: Amaury, Antoine, Florence, François, Hugo, Joanie, Jϕ, Jessica, Jonas, Mahsa and Mikaël. People sharing my office also participated in this kind environment: Cécile, Claire and Thomas.

I was always welcome by the Marelle, ProVal-Toccata and Mathematical Components teams, as well as πr^2 and Deducteam. I wish to thank in particular Michaël Armand, Benjamin Grégoire and Laurent Théry, without whom SMTCoq would not have been what it is today, Maxime Dénès, who I repeatedly asked for improvements about native-coq, Pascal Fontaine for his responsiveness, and Ali Assaf, Frédéric Besson, Guillaume Burel, Sylvain Conchon, Pierre-Emmanuel Cornilleau, Gilles Dowek, Jean-Christophe Filliâtre, François Garillot, Georges Gonthier, Hugo Herbelin, Olivier Hermant, Pierre Letouzey, Guillaume Melquiond, Andreï Paskevitch, Christine Paulin, Laurence Rideau and Pierre-Yves Strub for enlightening discussions, along with all the members of the DeCert and Corias initiatives.

More generally, I had the occasion to discuss and sometimes to publish with many interesting persons, among whom Thorsten Altenkirch, Nikolaj Bjorner, Jasmin Blanchette, Sascha Böhme, Olivier Danvy, Predrag Janicic, Delia Kesner, Marc Lasson, Marko Malicović, David Monniaux, Leonardo de Moura, Benjamin C. Pierce and Tjark Weber.

I thank John Harrison for following my progress about HOLLIGHTCOQ and accepting to distribute it with HOL Light.

I especially thank my advisors Germain Faure and Benjamin Werner, who granted me a great independence while knowing that I could always have assistance from them. They also encouraged me to present my work at various conferences all over the world and to take responsibility.

I thank John Harrison, Stephan Merz and Leonardo de Moura who kindly accepted to refer this thesis and gave insightful comments; and Benjamin Grégoire and Claude Marché as examiners.

I am also particularly grateful to Assia Mahboubi, Jean-Philippe Meline and S.J. Wray who commented earlier versions of this document.

It was a pleasure to supervise Thibaut Gauthier, for a Master internship, Gaëtan Gilbert, for a Licence internship, and Cyril Letrouit, showing interest in discovering science.

I thank the École Normale Supérieure de Lyon which financed this PhD., INRIA and the ForMath project which financed my professional travels, and the École Polytechnique, the Laboratoire d'Informatique de Polytechnique and INRIA for the work infrastructure.

Teaching at École Polytechnique was also a nice experience, brought by both the university staff and the students. I express gratitude to Albert Cohen and Stéphane Graham-Lengrand who let me take an active part in INF422 and INF551 and Stéphane Graham-Lengrand, Antoniu Pop and Sylvie Putot for their support.

The École Polytechnique is also a pleasant place to live, among the good mood of the students and their willingness to organize activities, in particular musical ones. I enjoyed pretty much playing the flute in the student symphonic orchestra, successively named SymphoniX and the Orchestre du Plateau de Saclay, and sometimes conducting it, for four years (*je sais, je vieux-chouffise*). It was also enjoyable to listen to the *midi musicaux*, the piano competitions, the contemporary music concerts, the Big Band, the brass band and the choir.

Finally, nothing would be enjoyable without Amélie, Cécile, Julien, my family, in particular my parents, my sister and my nephew, and Olivier.

Contents

Chapter 1

A matter of trust

> "There has to come a point, as Multivac becomes more complex and capable, when it can move of its accord out of our control. I may have shoved it past the point."
>
> "But if you have, how can we trust Multivac to – "
>
> "We have no choice," she said.
>
> Isaac Asimov, *The Winds of Change and Other Stories*, 1983

Like human beings, automatic devices cannot be trusted unless they have given evidence of their good faith. Unlike human beings, we have a total control on them to make them output such evidence. Hence trusting them should be easy and even automatic itself. But how do we trust "automatic trusters"?

1.1 Interactive theorem provers

To solve this chicken and egg dilemma, we first need to design an "automatic truster" whose evidence of its good faith can easily convince anyone. *Interactive theorem provers* (also called *proof assistants*) are good candidates for this:

- they are "trusters": they are software aimed at helping the definition and the verification of formal proofs;
- they are automatic: proof checking consists in verifying the well-formedness of a derivation tree, which is a mechanical task; and proof design is also partially automatized;
- they are convincing: their trusted bases are kept as small and understandable as possible.

To achieve this last assertion, the proof checker at the heart of an interactive theorem prover, also called the *kernel*, is sufficiently small so that one single person can apprehend it. Most of the time, proof assistants are written in languages with well established semantics. They run on large and complicated hardware, but by looking at the code, we can be reasonably sure that they do not deliberately exploit possible flaws of the hardware. Some of them also output proof terms, which can be checked by external tools, to enhance confidence.

To make them usable, many additions are built on top of the kernel; hence they do not belong to the trusted base. They provide for instance facilities to write terms, like standard notations, and especially to write proofs, usually called *tactics*. Tactics can be very simple, performing only one basic step of proofs, or arbitrary complicated decision procedures, as long as they produce a valid inference tree.

Formalization of a variety of challenging problems has emphasized the capacity of interactive theorem provers to handle and check large proofs. Safety-critical applications have been proved correct, like parts of the Paris Métro Line 14 control system using the B-Method. Advanced mathematical reasoning is also welcome, as enlightened by the recent formalization in Coq of the Feit-Thompson theorem about classification of finite groups. Proof assistants are also well suited for theorems involving computations, like the Four Color Theorem [Gon08] or the Kepler Conjecture, currently being studied by the Flyspeck Project[1] [HHM+10].

Despite these successes, the design and the use of interactive theorem provers stagnates at an academic level and in a very restricted community of computer scientists. The level of detail of formal proofs is far more exigent than for paper proofs and a consequent knowledge of the underlying formal system is often required to complete them.

Many interactive theorem provers coexist, implementing different logical frameworks and observing different credos. This variety brings the opportunity to find provers well-suited for different kinds of problem. However, communication between proof assistants is most of the time nonexistent and it is thus impossible to benefit from advantages coming from two of them. Besides, the lack of a unified framework loses new users when starting a formalization [Ben06].

1.2 Trusting other automatic devices

As we argued, interactive theorem provers are one possible base to build evidence of the good faith of more general automatic tools on top of them. In this setting, we usually consider two kinds of evidence.

In the *autarkic* approach, we formally establish the behavior of a tool in a given model. For instance, concerning software, this corresponds to proving that a program matches its specification. Establishing the behavior of a device is a difficult task, but when done, we are convinced once and for all of its correctness. However, if the program changes even a little, the proof must be changed accordingly.

In the *skeptical* approach [HT98], we ask for the tool to justify *a posteriori* all its actions and only formally check the justifications. This requires to check back any achievement of the device, but it is often easier to formally check justifications rather than tools themselves. In addition to require less effort, it is more robust to changes in the tool: as long as justifications remain the same, the skeptical checker is still relevant.

1.3 Automatic theorem provers

Automatic theorem provers form a particular subset of automatic devices, aimed at establishing a large variety of properties with four main goals:

1. speed: they should answer as fast as possible;

[1] The progress of this project is available at `http://code.google.com/p/flyspeck`.

2. expressivity: they should solve as many problems as possible;
3. automation: the users should have little work to do to establish the property they are interested in;
4. correctness: they should never give a wrong answer.

Obviously, the last goal clashes with the other three: to be fast, expressive and fully automatic, these theorem provers are very large software that are likely to contain bugs. Contrary to proof assistants, there is no small kernel ultimately checking the proof.

The success of automatic theorem provers relies on the rapidly expanding design of decision procedures to quickly solve hard problems (eg. Boolean satisfiability, validity of formulas in Presburger arithmetic...) and even undecidable ones (eg. non-linear integer arithmetic, termination of rewriting systems...). They can be used both as a back-end to prove the correctness of programs, or inside larger applications, for instance to find an optimal solution satisfying a set of constraints. They are widely used in particular for critical applications, which thus suffer from their lack of safety.

1.4 Motivations of this work

Interactive and automatic theorem proving are two rather separated worlds that could benefit from each other. On the one hand, proof assistants, while conserving the property of being small and safe, would require less human work. On the other hand, the use of automatic theorem provers would gain safety, ideally the same degree of confidence as interactive provers.

This collaboration would be especially interesting if we make no concession on any side. The interaction with a proof assistant should not prevent the automatic prover from being fast and expressive. It should not increase the trusted base of the proof assistant, while being expressive enough to solve many goals.

Cooperation between proof assistants would also be profitable. Different aspects of the same problem could be formalized in different interactive theorem provers – depending on the possibilities they offer, or the ease of developers for one prover or another – and then entirely checked by one single (possibly external) prover. This would also increase confidence between proof assistants, if they collaborate through a skeptical approach.

Once again, we should try to conciliate the expressivity of all the proof assistants that are working together. This may not be always possible, if their logical frameworks are incompatible; in this case, concessions may have to be made depending on the features of the proofs assistants that are really used to solve the problem we are interested in, or on the shallowness of the obtained results.

We must also be careful to keep theorem statements intelligible: their translation in the target prover must be formulated like one would have done directly in this prover.

This thesis explores the collaboration between the Coq proof assistant and other provers, with a skeptical approach.

First, we study a bi-directional cooperation with some automatic theorem provers, namely SAT and SMT solvers that decide the satisfiability of Boolean formulas, in which theory reasoning might be needed. We provide both efficient tools to formally check answers coming from SAT and SMT solvers and the possibility to enjoy the power of SAT and SMT solvers inside Coq without compromising soundness.

The second part of this work investigates the importing of proofs coming from the HOL Light interactive theorem prover into Coq. Great care has been taken both to be efficient and to obtain intelligible Coq statements from HOL Light ones, despite the differences between these two provers.

The principal motivation is to bypass the problems mentioned above.

By providing a simple way to check answers given by SAT and SMT solvers, we offer the possibility to the large community using these provers to finally trust them with great confidence, at a rather small cost. It is also a way for SAT and SMT solvers designers to find bugs in a systematic way.

On the interactive theorem provers side, we bring more automation to Coq by discharging a non trivial part of goals to SAT and SMT solvers. We expect this work to facilitate the use of Coq and to attract new users, ideally from outside the community of proof assistants.

Importing HOL Light theorems into Coq is a first step towards a cooperation between interactive theorem provers based on different paradigms. The choice of HOL Light and Coq was directed by the Flyspeck Project, whose pieces are written in HOL Light and others in Coq. We hope it can be used one day to reconstruct the entire proof of Kepler conjecture.

We favor this direction since the logic of Coq augmented with classical axioms is expressive enough to encode Higher-Order Logic, whereas the other direction would have required an encoding of the Calculus of Inductive Constructions into Higher-Order Logic which would have had a difficulty which is not required.

All these aspects share the idea that it is crucial to be able to trust automatic tools. We offer a basis to solve this question.

To achieve our purpose, we have to take into account that certificates coming from SAT and SMT solvers and HOL Light are very large objects. We need to process them efficiently, both in terms of speed and memory.

All this work crucially relies on the computational power of Coq, coming from its ability to efficiently normalize λ-terms. Certificates coming from external tools are checked using certified programs in a process called *computational reflection*.

The interaction with SAT and SMT solvers also fundamentally used the recent native version of Coq [BDG11], with efficient computation and data structures [AGST10].

We cared about modularity in many aspects, to make this work as reusable as possible. We offer the possibility to handle new SAT and SMT solvers by writing only an Ocaml preprocessor, but no Coq code nor proofs. We also provide a simple interface to enhance the expressivity of this work with new theory decision procedures. Finally, concerning the importation of HOL Light into Coq, we defined in Coq a model of HOL which is generic for any HOL-like prover and does not depend on the format of certificates.

In all of this work, Coq is at the same time a tool collaborating with others *and* the skeptical checker. Another approach for such a collaboration consists in using a dedicated back-end checker, like dedukti [BCH12], instead of one of the participants. The trusted base is likely to be smaller – a dedicated checker is simpler than a general purpose proof assistant

– but at the cost of more numerous and complicated encodings, especially if the back-end checker is minimalist.

To conclude, this thesis does not directly formalize new results, but formalizes tools that will build new theorems. It proposes the theoretical foundations for an interaction between Coq and external provers, in particular a robust, efficient and modular architecture; and an implementation of this architecture.

1.5 Achievements

The work realized during this thesis followed two main directions:

- a skeptical cooperation between Coq and other provers through proof witnesses relying on computational reflection, which ended up in two softwares: SMTCoq, a bi-directional cooperation between Coq and SAT/SMT solvers, and HOLLIGHTCOQ, a tool importing HOL Light theorems into Coq;
- a reflection about the theory of parametricity inside the Calculus of Inductive Constructions, the language implemented by Coq.

1.5.1 SMTCoq

The heart of SMTCoq is a modular and efficient checker for SAT and SMT proof witnesses written and proved correct in Coq. On top of it, we provide sets of checkers and tactics to work with two provers: the SAT solver ZChaff and the SMT solver veriT. The checkers provide a certified way of checking ZChaff and veriT answers, thus increasing their safety. As a complement, the tactics bring ZChaff and veriT power inside Coq, by calling them on some goals and checking back their answers.

We were really careful to be modular, in such a way that it should be easy to work with other provers than ZChaff and veriT and to check other theories than the one we provide (currently, congruence closure and linear integer arithmetic). A prototype checker for Z3 strengthens this modularity.

We were also motivated by efficiency. Experiments show that both checkers and tactics are really efficient and bypass the state-of-the-art [Web08, BW10, LC09].

SAT and SMT in Coq can also serve as a basis for more complex decision procedures. We explain how SMTCoq can be used to implement a decision procedure for machine integers, which relies on a translation into SAT (called bit-blasting) and a call to ZChaff.

SMTCoq was developed in collaboration with some members of the ANR DeCert initiative, especially Michaël Armand and Benjamin Grégoire, with ideas by Laurent Théry.

This work was published in two peer-reviewed proceedings: the proceedings of the First International Conference on Certified Programs and Proofs (CPP'11) [AFG$^+$11a] and the proceedings of the International Workshop on Proof-Search in Axiomatic Theories and Type Theories (PSATTT'11) [AFG$^+$11b].

1.5.2 HOLLIGHTCOQ

HOLLIGHTCOQ relies on an encoding of Higher-Order Logic in Coq and its translation into Coq terms. On top of it, we provide a checker for HOL Light proof certificates in the Proof recording format [OS06] that generates Coq files containing encoded versions of HOL Light theorems and their proofs. It is then sufficient to apply the translation to obtain theorems stated and proved in Coq.

We took great care that the theorems, while generated automatically, have Coq intelligible statements. This is realized by a judicious translation of HOL Light constants into commonly used Coq objects.

The model of Higher-Order Logic in Coq is generic, but the choice of the proof certificates greatly influences the design and the efficiency of the checker. HOLLIGHTCOQ enlightens the need to find a balance between the proximity of the certificates with respect to the model and the efficiency: a checker for too distant certificates will be hard to prove correct, but a checker for simple certificates is likely to be inefficient.

This work was published in the peer-reviewed proceedings of the First International Conference on Interactive Theorem Proving (ITP'11) [KW10].

1.5.3 Parametricity

Parametricity is a branch of the theory of programming languages, informally relying on the observation that parametric programs behave uniformly with respect to their arguments. It has been studied for a large class of type systems, from system F [Rey83] to the recent extension to most of the Pure Type Systems [BJP10]. Among its most popular applications, one can cite the possibility to obtain "theorems for free" [Wad89] just by looking at the type of some terms (and not the terms themselves), or safe program transformation in the Haskell programming language [GLJ93].

We studied how this theory can be transposed to the Calculus of Inductive Constructions, the language implemented by Coq, in the presence of an impredicative sort for propositions. We proposed a slight variant of the Calculus of Inductive Constructions in which most terms are parametric. This variant is a large subset of the Calculus of Inductive Constructions characterized by a different sort hierarchy, which syntactically separates informative terms (for which parametricity makes sense) from non informative terms (for which parametricity is meaningless).

We argue that parametricity in Coq has various useful consequences. First, it gives metatheoretical results, like the independence of some formulas with the Calculus of Inductive Constructions; in particular we give an original proof of the independence of excluded middle. Second, "theorems for free" have practical applications in a proof assistant like Coq; we give the example of general results that can be obtained by parametricity in the finite group theory, a branch of mathematics widely formalize in Coq [GMR$^+$07].

These results can be implemented in Coq: even if parametricity is a metatheoretical result, theorems for free can be obtained for closed terms by reflection.

This work is a collaboration with Marc Lasson. It was published in the peer-reviewed proceedings of the 21st Annual Conference on Computer Science Logic (CSL'12) [KL12].

For a matter of coherence, this work is not detailed in this thesis. Indeed, while one of its long-term applications is to add automation to Coq via free theorems, the mechanisms involved and the background are rather far from the ones of SMTCoq and HOLLIGHTCOQ. We preferred to produce a coherent, self-contained document about skeptical cooperation between theorem provers; that is why we choose not to detail this aspect of this PhD. thesis work here.

1.6 Organization of this document

Part I presents the Coq interactive theorem prover. We especially detail one of its fundamental features: internalizing computation inside proofs (**Chapter 2**). We also discuss different embeddings of first-order languages inside Coq (**Chapter 3**).

This part serves as a prerequisite for the remainder of the thesis. **Parts II** and **III** are independent from each other.

Part II presents SMTCoq, the bi-directional cooperation between Coq and SAT and SMT solvers. After a short introduction recalling the stakes of this cooperation, we present the SATisfiability and Satisfiability Modulo Theories problems (**Chapter 4**). We emphasize possible certificates for SAT and SMT solvers that will serve as evidence to trust them (Sections 4.1.3, 4.2.3 and 4.3). **Chapter 5** details the Coq checker for SAT and SMT certificates which is at the heart of SMTCoq. Its architecture is designed to be modular both in terms of provers and theories that can be checked (Section 5.1). On top of it, we wrote utilities to achieve our goals (**Chapter 6**): a certified checker for SAT and SMT evidence (Section 6.1) and Coq tactics to call external solvers (Section 6.2). Their performances are evaluated in **Chapter 7**. We compare in particular with the checkers for SAT and SMT proof witnesses written in Isabelle/HOL by Alwen Tiu, Tjark Weber and Sascha Böhme (Section 7.1.2) and the Ergo SMT solver written in Coq by Lescuyer *et al.* (Section 7.2.2). We finally discuss the perspectives of this work (**Chapter 8**). In particular, we developed the theoretical background required for an application of SAT solvers in Coq: a decision procedure for machine integers using bit-blasting (Section 8.2).

Part III depicts HOLLIGHTCOQ, which imports HOL Light proofs into Coq. First, a brief introduction evokes the context of the communication between interactive theorem provers. **Chapter 9** presents the paradigms shared by provers based on Higher-Order Logic. Even if they do not have proof objects, they can be instrumented to generate certificates. We compare two existing implementations: Proof recording (Section 9.2.1) and OpenTheory (Section 9.2.2). **Chapter 10** details our model of Higher-Order Logic in Coq and its careful translation into intelligible Coq terms. On top of it, we wrote a checker for Proof recording certificates, which automatically generates at the same time Coq theorems coming from HOL Light and their proofs. This checker is evaluated in **Chapter 12** on two aspects: the intelligibility of the generated theorems (Section 12.1) and the time and memory consumption of HOLLIGHTCOQ (Section 12.2). We finally discusses the perspectives of this work (**Chapter 13**), in particular the changes to be made to handle more efficient and modular certificates, like OpenTheory (Section 13.2).

Part I

Coq: an interactive theorem prover based on Type Theory

This part presents the Coq interactive theorem prover, concentrating on the aspects that will be used for the remainder of the thesis.

Coq is based on the Curry-Howard correspondence: theorem statements are types and their proofs are programs inhabiting them. The type system is the Inductive Calculus of Constructions (CIC in short), which endows Coq with a very expressive intuitionistic logic.

Computation plays an important role in this system, since terms and types are considered up to β-equivalence. A closed term is thus equal to its normal form. This allows to mix computations with proofs, and even to replace proofs with computations: this is called *computational reflection* and is very useful to efficiently build small proofs of complex theorems. An example is the proof of the four-color theorem [Gon08], for which we do not know of any proof which does not require a program enumerating hundreds of cases.

The aim of this thesis is to make Coq collaborate with external provers through proof witnesses. To achieve this goal, we intensively exploit the computational power of Coq on two different aspects.

The first one appears in the formalization of the languages used by the external provers. In Coq, we need both to be able to manipulate terms of these languages and to establish in the end theorems stated in the Coq language. The first aspect corresponds to a *deep embedding* of the languages of the external provers: we represent them in such a way as to have access to the structure of the terms (usually, using inductive data types). The second aspect is a *shallow embedding* of the languages, using directly Coq terms to represent the object language. To switch between the two embeddings, we reflect the deep terms into Coq, through computation.

We also use computational reflection to handle the proofs coming from the external provers. We use them as certificates given to a checker, which is a Coq program which guarantees the correctness of proofs.

In **Chapter 2**, we explain the particularities of Coq among other interactive theorem provers, especially about computation. We briefly recall Coq's syntax, through examples explaining how computation can be used to skeptically trust external tools, in an efficient way. Efficiency of proof checking is increased by the use of efficient data structures; efficiency of proof building is enhanced by the use of the Ssreflect plugin.

In **Chapter 3**, we present the difference between deep and shallow embeddings. We give a systematic way to switch between them in the case of the simply typed λ-calculus with prenex polymorphism, a language including those of SAT, SMT and HOL provers. It thus details the theoretical basis for the developments of **Parts II and III**.

Chapter 2

Proofs and computation in Coq

It is important to understand the main features of Coq and its particularities among other theorem provers, to fully appreciate the theoretical and implementation choices made to interface it with external provers.

2.1 Interactive theorem proving through the ages (brief overview)

The development of interactive theorem provers corresponds to a will to have small programs able to check the correctness of complex proofs.

The first ones appeared in the early 70's, when Nicolaas Govert de Bruijn imagined Automath [dB70] and Robin Milner designed Logic for Computable Functions [Mil72] (LCF in short). Both laid the foundations for the requirements of proof assistants: having a small automated proof checker to verify the correctness of complex mathematical theories. Since then, many different proof assistants have been implemented, with the credo that they should be as easy as possible to use without losing safety.

Automath and LCF also initiated the two main trends concerning the design of interactive theorem provers.

LCF is written as a library in a functional programming language. Its safety relies on an abstract data type for theorems, in such a way that objects of this type can be defined using only the few inference rules provided by the kernel. These inference rules implement the Classical Higher-Order Logic in a natural deduction style. Current LCF-like provers are HOL Light, HOL4 and Isabelle.

Automath exploits the Curry-Howard correspondence: proofs are λ-terms inhabiting propositions and the kernel is a type checker. It imagined a first notion of dependent types, that was since refined into different type theories like CIC or the Computational Type Theory, whose semantics are well understood. These logical frameworks are intentional and intuitionistic. Nowadays provers based on Type Theory include Coq, Agda, NuPRL and Matita.

The differences in the design entail differences on many aspects. We are going to emphasize the main particularities of provers based on Type Theory illustrated by examples in Coq. The particularities of LCF-like provers will be presented in **Part III**.

2.2 Presentation of Coq

The Calculus of Constructions is a dependently typed intuitionistic λ-calculus allowing to express programs, specifications and proofs within the same language. Inductive definitions offer the opportunity to declare new data types and predicates.

Coq's language, GALLINA, implements CIC. A large set of vernacular commands and tactics completes the picture by facilitating the interaction with the system, especially the design of proofs. We briefly present parts of GALLINA and how computation is handled in Coq.

2.2.1 Conversion

Computations are taken into account in the formalism by the *conversion rule*:

$$\text{(CONV)}\ \frac{\Gamma \vdash t : A \qquad \Gamma \vdash B : s}{\Gamma \vdash t : B}\ \text{if } A \equiv_c B$$

In Coq (since version 8.4), the equivalence relation $\equiv_c$ contains:

- the standard β and α-conversions;
- the ζ-conversion of local definitions (the **let** ... := ... **in** ... construction);
- the η-conversion of terms with a functional type.

Example 2.1 *These two terms are convertible:*

```
(fun (x:Type → Type) ⇒ x) (fun (z:Type) ⇒ let a := z in a)
      ≡c (fun (y:Type) ⇒ y)
```

This relation is extended when defining new inductive types (ι-conversion) and global constants (δ-conversion).

Global constants

The definition of a new constant:

```
Definition c : A := body.
```

adds a new object `c` of type `A` in the environment and extends the computational equality by `c` $\equiv_c$ `body`. The type `A` can be omitted if Coq can infer it.

Example 2.2 *The polymorphic identity function can be defined this way:*

```
Definition identity := fun (A:Type) (x:A) ⇒ x.
```

or with syntactic sugar for function definition:

```
Definition identity (A:Type) (x:A) := x.
```

These two terms are convertible:

```
(identity (Type → Type)) (identity Type) ≡c (fun (y:Type) ⇒ y)
```

If giving the λ-term using the **Definition ... := ...** syntax is appropriate to define terms with a computational content (like the identity function), it is not suited to build proofs. To achieve this, we rather use tactics that change the goal step-by-step until it is proved.

Example 2.3 *A proof of the identity theorem can be described by a functional definition:*

```
Definition id1 : forall (A:Prop), A → A := fun A H ⇒ H.
```

or using tactics (here, intros *and* exact*):*

```
Lemma id2 : forall (A:Prop), A → A.
  intros A H.
  exact H.
Qed.
```

In the latter case, we usually use the **Lemma** *keyword instead of* **Definition**.

Tactics internally build a λ-term whose type should be the lemma and **Qed** actually calls Coq type checker to validate it and finally define the constant.

Inductive definitions

Inductive definitions add new types and constructors in the global context. Objects of an inductive type can be destructed using pattern matching, which extends conversion. We present the syntax for an example.

Example 2.4 *We can define the type* F *of formulas built with* $\perp$*,* $\vee$ *and variables (represented by natural numbers):*

```
Inductive F : Set :=
| Bottom : F
| Var : nat → F
| Or : F → F → F.
```

A function counting the number of connectives in a formula is:

```
Fixpoint connectives f :=
  match f with
    | Bottom ⇒ 1
    | Var _ ⇒ 0
    | Or f1 f2 ⇒ (connectives f1) + (connectives f2) + 1
  end.
```

We can now make clear that computations are taken into account in the conversion relation. For instance, these two terms are convertible:

connectives (Or Bottom (Var 0)) $\equiv_c$ 2

The previous example can be written in general purpose functional programming languages. Coq offers more possibilities with dependent types.

Example 2.5 *We can define an inductive predicate distinguishing the ground formulas:*

```
Inductive ground : F → Prop :=
| Ground_bottom : ground Bottom
| Ground_or : forall f1 f2, ground f1 → ground f2 → ground (Or
   f1 f2).
```

Moreover, the return type of a pattern matching can differ in the different branches: this is called *dependent pattern matching*. We illustrate this possibility on our running example.

Example 2.6 *We start with defining a function deciding the predicate of the previous example (this function does not require dependent pattern matching):*

```
Fixpoint ground_dec f : Prop :=
  match f with
    | Bottom ⇒ True
    | Var _ ⇒ False
    | Or f1 f2 ⇒ (ground_dec f1) ∧ (ground_dec f2)
  end.
```

We can prove the correctness of this function using dependent pattern matching:

```
Fixpoint ground_dec_correct f : ground_dec f → ground f :=
  match f return ground_dec f → ground f with
    | Bottom ⇒ fun _ ⇒ Ground_bottom
    | Var _ ⇒ fun h ⇒ match h with end
    | Or f1 f2 ⇒ fun h ⇒
      match h with
        | conj h1 h2 ⇒ Ground_or f1 f2 (ground_dec_correct f1
           h1) (ground_dec_correct f2 h2)
      end
  end.
```

where conj *is the constructor of the* ∧ *connective. The* **return** *statement makes it clear that the return type of the function depends on the object that is matched.*

The return type of a pattern matching may also depend on the parameters and arguments of the inductive type, as illustrated by the following example.

Example 2.7 *We can prove the completeness of the* ground_dec *function this way:*

```
Fixpoint ground_dec_complete f (h : ground f) : ground_dec f :=
  match h in ground f return ground_dec f with
    | Ground_bottom ⇒ I
    | Ground_or f1 f2 h1 h2 ⇒ conj (ground_dec_complete f1 h1)
       (ground_dec_complete f2 h2)
  end.}
```

The **in** *and* **return** *statements make it clear that the return type of the function depends on the type of the object that is matched.*

For soundness reasons, inductive definitions and pattern matching must be restricted, but we are not going to detail it here (see Section 4.5 of the Coq's manual [Tea11] for details).

2.2.2 Computational reflection

A direct consequence of the conversion rule is that computation does not appear in proofs: two convertible propositions have exactly the same proofs. For instance, in **Example 2.5**, `even (connectives (Or Bottom (Var 0)))` and `even 2` have the same proofs.

The length of proofs can thus be reduced by maximizing the computational part: this principle is called *computational reflection*. Reducing the length of proofs is crucial for large developments, since proofs are stored in memory.

The idea of computational reflection, as proposed in [ACHA90], is to turn the proof search that is traditionally performed by tactics into an internal computation, performed by programs that evaluate within the logic of Coq.

To give a concrete example, let us explain how Coq manages to prove ring equalities automatically. Consider the equality $(x + y) - x = y$ where x and y are two variables that run over $\mathbb{Z}$. A dedicated data-structure in Coq represents ring expressions. It is composed of the operators `mult` for multiplication, `plus` for addition, `minus` for subtraction and `var` for variables, indexed by integers. Associated with the data-structure, there are two functions that are computable inside the logic. First, the interpretation function $[\bullet]_\rho$ that maps the abstract datastructure to a concrete domain, where ρ explains the mapping of the variables. So we have for example:

$$[\texttt{minus (add (var 0) (var 1)) (var 0)}]_{\{0\to x,1\to y\}} \equiv_c (x+y)-x$$

Second, the normalization function `normalize` computes a normal form. So, we have:

$$\texttt{normalize (minus (add (var 0) (var 1)) (var 0))} \equiv_c \texttt{var 1}$$

Its correctness lemma `normalize_correct` states that terms with equal normalizations have equal interpretations:

$$\textbf{forall}\ \rho,\ \texttt{e}_1,\ \texttt{e}_2,\ \texttt{normalize e}_1\ \texttt{=}\ \texttt{normalize e}_2 \to [\texttt{e}_1]_\rho\ \texttt{=}\ [\texttt{e}_2]_\rho$$

With these two functions and the correctness lemma, it is possible to give directly the proof of our initial equality:

```
normalize_correct {0 → x,1 → y}
     (minus (add (var 0) (var 1)) (var 0))
     (refl_equal (normalize (var 1)))
```

where `refl_equal` corresponds to the reflexivity of equality. It is then the task of the proof checker of Coq to verify that this proof term is valid. We notice that the length of this proof is linear in sum of the lengths of the terms that are normalized.

In this example, like in most applications using computational reflection, the computation inspects the **structure** of the considered terms (here, `plus`, `var`, ...) and interprets them into Coq terms (here, $[\bullet]_\rho$). We will explore in details in **Chapter 3** how to define and compute these structure and interpretation, for the languages of terms used in **Parts II and III**.

2.2.3 Propositions versus Booleans

In Coq, propositions are objects of type **`Prop`**, the dedicated sort of propositions. For instance, `ground f` of **Example 2.5** and `ground_dec f` of **Example 2.6** are propositions.

If propositions are the standard way of stating properties and theorems in Coq, they are not well suited for computation. If we come back to **Example 2.6**, we have that

```
ground_dec (Or Bottom Bottom) ≡c conj True True
```

but

```
conj True True ≢c True
```

because `conj: Prop → Prop → Prop` cannot be defined as a function that would inspect the truth of its arguments. It implies that given a closed term of type **Prop**, we do not have the property that it is convertible to either `True` or `False`.

If we are interested in computing with formulas, we need to switch to the set of Booleans, defined in Coq as an inductive data-types:

```
Inductive bool : Set :=
| true : bool
| false : bool.
```

which do have a computational content: it enjoys the property that any closed term of type `bool` is convertible to either `true` or `false`.

Example 2.8 *The function deciding if a formula is ground of* **Example 2.6** *can return a Boolean instead of a proposition:*

```
Fixpoint ground_dec_bool f : bool :=
  match f with
    | Bottom ⇒ true
    | Var _ ⇒ false
    | Or f1 f2 ⇒ andb (ground_dec_bool f1) (ground_dec_bool f2)
  end.
```

where `andb` *is the function computing the conjunction of two Booleans.*
We have that:

```
ground_dec_bool (Or Bottom Bottom) ≡c true
```

This data structure with a computational content is really suited to write **decision procedures** (like `ground_dec_bool`), which can in particular be used in computational reflection (whereas functions returning propositions cannot).

Example 2.9 *Like* `ground_dec`, `ground_dec_bool` *can be proved correct:*

```
Definition ground_dec_bool_correct f : ground_dec_bool f = true
   → ground f := ...
```

With this lemma, for any Coq closed term `f : F`, *a reflexive proof of* `ground f` *is:*

```
ground_dec_bool_correct f (refl_equal true)
```

In **Part II**, we use a Boolean decision procedure to reflexively check certificates coming from SAT and SMT solvers.

The Ssreflect plugin [GM08] gives (among other features) a small scale reflection from `bool` to `prop`, which is useful to transparently manipulate Booleans like propositions.

2.3 About efficiency

Computational reflection makes proofs small, but at the cost of lots of computation. Hence computation must be efficient. Recent developments have improved Coq with both efficient reduction and efficient data structures.

2.3.1 Different computation implementations in Coq

There are three different reduction mechanisms implemented in Coq's kernel:

- the default reduction that uses a call-by-need evaluation [Bar99], that we call the *internal reduction*;
- a reduction that uses an optimized call-by-value evaluation bytecode-based virtual machine [Gré03], that we call the *VM reduction*;
- a machine-based reduction, available through the Ocaml's compiler [BDG11], that we call the *native reduction*.

Internal and VM reductions are available in the standard distribution of Coq; the native reduction is still experimental and is available in the development version of Coq.

The internal reduction is well-suited for conversion in usual Coq proofs. On the contrary, the VM and native reductions are very efficient to fully evaluate algebraic objects. This is what we need to do computational reflection. Experiments show that the native reduction is most of the time more efficient than the VM, despite the entry cost of compilation to machine code [BDG11].

2.3.2 Efficient data structures

A complementary approach to improve efficiency is to have access to low-level destructive data structures supporting primitive operations, that can be computed very quickly by processors.

A first work extended the VM reduction with machine integers [Spi06]. These are integers between 0 and $2^{31}-1$, taken modulo 2^{31}, and are directly manipulated by the machine. It gives access to a large set of constants whose manipulation is efficient. In particular, some objects can be encoded using binary integers, in such a way that operations on them correspond to bitwise operations.

The native reduction is well suited to handle imperative data structures, since we directly use Ocaml terms. A recent work [AGST10] extends Coq with machine integers and arrays for the native reduction; it is available in native-coq [Dén], a fork of the development version of Coq. The interface of arrays is persistent, as presented in [Bak91]: accessing the latest updated array is done in constant time and accessing the history of updates is done in linear time. Having a functional interface does not compromise Coq's soundness and we have the same efficiency as with destructive arrays as long as we do not want to access the history. This extension of course increases the trusting base, but at very precise places and for the benefit of efficiency (as we will evaluate on Section 7.3).

2.4 Impact on the work presented in this thesis

The particularities of Coq presented in this chapter have entailed both theoretical and implementation choices for the remaining of the thesis.

On a theoretical point of view, the external tools we interface with Coq implement extensional and classical logics, so we had to imagine an integration that fits Coq's behaviour.

All the developments are based on computational reflection. As we argued, while giving generic proof tools, it reduces memory consumption while being efficient since we use VM or native reduction and imperative data structures. **Part III** also uses the Ssreflect tactics and libraries package to ease the development, hence the comprehension of the code.

Chapter 3

Embedding logical frameworks in Coq

We argued that computational reflection required (among others) two components: a Coq data-structure representing the terms we manipulate, and a Coq interpretation function mapping these terms to their concrete Coq counterparts. In the case of an interaction with external provers, the terms are formulas belonging to the logical frameworks of these provers.

The Coq data-structure representing the terms is called a *deep embedding* of the logical framework: we define in Coq a data type representing the formulas, with the possibility to manipulate them within Coq. On the other side, the concrete Coq counterpart is called a *shallow embedding*: there is no intermediate data-structure. These two notions have been originally introduced for hardware description languages in HOL [BGG^{+}92], and are now commonly applied to any kind of embedding.

In this chapter, we explain the distinction between these two possible embeddings and how to write the interpretation function as a Coq program. We also study the reverse operation that extract the structure from a shallow Coq term, called *reification*. We illustrate this in the case of the simply-typed λ-calculus with prenex polymorphism, a language expressive enough to contain both underlying languages of SAT and SMT solvers and Higher-Order Logic.

3.1 Deep and shallow embeddings

In order to represent terms from one logical framework A inside another formalism B, we have two possible ways:

- a *deep* embedding: define data-types in B that represent types and terms of A; we can then define, inside B, what it means to be provable in A;
- a *shallow* embedding: represent types and terms of A using their counterparts in B; this translation must preserve provability.

Example 3.1 *Type* `F` *of* **Example 2.5** *is a deep embedding of propositions built with* $\bot$, $\vee$ *and variables. The proposition* $x \vee \bot$ *is represented by* `Or (Var 0) Bottom` *in this deep embedding, with the integer* 0 *representing* x*. Its shallow representation in Coq is* `x \/ False` *with* `x : Prop`.

The deep embedding gives access to the structure of the terms of the language. As we noticed in Section 2.2.2, this is very useful to use computational reflection. The main drawback is that, for each deep embedding, a user must redefine the behavior of the terms with respect to standard operations like substitution, which can be really painful.

On the contrary, the shallow embedding does not give access to the structure of terms, but offers the possibility to use some aspects of the meta-language (here, Coq), like substitution, binders. . . for free.

In the developments presented in this thesis, we want to prove Coq theorems using external tools, by computational reflection. Theorem statements are naturally formulated with Coq terms, in a shallow embedding. Computational reflection requires an access to the structure of terms, thus a deep embedding. It implies that we need to switch between the two representation. In fact, the deep embedding represents an interface between Coq and the external tools, as illustrated by Figure 3.1.

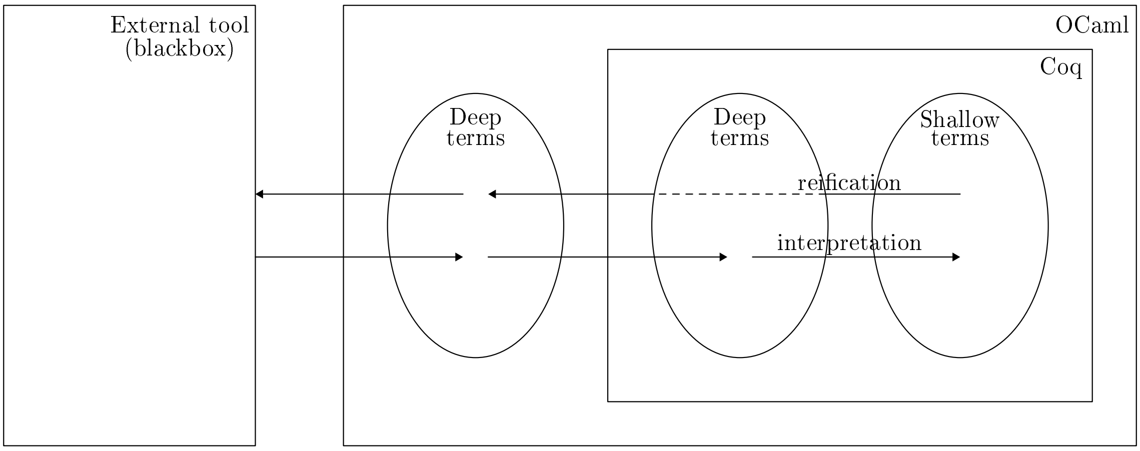


Figure 3.1: Use of deep and shallow embeddings in an interaction with external tools

Such a translation between one level of language to another is studied in particular for *normalization by evaluation* [BS91], a way of computing normal forms of an object language using the normalizer of the target language. In particular, it has been implemented for the simply typed λ-calculus (STLC in short) with named variables in Coq by Garillot and Werner [GW07].

As shown on Figure 3.1, computational reflection is performed inside Coq, so the interpretation function is written in Coq, like in [GW07]. We thus extend this work to write our own interpretation function.

However, reification can be used as an oracle without compromising soundness (see Section 6.2.2), so we can write it directly at the ML level, as suggested by Figure 3.1. Reification of Coq terms is not documented; the only reference we know explaining reification in the purpose of implementing new Coq tactics is a code-oriented tutorial by Braibant and Boutillier[1]. We will briefly complement and illustrate it for the purpose of the embedding of logical frameworks.

[1] A tutorial on how to write OCaml tactics for the Coq proof assistant, available on github https://github.com/braibant/coq-tutorial-ml-tactics.

The remaining of this chapter is dedicated to interpretation and reification. For interpretation, we focus on the simply-typed λ-calculus with prenex polymorphism, which contains the languages of SAT and SMT solvers and HOL provers. We illustrate this translation with many examples. In the last section, we present reification at the ML level on a toy example, to give a general overview of the manner to do it for Coq.

3.2 From deep to shallow: interpretation

Going from the deep to the shallow embedding consists in *interpreting*, or *compiling*, the terms of the source language into their Coq counterparts. We consider the simply typed λ-calculus with prenex polymorphism; since it is included in CIC, the semantics of the interpretation function is simply an injection.

As we argued, we want to formalize this injection as a Coq program, which is not trivial. In particular, this function must handle deep terms which have different types in STLC and thus whose interpretations have different Coq types: the return type of the interpretation function actually depends on its argument. [GW07] solves this issue for the simply-typed λ-calculus with named variables; we extend it to locally-nameless variables [ACP$^+$08] and prenex polymorphism.

3.2.1 General idea

We adopt the following syntax for the simply-typed λ-calculus with named variables:

$$\begin{array}{rcl} A, B & \triangleq & o \mid A \hookrightarrow B \\ u, v & \triangleq & x^A \mid \lambda x^A.u \mid u\ v \end{array}$$

Variables are typed, both in their use and in their abstraction. For the moment, there are no type variables nor polymorphism; we will handle this in Section 3.2.4.

Defining an interpretation function for types $[\bullet]$ respecting the equations

$$\begin{array}{rcl} [o] & = & \text{Prop} \\ [A \hookrightarrow B] & = & [A] \rightarrow [B] \end{array}$$

is straightforward. This is not the case of the interpretation function for terms $|\bullet|_{\mathcal{I}}$. Informally, it must satisfy the equations

$$\begin{array}{rcl} |x^A|_{\mathcal{I}} & = & \mathcal{I}(x^A) \\ |\lambda x^A.u|_{\mathcal{I}} & = & z \mapsto |u|_{\mathcal{I}(x^A \leftarrow z)} \\ |u\ v|_{\mathcal{I}} & = & |u|_{\mathcal{I}}(|v|_{\mathcal{I}}) \end{array}$$

where $\mathcal{I}$ is an environment interpreting the free variables, and recursively enriched when interpreting abstractions. The difficulty is to type this function: its codomain depends on its argument. More precisely, it depends on the deep type of its argument, in this way:

$$\text{if } \vdash t : A \text{ in STLC, then } \vdash |t| : [A] \text{ in CIC}$$

.

The idea of [GW07] is to use an intermediate interpretation function $|\bullet|'_{\mathcal{I}}$ that takes into account the deep types of terms: it does not only compute its interpretation, but also its deep type. It thus *refines* the typing function of STLC: it returns a dependent pair whose first component is the deep type of the argument (like the typing function) and whose second component is the actual interpretation. The true interpretation function can be easily written by returning the second component. For ill-typed terms, the function simply returns an error (using the `option` type).

The typing function and its refinement can be described by these equations:

$$\begin{array}{rcl}
\mathrm{infer}(x^A) & = & A \\
\mathrm{infer}(\lambda x^A.u) & = & A \hookrightarrow \mathrm{infer}(u) \\
\mathrm{infer}(u\ v) & = & \begin{cases} B \text{ if } \mathrm{infer}(u) = A \hookrightarrow B \text{ and } \mathrm{infer}(v) = C \text{ and } A = C \\ \text{fails otherwise} \end{cases} \\
 & & \\
|x^A|'_{\mathcal{I}} & = & (A, \mathcal{I}(x^A)) \\
|\lambda x^A.u|'_{\mathcal{I}} & = & (A \hookrightarrow U, z \mapsto i) \text{ if } |u|'_{\mathcal{I}(x^A \leftarrow z)} = (U, i) \\
|u\ v|'_{\mathcal{I}} & = & \begin{cases} (B, i\ j) \text{ if } |u|'_{\mathcal{I}} = (A \hookrightarrow B, i) \text{ and } |v|'_{\mathcal{I}} = (C, j) \text{ and } A = C \\ \text{fails otherwise} \end{cases}
\end{array} \tag{3.1}$$

which highlight the correspondence between the two.

Returning the deep type for the intermediate function is fundamental to compute the interpretation of an application: we need not only to check that the type of the domain of the function is the type of the argument, but also to coerce $j : [C]$ into $j : [A]$ in order for $i\ j$ to be well typed in CIC. [GW07] gives also an original way to perform this coercion in Coq, that we present in the next section.

3.2.2 Coq formalization

We now formalize this idea in Coq, as presented in [GW07].

Embeddings

The definition of types and terms is straightforwardly written in Coq:

```
Inductive type : Set :=
| o : type
| a : type → type → type.

Local Notation "A⟶B" := (a A B) (at level 50).

Inductive term : Set :=
| Var : nat → type → term
| Lam : nat → type → term → term
| App : term → term → term.
```

We define the syntactic Boolean equality between two types, written `A == B`.

Type inference

Type inference is transposed in Coq using the `option` type to represent failures.

```
Fixpoint infer (t:term) : option type :=
  match t with
    | Var _ A ⇒ Some A
    | Lam _ A u ⇒
      match infer u with
        | Some B ⇒ Some (A⟶B)
        | _ ⇒ None
      end
    | App u v ⇒
      match infer u, infer v with
        | Some (A⟶B), Some C ⇒
          if A == C then Some B else None
        | _, _ ⇒ None
      end
  end.
```

We define a relation stating that a term t is well-typed of type A in a context g if the inference function returns Some A:

```
Definition wt (t: term) (A: type) : Prop := infer t = Some A.
```

Type coercion

To infer the type of an application, we observed that a Boolean equality between the domain of the function and the type of its argument is sufficient. However, we explained that we need more information for the interpretation function, in particular a coercion between these two types. This informative data type can be written in Coq as:

```
Inductive cast_result (A: Type) (n m: A) : Type :=
| Cast (k: forall P, P n → P m)
| NoCast.
```

where Cast contains a coercion function (which is the elimination principle for equality) when n and m are equal. This idea was suggested by Georges Gonthier.

For Coq terms with a decidable equality, we can at the same time prove the decidability and build the coercion. This is the case for the deep types of STLC:

```
Fixpoint cast (A B: type) : cast_result type A B :=
  match A, B return cast_result type A B with
    | o, o ⇒ idcast
    | C⟶D, E⟶F ⇒
      match cast C E, cast D F with
        | Cast k1, Cast k2 ⇒
          let k P :=
            let Pb G := P (G⟶D) in let Pc G := P (E⟶G) in
              fun x ⇒ k2 Pc (k1 Pb x)
                in Cast k
        | _, _ ⇒ NoCast
      end
    | _, _ ⇒ NoCast
  end.
```

This function enjoys a property equivalent to the axiom K in this decidable context: when applied to the same objects, it returns the identity coercion.

```
  Lemma cast_same : forall A, cast A A = Cast A A (fun _ x ⇒ x).
```

Interpretation

We are now prepared to write the interpretation functions. We start with types:

```
Fixpoint interp_type (A:type) : Type :=
  match A with
    | o ⇒ Prop
    | B⟶C ⇒ (interp_type B) → (interp_type C)
  end.
```

The environments to interpret free variables are total functions mapping a pair of a variable and a type to its interpretations:

```
Definition sem_env := nat → forall A:type, interp_type A.
```

When we will interpret abstractions, the environments will be extended, as presented in equations 3.1. We thus define a function to extend them:

```
Definition extend_env (I:sem_env) x A a : sem_env :=
  fun y B ⇒
    if y == x then
      match cast A B with
        | Cast k ⇒ k a
        | _ ⇒ I y B
      end
    else I y B.
```

The intermediate interpretation function for terms expects a term and returns a dependent pair composed of a deep type `A` and a function mapping an environment to the interpretation of the term; its return type is thus `option {A:type & sem_env → interp_type A}` using the Coq notation `{ • & •}` to represent dependent pairs (its constructor is `existT`):

```
Fixpoint interp_aux (t:term) : option {A:type & sem_env →
    interp_type A} :=
  match t with
    (* Variables: we use the environment *)
    | Var x A ⇒ Some (existT A (fun I ⇒ I x A))

    (* Abstraction: we extend the environment in the recursive
       call *)
    | Lam x A u ⇒
      match interp_aux u with
        | Some (existT U c) ⇒
          Some (existT (A⟶U) (fun I z ⇒ c (extend_env I x A
             z)))
        | _ ⇒ None
      end

    (* Application: we have to apply a coercion when the types
       of the codomain and the argument match *)
    | App u v ⇒
      match interp_aux u, interp_aux v with
        | Some (existT (A⟶B) b), Some (existT C c) ⇒
          match cast C A with
            | Cast k ⇒ Some (existT B (fun I ⇒ (b I) (k (c I))))
            | _ ⇒ None
          end
        | _, _ ⇒ None
      end
  end.
```

Coercions appear to interpret applications (as we said) and to type the `extend_env` function.

We can define the true interpretation function using dependent pattern matching to return the second projection:

```
Definition interp t :=
  match interp_aux t as p return
    match p with
      | Some (existT A _) ⇒ option (sem_env → interp_type A)
      | None ⇒ option Prop
    end
    with
    | Some (existT A b) ⇒ Some b
    | None ⇒ None
  end.
```

Remark 1 *It is crucial that* `interp_aux` *has type* `option {A:type & sem_env → interp_type A}`*, rather than* `sem_env → option {A:type & interp_type A}` *which seems more natural. In the second case, it would be impossible to do the recursive call for abstractions: we have nothing to extend* `I` *with at this stage.*

Adequacy of type inference

The fact that interpretation is a refinement of type inference entails that terms are well-typed if and only if they have an interpretation:

```
Lemma infer_interp : forall t A,
  infer t = Some A ↔ exists b, interp_aux t = Some (existT _ A
     b).
```

3.2.3 Locally nameless variables

If one wants to implement standard operations on deep terms, for instance comparison or substitution, named variables are painful since terms have to be considered up to α-conversion and variable capture may happen. The locally nameless representation for variables [ACP+08] avoids these pitfalls without the negative effects of de Bruijn indices. The key idea is to name free variables, but to represent bound variables by de Bruijn indices: a bound variable is the number of abstractions between it and its binder.

The syntax of terms is added a second representation for variables and the type of the abstraction is consequently changed:

$$u, v \triangleq x^A \mid \mathbf{n} \mid \lambda^{\mathbf{A}}.\mathbf{u} \mid u\ v$$

Note that we do not require de Bruijn indices to come with their types anymore: since we are sure that these variables are bound, we are able to find their types in their abstractions.

Environments cannot be managed as before. Now, the environment for named variables $\mathcal{I}$ can be fixed once and for all, since it will not change during interpretation. However, to type and interpret bound variables, we need a context g, which is a list of deep types, recursively enriched when interpreting abstractions.

Let us start with the typing function. The context to type bound variables is the list of the types previously encountered in abstractions. Typing the de Bruijn index n is thus returning the n^{th} element of the context, if it exists. The recursion starts with an empty context, since a well-formed term must be closed for de Bruijn indices.

$$\begin{array}{rcl}
\text{infer'}_g(x^A) & = & A \\
\text{infer'}_g(n) & = & g[n] \\
\text{infer'}_g(\lambda^A.u) & = & A \hookrightarrow \text{infer'}_{A::g}(u) \\
\text{infer'}_g(u\ v) & = & \left\{ \begin{array}{l} B \text{ if infer'}_g(u) = A \hookrightarrow B \text{ and infer'}_g(v) = C \text{ and } A = C \\ \text{fails otherwise} \end{array} \right.
\end{array}$$

$$\text{infer}(t) \quad = \quad \text{infer'}_{[\,]}(t)$$

The interpretation of a context g, named $\mathcal{G}$, is a function that maps an integer n to some object belonging to the interpretation of $g[n]$. Hence, the interpretation of a de Bruijn index n is $\mathcal{G}(n)$. Interpretation is thus described by these equations:

$$\begin{array}{rcl}
|x^A|'_{\mathcal{I},g,\mathcal{G}} & = & (A, \mathcal{I}(x^A)) \\
|n|'_{\mathcal{I},g,\mathcal{G}} & = & (g[n], \mathcal{G}(n)) \\
|\lambda^A.u|'_{\mathcal{I},g,\mathcal{G}} & = & (A \hookrightarrow U, z \mapsto i) \text{ if } |u|'_{\mathcal{I},A::g,\mathcal{G}(0 \leftarrow z, p+1 \leftarrow \mathcal{G}(p))} = (U, i) \\
|u\ v|'_{\mathcal{I},g,\mathcal{G}} & = & \left\{ \begin{array}{l} (B, i\ j) \text{ if } |u|'_{\mathcal{I},g,\mathcal{G}} = (A \hookrightarrow B, i) \text{ and } |v|'_{\mathcal{I},g,\mathcal{G}} = (C, j) \text{ and } A = C \\ \text{fails otherwise} \end{array} \right.
\end{array}$$

Note than when interpreting an abstraction, both the context and its interpretation must be enriched. For this latter, we must shift all the interpretations.

To implement this idea, the only difficulty is to convince Coq that $\mathcal{G}(n)$ has type $[g[n]]$: once more, we need to use dependent pattern matching to type the interpretation of de Bruijn indices.

The syntax of terms is added a new constructor:

```
Inductive term : Set :=
| Var : nat → type → term
| Dbr : nat → term
| Lam : type → term → term
| App : term → term → term.
```

The typing function is extended using an auxiliary function returning the n^{th} element of a list, if it exists:

```
Definition context := list type.

Fixpoint opt_nth (g:context) n :=
  match g,n with
    | nil, _ ⇒ None
    | t::_, O ⇒ Some t
    | _::q, S p ⇒ opt_nth q p
  end.

Fixpoint infer_aux (g:context) (t:term) : option type :=
  match t with
    | Var _ A ⇒ Some A
    | Dbr n ⇒ opt_nth g n
    | Lam A u ⇒
      match infer_aux (A::g) u with
        | Some B ⇒ Some (A⟶B)
        | _ ⇒ None
      end
    | App u v ⇒
      match infer_aux g u, infer_aux g v with
        | Some (A⟶B), Some C ⇒
          if A == C then Some B else None
        | _, _ ⇒ None
      end
  end.

Definition infer := infer_aux nil.
```

As we explained, the interpretation of a context g is a function that maps an integer n to some object belonging to the interpretation of the n^{th} type of g:

```
Definition interp_context (g: context) : Type :=
  forall (n: nat), match opt_nth g n with
                    | Some A ⇒ interp_type A
                    | _ ⇒ unit
                  end.
```

We define functions that remove or add an element at the beginning of the interpretation of a context, shifting the other elements:

```
Definition interp_tail A g (f: interp_context (A::g)) :
   interp_context g :=
  fun n ⇒ f (S n).
```

```
Definition interp_cons A g (f: interp_context g)
  (a: interp_type A) : interp_context (A::g) :=
  fun n ⇒ match n with
             | O ⇒ a
             | S n ⇒ f n
           end.
```

and a function that interprets an empty context:

```
Definition interp_nil : interp_context nil := fun _ ⇒ tt.
```

The only obstacle is to type in Coq the interpretation of de Bruijn indices. We must recursively explore the context and its interpretation:

```
Fixpoint interp_dbr (g:context) (n:nat) {struct n} :
  option {A: type & interp_context g → interp_type A} :=
  match g,n return
    option {A: type & interp_context g → interp_type A} with
    | nil, _ ⇒ None
    | A::g, O ⇒ Some (existT A (fun f ⇒ f O))
    | A::g, S n ⇒
      match interp_dbr g n with
        | Some (existT B b) ⇒
          Some (existT B (fun f ⇒ b (interp_tail f)))
        | _ ⇒ None
      end
  end.
```

which is finally used to define the whole interpretation function:

```
Variable I : sem_env.

Fixpoint interp_aux (g:context) (t:term) :
  option {A:type & interp_context g → interp_type A} :=
  match t with
    (* Named variables: nothing changed *)
    | Var x A ⇒ Some (existT A (fun _ ⇒ I x A))

    (* Indices: we use the function above *)
    | Dbr n ⇒ interp_dbr g n

    (* Abstractions: we use interp_cons instead of extend_env *)
    | Lam A u ⇒
      match interp_aux (A::g) u with
        | Some (existT U c) ⇒ Some (existT (A⟶U) (fun f z ⇒ c
           (interp_cons f z)))
        | _ ⇒ None
      end
    (* Applications: nothing changed *)
    | App u v ⇒
      match interp_aux g u, interp_aux g v with
        | Some (existT (A⟶B) b), Some (existT C c) ⇒
          match cast C A with
            | Cast k ⇒ Some (existT B (fun f ⇒ (b f) (k (c f))))
            | _ ⇒ None
          end
        | _, _ ⇒ None
      end
  end.

Definition interp t :=
  match interp_aux nil t as p return
    match p with
      | Some (existT A _) ⇒ option (interp_type A)
      | None ⇒ option Prop
    end
    with
    | Some (existT A b) ⇒ Some (b interp_nil)
    | None ⇒ None
  end.
```

Remark 2 *At first reading, one might think that defining the interpretation of contexts as* **Definition** `interp_context := list {A:type & interp_type A}` *would simplify this development, especially the* `interp_dbr` *function. This is indeed the case, but at the expense of the* `interp_aux` *function. This latter would then have type* `term →` `interp_context →` `option {A:type & interp_type A}` *which is impossible as we explained in* **Remark 1***.*

3.2.4 Prenex polymorphism

Prenex polymorphism can be easily implemented in STLC by adding named type variables, which are implicitly universally quantified. This does not complicates term equality: terms differing by type variables are considered as different.

This addition is quite transparent for interpretation: we just have to add an environment for type variables behaving exactly like `sem_env`, at the type and the term levels.

3.3 From shallow to deep: reification

Reification computes the deep representation of a concrete term, by extracting its structure. Contrary to interpretation, non-trivial reification functions cannot be implemented in CIC. We must use an oracle that returns a deep term and the only thing that we can check is that its interpretation is indeed the original concrete term.

3.3.1 The quote tactic

Coq provides the quote tactic, that is able to perform simple reification when the source language has named variables, no binders and is mono-sorted, like in **Example 2.5**. Once written, the deep embedding (using primitives given by quote to represent the variables) and the interpretation function, quote transforms any goal g into the interpretation function applied to the deep representation of g.

Example 3.2 *We give an example of the* quote *tactic applied to* **Example 2.5**. *We first need to use the primitive* index *provided by* quote *to represent free variables:*

```
Inductive F : Set :=
| Bottom : F
| Var : index → F
| Or : F → F → F.
```

and varmap *to interpret them:*

```
Fixpoint interp (rho:varmap Prop) F :=
  match F with
    | Bottom ⇒ False
    | Var x ⇒ varmap_find True x rho
    | Or a b ⇒ (interp rho a) ∨ (interp rho b)
  end.
```

We can for instance use the tactic on the following goal:

```
Parameter P: Prop.
Goal False ∨ P ∨ True.
  quote interp.
```

which transforms the goal into:

```
interp
  (Node_vm True (Node_vm P (Empty_vm Prop) (Empty_vm Prop))
     (Empty_vm Prop))
  (Or Bottom (Or (Var (Left_idx End_idx)) (Var End_idx)))
```

Environments are represented using balanced trees. True *is reified as a variable since it does not belong to the source language.*

3.3.2 More complex reification

quote is a very useful tactic, but it works only for simple cases. In the developments presented in this thesis, we need to reify languages whose terms have different types and with sharing (see **Part II**).

There is currently no generic tool in Coq to do this, so we have to get our hands a little dirty by writing case-by-case reification functions at the ML level, manipulating the Ocaml type for Coq terms: constr.

The idea is quite simple: in Ocaml, the constr type can be inspected to get back the structure of shallow Coq terms. However, the manipulation of Coq terms at the ML level is not documented yet. This section briefly explains the key points needed for reification, hoping that a generic documentation will appear in the future. We illustrate this on **Example 2.5** for the sake of clarity (the techniques are the same for a more complex language).

In Coq, we have a deep embedding `term` of some source language and an interpretation function `interp_term : sem_env → term → Type`. We want to write a tactic that, given a goal, computes `I` and `t` to change it into `interp_term I t`. `t` must be maximally reified: variables are only parts of the goal that do not belong to the source language.

In Ocaml, we need to define:

- the deep embedding `term` of the language;
- a hash table `valuation : (constr, nat)Hashtbl.t` to associate a variable to each subterm that cannot be reified further;

and use them to write our main function: `reification : constr → term`. This function tries to recursively recognize terms of our source language; when it reaches an atomic subexpression, it returns a variable and adds the correspondence in `valuation`.

We finally need a function mapping Ocaml deep terms onto Coq deep terms `make_constr : term → constr` and another one mapping our valuation onto a Coq environment for the interpretation function `make_valuation : unit → constr`, to build our tactic.

We present the code when the source language is **Example 2.5**.

Example 3.3 *We first recall the terms of **Example 2.5** and give a way to interpret them where the environment is a list of propositions:*

```
Inductive term : Set :=
| Bottom : term
| Var : nat → term
| Or : term → term → term.


Section Interpretation.

  Variable valuation : list Prop.

  Let rho x :=
    match opt_nth valuation x with
      | Some P ⇒ P
      | None ⇒ False
    end.

  Fixpoint interp_term F :=
    match F with
      | Bottom ⇒ False
      | Var x ⇒ rho x
      | Or a b ⇒ (interp_term a) ∨ (interp_term b)
    end.
```

```
End Interpretation.
```

The deep embedding in Ocaml is very similar (we also have to embed natural numbers):

```
type nat = O | S of nat

type term =
  | Bottom
  | Var of nat
  | Or of term * term
```

We define the hash table and a function to get the next available variable:

```
let valuation : (Term.constr, nat) Hashtbl.t = Hashtbl.create 17
let current_nat = ref O
let next_nat () =
  let res = !current_nat in
  current_nat := S !current_nat;
  res
```

In Ocaml, we have access to Coq's constants by finding them in the modules in which they are defined. For instance, the `False` *constant is defined in the* `Coq.Init.Logic` *module and can be accessed like this:*

```
let cFalse = lazy (Coqlib.gen_constant_in_modules "Reification"
    [["Coq";"Init";"Logic"]] "False")
```

For every constant `CST` *we need, we access it in Ocaml and call it* `cCST`*.*

The reification function is just a recursive match of a `constr` *against the constants it could be:*

```
let rec reification t =
  (* Decomposing the term into a function applied to (possibly
     zero) arguments *)
  let c,args = Term.decompose_app t in

  (* If the function is False... *)
  if c = Lazy.force cFalse then (
    Bottom

  (* If the function is or... *)
  ) else if c = Lazy.force cor then (
    match args with
      | [a;b] → Or (reification a, reification b)
      | _ → assert false

  (* Otherwise we return a variable *)
  ) else (
    let n =
      try
        Hashtbl.find valuation t
      with
        | Not_found →
          let n = next_nat () in
          Hashtbl.add valuation t n;
          n in
    Var n
  )
```

We define the `make_constr : term → constr` *and* `make_valuation : unit → constr` *functions in a straightforward recursion. Coq applications can be built using the* `Term.mkApp : constr → constr array → constr` *function.*

We finally build the tactic by putting everything altogether:

```
let reify gl =
  (* Initialization of the hash table *)
  Hashtbl.clear valuation;
  current_nat := 0;

  (* Extraction of the conclusion of the goal *)
  let concl = Tacmach.pf_concl gl in

  (* Reification and construction of the new conclusion *)
  let r = reification concl in
  let c = make_constr r in
  let v = make_valuation () in
  let concl' = mkApp (Lazy.force cinterp_term) [|v;c|] in

  (* Finally changing the conclusion *)
  Tactics.change_in_concl None concl' gl
```

and extend Coq's tactics with this new one (this is camlp5 syntax):

```
TACTIC EXTEND reify
| [ "reify" ] → [ reify ]
END
```

We can for instance use the tactic on the following goal:

```
Parameter P: Prop.
Goal False ∨ P ∨ True.
  reify.
```

which transforms the goal into:

```
interp_term (True :: (P :: nil)%list) (Or Bottom (Or (Var 1)
   (Var 0)))
```

`True` *is reified as a variable since it does not belong to the source language.*

Part II

Cooperation with automatic theorem provers: SMTCoq, collaborating with SAT and SMT solvers through proof witnesses

This part of the thesis explores the communication between Coq and SAT and SMT solvers.

SAT and SMT solvers are modern automated provers, especially successful nowadays thanks to both their performance and their expressivity: many decision and optimization problems can be encoded into their logic. As we argued in the introductory chapter, their efficiency is at the cost of unsafety: as they grow on performance and complexity, it is well established that they are likely to contain bugs [BB09].

We investigate here a skeptical cooperation between SAT, SMT solvers and Coq: in addition to yes/no answers, the involved SAT and SMT solvers must return proof witnesses that justify these answers and that can be checked efficiently in Coq. This certified Coq checker serves as a basis to SMTCoq, a collection of tools that makes concrete the possibilities offered by such a cooperation: increasing the trust in SAT and SMT solvers and enjoying their power in Coq. In addition to the straightforward automation they bring, they can be used inside more complex decision procedures.

In **Chapter 4**, we formalize SAT and SMT problems, and discuss possible certificates for them. A Coq checker of the certificates of unsatisfiability is presented in **Chapter 5**, which is at the heart of SMTCoq (**Chapter 6**): a set of commands and tactics to check SAT and SMT answers and to enjoy automation in Coq. We then compare SMTCoq with existing tools in Coq and Isabelle in **Chapter 7**. We finally discuss the perspectives of this work (**Chapter 8**), notably a direct application of SMTCoq: the SAT part can be used to define a new decision procedure for machine integers, using bit-blasting.

Chapter 4

The SATisfiability and Satisfiability Modulo Theories problems

4.1 SAT solvers

4.1.1 The SAT problem

The Boolean Satisfiability Problem (SAT in short) is the problem of determining whether there exists an assignment of the Boolean variables appearing in a formula in conjunctive normal form (CNF in short) to $\top$ or $\bot$ such that this formula is true. Solving this problem has been an active research area for long, since many classes of decision problems can be encoded into SAT. Among them, one can cite formal hardware verification [CTVW04], RNA structure prediction [GOS$^+$12], decision procedures for bit-vectors using bit blasting [KS08]. . . (this list is far from being exhaustive).

Historically, SAT is the first known example of a $\mathcal{NP}$-complete problem [Coo71]. Nonetheless, efficient algorithms have been developed that can solve problems containing thousands of variables and constraints in reasonable time. Most of them are extensions of the Davis–Putnam– Logemann–Loveland algorithm, also known as DPLL [DP60, DLL62], but others are based on survey propagation [BMZ05] or binary decision diagrams [Bry86]. Some of these algorithms can be easily instrumented to produce proof witnesses [NOT06]. Here we are only interested on the structure of proof witnesses in order to check them, but not on the way they are produced (one may refer to [NOT06] for their production).

We recall basic definitions and notations.

Definition 4.1 (Literal, clause, formula) *We consider a countable set of variables $\mathcal{V}$.* Literals *l and* clauses *C are given by the following grammar:*

$$l ::= v|\bar{v}$$

$$C ::= \Box|l \vee C$$

where $v \in \mathcal{V}$, $\Box$ stands for the empty clause and $\bar{\bullet}$ represents the involutive boolean negation. Clauses are considered up to associativity and commutativity of $\vee$. If a clause is nonempty, $\Box$ is omitted.

A formula in CNF*, also called* SAT instance*, is a finite set of clauses $\mathcal{S}$, seen as their conjunction.*

Definition 4.2 (Valuation, satisfiability) *A* valuation *is a function $\rho : \mathcal{V} \to \{\top, \bot\}$ associating a Boolean to each variable.*

We define the associated interpretation of formulas in CNF into $\{\top, \bot\}$ by induction:

- *for literals: $|v|_\rho = \rho(v)$ and $|\bar{v}|_\rho = \neg\rho(v)$;*
- *for clauses: $|\Box|_\rho = \bot$ and $|l \vee C|_\rho = |l|_\rho \vee |C|_\rho$;*
- *$|S|_\rho$ is the conjunction of the interpretations of its clauses, with the usual convention that $|\emptyset|_\rho = \top$.*

A set of clauses $\mathcal{S}$ is satisfiable *if and only if there exists a valuation ρ such that $|\mathcal{S}|_\rho = \top$. Conversely, $\mathcal{S}$ is* unsatisfiable *if and only if for any valuation ρ, $|\mathcal{S}|_\rho = \bot$.*

Here are basic examples of satisfiable or unsatisfiable sets of clauses.

Example 4.1 *The following sets of clauses are satisfiable:*

$$\emptyset \qquad \{x\} \qquad \{x \vee \bar{x}\} \qquad \{x \vee \bar{y}, \bar{x} \vee y\}$$

The following sets of clauses are unsatisfiable:

$$\{\Box\} \qquad \{x, \bar{x}\} \qquad \{x \vee y, \bar{x}, \bar{y}\} \qquad \{x \vee y, \bar{x} \vee y, \bar{y}\}$$

We can now define the SAT problem:

Definition 4.3 (SAT problem) *A SAT problem consists in deciding if a formula in CNF is satisfiable or not.*

The pigeon-hole problems are well-known unsatisfiable SAT problems:

Example 4.2 *The pigeon-hole problem of length n, called H_n, is the problem to know whether it is possible to put $n+1$ pigeons in n holes, such that each hole contains at most one pigeon.*

It can be encoded in SAT in the following way. We consider $n(n+1)$ variables $(x_{i,j})_{(i,j)\in[\![1,n+1]\!]\times[\![1,n]\!]}$, representing the fact that the i^{th} pigeon is in the j^{th} hole. We have two sets of constraints:

- *every pigeon must be in a hole: $\forall i, x_{i,1} \vee \cdots \vee x_{i,n}$;*
- *a hole contains at most one pigeon: $\forall j, \forall i \neq i', \bar{x}_{i,j} \vee \bar{x}_{i',j}$.*

4.1.2 Input

The input problem is a formula in CNF. SAT solvers share a common input format called the DIMACS format (named after the SAT challenge proposed by the Center for Discrete Mathematics and Theoretical Computer Science in 1992). It is very simple: after a heading stating the numbers of variables and clauses, the clauses are listed, separated by “0” (and, usually, a carriage return). These clauses are lists of literals separated by spaces. Positive (resp. negative) literals are represented by positive (resp. negative) non-zero integers. It is possible to add comments, on lines starting with `c`.

Example 4.3 *The set of clauses $\{x \vee y, \bar{x} \vee y, \bar{y}\}$ is encoded in* DIMACS *as:*

```
c A toy example
p cnf 2 3
1 2 0
-1 2 0
-2 0
```

where 1 *represents* x *and* 2 *represents* y*.*

Since there is an agreement on the input format, it is possible to share easily SAT examples and benchmarks. A large set of benchmarks is available from the different SAT competitions (the last one was the SAT Challenge 2012[1]).

4.1.3 Certificates

Satisfiability

The class of satisfiable problems has an obvious certificate: a **valuation**. It is small (linear in the number of variables), easy to check (it is sufficient to compute the interpretation of the set of clauses given this valuation) and efficient to check (computing the interpretation is linear in the length of the initial problem).

Example 4.4 *We consider the following set of clauses:* $\mathcal{S} = \{x \vee y, x \vee \bar{y}, \bar{x} \vee z\}$*.* $\mathcal{S}$ *is satisfiable and a certificate is* $\{x \mapsto \top, y \mapsto \bot, z \mapsto \top\}$*.*

Notice that there is no uniqueness of the certificate: we could have chosen $\{x \mapsto \top, y \mapsto \top, z \mapsto \top\}$.

Unsatisfiability

Certificates for the class of unsatisfiable problems are not obvious, though. The simplest would be truth tables, but they are of course exponential in the number of variables and thus at least as long to check as to produce. The other certificates that have been imagined include:

- **proofs of a contradiction by resolution**;
- Frege systems, which are Hilbert-style propositional proof systems for reasoning with propositional formulas;
- cutting planes [CK05] and other geometric systems.

The idea that has received the most attention and which is implemented by many solvers is to return a proof by resolution.

Let first recall the resolution rule, introduced by Robinson [Rob65].

Definition 4.4 (Resolution rule) *The* resolution rule *builds the clause* $C \vee D$ *from two clauses* $v \vee C$ *and* $\bar{v} \vee D$*, where no variable appear with one polarity in* C *and the other in* D*:*

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

The variable v *is called the* resolution variable*. A comb tree of resolutions is a* resolution chain*.*

[1] The SAT Challenge 2012: http://baldur.iti.kit.edu/SAT-Challenge-2012.

It is straightforward that resolution builds a clause that is equisatisfiable with the two initial clauses:

Lemma 1 (Correctness) *We take two clauses C and D such that no variable appear with one polarity in C and the other in D.*
For any valuation ρ, $|v \vee C|_\rho = \top$ and $|\bar{v} \vee D|_\rho = \top$ if and only if $|C \vee D|_\rho = \top$.

This rule has also been shown to be refutationally complete [Rob65]:

Lemma 2 (Completeness) *Given any unsatisfiable set of clauses $\mathcal{S}$, there exists a proof by resolution of the empty clauses whose leaves belong to $\mathcal{S}$.*

It entails that proofs by resolution are theoretically possible certificates. Moreover, they have shown to be rather compact in practice, especially for some classes of problems [DT10, MT12] (even if other kinds of problems do not have polynomially long resolution proofs, eg. pigeon-hole problems presented in **Example 4.2**). Extensions have also been considered, like adding symmetry detection steps [ASM06].

Checking proofs by resolution is polynomial in their lengths and can be done very efficiently as the remaining of **Part II** will show. Finally, they are easy to produce from SAT solvers reasoning [NOT06]. All these fact certainly explain their success in modern SAT solvers.

Let illustrate these certificates.

Example 4.5 *We consider the following set of clauses: $\mathcal{S} = \{x \vee y, x \vee \bar{y} \vee z, \bar{x} \vee z, \bar{z}\}$. $\mathcal{S}$ is unsatisfiable and a certificate is:*

$$\dfrac{\dfrac{x \vee y \qquad \dfrac{x \vee \bar{y} \vee z \qquad \bar{z}}{x \vee \bar{y}}}{x} \qquad \dfrac{\bar{x} \vee z \qquad \bar{z}}{\bar{x}}}{\Box}$$

Like for satisfiability, there is no uniqueness of proofs by resolution.

4.2 SMT solvers

Given the success of SAT solving, it has been extended to decide problems defined in more expressive systems. In particular, the Satisfiability Modulo Theories Problem (SMT in short) is the problem of determining if a formula is satisfiable with respect to combinations of background theories. SMT solvers also have a large amount of applications, from type inference [RKJ08] to software synthesis [KMPS10]. This problem is decidable under some conditions (see [NO79] for instance), but may be undecidable otherwise. A usual extension is to allow quantifiers in the formulas, which is in general a source of undecidability.

The basic algorithm behind an SMT solver is an interaction between a SAT solver and decision procedures for conjunctions of formulas for each theory supported by the solver [NOT06]. Starting from a propositional abstraction of the initial problem – which consists in replacing all the atoms of theories by fresh variables – the SAT solver generates models and the theory solvers try to refute them. When a SAT model is consistent with all the theories, the initial problem is found satisfiable. Otherwise, a new clause corresponding to a theory lemma is added to the SAT problem in order to rule out the model. The SAT solver can then be called again to generate another model. Since there are only a finite number of SAT models, this enumeration eventually terminates. If the empty clause is derived by the SAT solver, the initial problem is unsatisfiable. Large efforts are made to make this interaction as efficient as possible and to design effective decision procedures. Once again our goal is to understand possible SMT certificates and how they can be formally checked.

4.2.1 The SMT problem

We start by defining our notion of theory and give some examples.

Definition 4.5 (Theory) *A* signature *is the data of:*

- *a set of types $\mathcal{S}_T$;*
- *a set $\mathcal{S}_f$ of function symbols with given arity and type;*
- *a set $\mathcal{S}_P$ of predicate symbols with given arity and type.*

Given a signature Σ, terms *t and* atoms *a are typed objects constructed by the following grammars:*

$$t ::= x|f(t_1, \dots, t_n)$$

$$a ::= P(t_1, \dots, t_m)$$

where x is a variable, $f \in \mathcal{S}_f$ has arity n and $P \in \mathcal{S}_P$ has arity m such that types are respected.

A many-sorted theory *(abbreviated as* theory*) $\mathcal{T}$ is a subset of the set of first-order formulas built on top of terms. Elements of a theory are called* theory lemmas*.*

We define the interpretation *of a signature by:*

- *interpreting each type variable;*
- *interpreting each function symbol $f : A_1 \to \cdots \to A_n \to A$ by a function $|f| : |A_1| \to \cdots \to |A_n| \to |A|$;*
- *interpreting each predicate symbol $P : A_1 \to \cdots \to A_m \to \text{Bool}$ by a predicate $|P| : |A_1| \to \cdots \to |A_m| \to \text{bool}$;*

We give two common examples of theories: congruence closure and linear integer arithmetic.

Example 4.6 *The theory of congruence closure $\mathcal{T}_{EUF}$ is defined by the signature containing:*

- *one type, written U;*
- *function symbols of type $U \to \cdots \to U$;*
- *predicate symbols of type $U \to \cdots \to U \to Bool$ and one particular predicate symbol $=: U \to U \to Bool$ for equality.*

Two examples of atoms of this theory are $f(x, g(y)) = e$ *and* $P(f(z, e))$ *where* $x, y, z : U$ *are variables,* $e : U, f : U \rightarrow U \rightarrow U, g : U \rightarrow U$ *are function symbols and* $P : U \rightarrow Bool$ *is a predicate symbol.*

The theory lemmas are the formulas constructed by the closure of reflexivity, symmetry and transitivity of equality and congruence of functions and predicates with respect to equality. An example of a theory lemma is $e = f(x, g(y)) \Rightarrow P(f(z, e)) \Rightarrow P(f(z, f(x, g(y))))$.

Example 4.7 *The theory of linear integer arithmetic* $\mathcal{T}_{LIA}$ *is defined by the signature containing:*

- *one type, written* I*;*
- *function symbols* $0 : I$, $S : I \rightarrow I$, $P : I \rightarrow I$, $+ : I \rightarrow I \rightarrow I$, $- : I \rightarrow I \rightarrow I$
- *predicate symbols* $=: I \rightarrow I \rightarrow Bool$, $<: I \rightarrow I \rightarrow Bool$, $\leqslant: I \rightarrow I \rightarrow Bool$, $>: I \rightarrow I \rightarrow Bool$, $\geqslant: I \rightarrow I \rightarrow Bool$

Two examples of atoms of this theory (using the standard notations) are $3x + 2y = -1$ *and* $-7 \leqslant -4$ *where* $x, y : I$ *are variables.*

The theory lemmas are the formulas whose standard interpretation in $\mathbb{Z}$ *are provable. Two examples of theory lemmas are* $3x + 2y = -1 \Rightarrow -x + y = -3 \Rightarrow x = 1$ *and* $-7 \leqslant -4$.

We now consider that we work within a combination $\mathcal{C}$ of many-sorted theories $\mathcal{T}_1, \ldots, \mathcal{T}n$. In this setting, we can define the notion of SMT instance.

Definition 4.6 (SMT instance) *An* SMT instance *is a SAT instance in which some Boolean variables are replaced by first-order formulas over atoms of* $\mathcal{C}$.

This definition introduces a **stratification**: an SMT instance is a formula in CNF built on top of first-order formulas, which themselves contain terms of $\mathcal{C}$. It implies that we have two levels of propositional connectives: conjunction, disjunction and negation used to define the formula in CNF, written using the notations of **Definition 4.1** (ie. $\vee$ for disjunction and using a set for conjunction) opposed to the connectives used in first-order formulas, written with different symbols to avoid the confusion (eg. $\bigvee$ for disjunction and $\bigwedge$ for conjunction). We illustrate this by an example.

Example 4.8 *An example of an SMT instance in the theory* $\mathcal{T}_{LIA}$ *is*

$$\{x \geqslant 7 \bigwedge y \leqslant -4, \overline{x \geqslant 2} \vee \overline{y < 3}\}$$

It contains two clauses: $x \geqslant 7 \bigwedge y \leqslant -4$ *and* $\overline{x \geqslant 2} \vee \overline{y < 3}$. *The first clause contains only one positive literal and the second clause contains the negative literals* $\overline{x \geqslant 2}$ *and* $\overline{y < 3}$.

Starting from interpretations for the signatures of $\mathcal{T}_1, \ldots, \mathcal{T}n$, a notion of satisfiability entails.

Definition 4.7 (Satisfiability) *A* valuation *assigns, for each variable* $v : T$ *of some theory, a term* $t : |T|$.

Given a valuation ρ, *we define the* interpretation *of SMT instances by the standard interpretation of formulas in CNF over first-order formulas, using the interpretation of signatures to interpret symbols.*

An SMT instance $\mathcal{S}$ is satisfiable *if there exists a valuation ρ such that $|S|_\rho$ is valid. It is* unsatisfiable *otherwise.*

Here are basic examples of satisable or unsatisable SMT instances, in the combination of $\mathcal{T}_{\text{EUF}}$ and $\mathcal{T}_{\text{LIA}}$.

Example 4.9 *The following instances are satisfiable:*

$$\{x = y\} \qquad \{x \geqslant 3\} \qquad \{g(0) = g(x - x)\}$$

The following instances are unsatisfiable:

$$\{x = y, f(x) \neq f(y)\} \qquad \{x \geqslant 3, x < -2\} \qquad \{g(1) = g(x - x), g(0) \neq g(1)\}$$

We can now define the SMT problem:

Definition 4.8 (SMT problem) *An SMT problem consists in deciding if an SMT instance is satisfiable or not.*

4.2.2 Input

While many SMT solvers implement their own input format, most of them also implement subsets of a common format: SMT-LIB v2 [BST10]. This format allows to describe any first-order formula of the combination of many theories like $\mathcal{T}_{\text{EUF}}$ and $\mathcal{T}_{\text{LIA}}$, but also arrays, bit-vectors... One may refer to the SMT-LIB v2 documentation for more details; we just give an example.

Example 4.10 *The SMT problem $\{g(0) = g(x - x)\}$ is encoded in SMT-LIB v2 as:*

```
(set-logic QF_UFLIA)
(declare-fun g (Int) Int)
(declare-fun x () Int)
(assert (= (g 0) (g (- x x))))
(check-sat)
(exit)
```

We first set the logic we consider (here, the quantifier free combination of $\mathcal{T}_{\text{EUF}}$ and $\mathcal{T}_{\text{LIA}}$), before declaring variables with their types. We then assert the problem and ask if it is satisfiable or not.

Like for SAT, having a common format is useful to design a set of examples and benchmarks. Such benchmarks are given by the SMT competitions[2] and the SMT-LIB.

4.2.3 Certificates

Satisfiability

Like for the SAT case, **valuations** are small and efficient certificates for the class of satisfiable problems. We give an example in $\mathcal{T}_{\text{LIA}}$.

Example 4.11 *We consider the following set of clauses: $\mathcal{S} = \{x \geqslant 7 \bigwedge y \leqslant -4, y < 2\}$. $\mathcal{S}$ is satisfiable and a certificate is $\{x \mapsto 9, y \mapsto -4\}$.*

[2] The SMT-COMP 2012: `http://smtcomp.sourceforge.net/2012`.

Unsatisfiability

If for the SAT problems, despite the different possible certificates for unsatisfiability, SAT solvers seem to agree on one of them, the output format for SMT solver vary widely from one prover to another, both in the syntax and in the spirit. The most used SMT solvers that give proof witness are Z3 [dMB08], CVC3 [BT07] and veriT [BdODF09] (some other solvers provide less informative witnesses). Since we want to support different solvers (possibly returning completely different formats of proof witnesses) in a generic way, we define here **our own notion of certificates**, into which the proof witnesses of the different SMT solvers can be translated. These certificates correspond to a proposal from the ANR DeCert initiative for a common output format (see Section 8.1.1) and are close to the format of the output of veriT.

To also share the code with the SAT checker, we chose certificates that extend SAT certificates, in such a way that they remain compact and quite easy to check: it is still a **proof tree of the empty clause, with additional rules**.

A rule takes a set of clauses as hypotheses and returns a clause which is implied. The set of hypotheses might be empty, in which case the returned clause must be a tautology. There are three kinds of rules, relying on the stratification of SMT instances:

- resolution still handles the propositional reasoning at the level of formulas in CNF;

- CNF rules handle the propositional reasoning at the level of first-order formulas of theories: combined with resolution, they transform first-order formulas into conjunctive normal form;

- theory rules handle the theory reasoning at the level of terms: each theory $\mathcal{T}$ has a set of rules defining the deductions in this particular theory.

The following example illustrates CNF and theory rules; they are detailed in the description of the Coq checker in the next chapter.

Example 4.12 *CNF rules include:*

$$\frac{P \bigwedge Q}{P}\ \text{CNF} \qquad\qquad \frac{}{\overline{P \bigwedge Q} \vee P}\ \text{CNF}$$

We give examples of rules for the theories $\mathcal{T}_{\text{EUF}}$ and $\mathcal{T}_{\text{LIA}}$:

$$\frac{}{\overline{x = y} \vee \overline{P(f(x))} \vee P(f(y))}\ \text{EUF} \qquad\qquad \frac{}{\overline{x \geqslant 7} \vee x \geqslant 2}\ \text{LIA}$$

It is important that each theory rule deals only with one theory, since it allows to use dedicated decision procedures, instead of a combination of decision procedures, even if the initial problem relies on a combination of theories. Moreover, since we separated CNF computation from theory reasoning, the decision procedures for theories will not need to handle propositional reasoning.

The following example illustrates these three levels of deduction.

Example 4.13 *We consider the following SMT problem:* $\mathcal{S} = \{x \geqslant 7 \bigwedge y \leqslant -4, \overline{x \geqslant 2}\}$*.* $\mathcal{S}$ *is unsatisfiable and a proof witness is:*

$$\dfrac{\dfrac{x \geqslant 7 \bigwedge y \leqslant -4}{x \geqslant 7}\ \text{CNF} \qquad \dfrac{\dfrac{}{\overline{x \geqslant 7} \vee x \geqslant 2}\ \text{LIA} \qquad \overline{x \geqslant 2}}{\overline{x \geqslant 7}}\ \textsc{Reso}}{\square}\ \textsc{Reso}$$

The next chapter will emphasize the modularity of this certificate format: it is easy to add new theories to the checker. Besides, theories can be considered in the broader sense: for instance, we present in Section 5.3.4 a way to handle silent simplifications performed by SMT solvers that fits in this model.

4.3 Conclusion on certificates

When a problem is satisfiable, a certificate is an assignment of the variables present in the problem. Checking such a certificate is very easy in Coq: one has to replace each variable by its assignment, in order to obtain a closed term, that can be computed into $\top$ or $\bot$.

Unfortunately, we are far more interested by unsatisfiability: as we will see in **Chapter 6**, provability problems can be encoded into unsatisfiability ones. That is why, in the remaining of this part, we emphasize the design of an unsatisfiability checker and its applications.

For unsatisfiability, we presented a very generic format for certificates: we work with clauses and resolution; theories are separated from one another and appear only on specific points of the certificate. This genericity will help us define a checker both efficient and modular, in the next chapter.

Our certificate format is different from answers returned by automatic provers: it corresponds to objects that are checked in Coq and a translation must be done to actually verify concrete proof witnesses. We thus distinguish these two notions in the remaining of **Part II**, using the following vocabulary:

- *certificates* are the formal objects presented in this chapter;
- *proof witnesses* are the concrete objects returned by SAT and SMT solvers.

Chapter 5

An efficient and modular Coq checker for SAT and SMT

We want to exploit the genericity of our certificate format to design a modular checker on two aspects:

- it must be easy to support new provers in the processus. This is why we distinguished the notions of proof witness from certificate: the checker works with certificates and, to add a new prover, one only has to write an unproved preprocessor encoding its specific format of proof witness into our format of certificate;
- it must be easy to support and check new theories. Our certificates facilitate this: since theory lemmas are pure (the theory rules belong only to one theory), there is no interaction between theories, so we can add a new one independently from what already exists. Communication between them is ensured by the generic resolution rule.

Note that SMT solvers are most of the time also designed to be modular in terms of theory solvers: for instance, the Nelson-Oppen algorithm [NO79] allows to combine independent theory solvers as long as they agree on the equality predicate. However, this process is complete only under certain restrictions that the Coq checker presented here does not require.

This chapter presents the implementation of a Coq checker for the certificates presented in the previous chapter achieving these goals.

5.1 Architecture of the Coq checker

The Coq checker, presented in Figure 5.1, is a certified Coq program checking if a certificate proves the unsatisfiability of a given input problem. It is proved to be correct, but it is not complete: this means that whenever it answers "yes", we are sure that the initial problem is indeed unsatisfiable; but otherwise, we do not know. Since we do not have control on the SAT or SMT solver, we cannot consider being complete. The best we can do is solving as many problems as possible on the set of existing benchmarks, as we will see in **Chapter 7**.

In our SAT/SMT certificates, new clauses are generated until reaching the empty clause. *Small checkers* are dedicated to the computation of these clauses, each of them in one particular domain. The role of the *main checker* is to dispatch pieces of the certificate (called

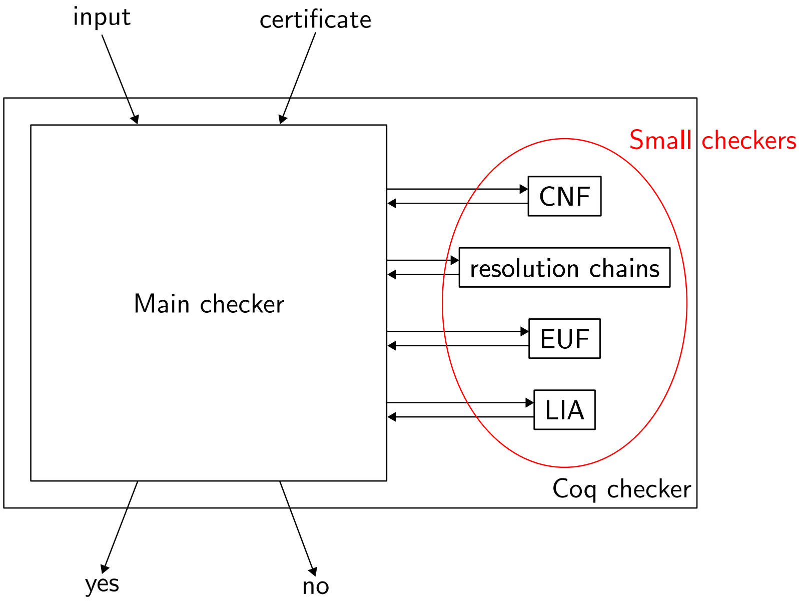


Figure 5.1: Architecture of the Coq checker

steps) to the corresponding small checkers and to check that the last computed clause is the empty clause.

Small checkers are independent from one another: they only need to share the representation of clauses and (a part of) the set of clauses computed so far. We will see how to choose this in order to be efficient, in time and in space. For the moment, we consider the theoretical representations of clauses C and sets of clauses $\mathcal{S}$ defined in **Chapter 4**. We call *states* the different sets of clauses appearing during computation. We note `step` the type for steps.

Definition 5.1 (Small checker) *A* small checker *is the data of:*

- *a function* `sc` : $\mathcal{S} \rightarrow$ `step` $\rightarrow \mathcal{S}$
- *a proof of correctness of this function* `sc_ok` : $\forall$`(s:`$\mathcal{S}$`) (c:step)`$\mathcal{I}$`,`$|s|_{\mathcal{I}} \rightarrow$ $|$`sc s c`$|_{\mathcal{I}}$

where $|\bullet|_{\mathcal{I}}$ *is the interpretation function for states as presented in Section 3.2.*

`sc_ok` expresses the fact that `sc` transforms a satisfiable state into a satisfiable one: the result is a consequence of the argument. After many applications of such small checkers, if the empty clause belongs to the final state, it thus mean that the initial state was unsatisfiable.

Notice that if the step given as input of a small checker is wrong, the checker must all the same return an implied clause. In this case, it returns a clause containing only one literal whose interpretation is always $\top$, so that **the correctness property is preserved whatever the small step is**.

This presentation emphasizes the second aspect of modularity: adding a new theory is just giving a new small checker, with the lightweight interface presented above.

Now that we presented the general ideas behind SMTCoq, we shall enter into the details of its implementation.

5.2 The main checker

5.2.1 Representation of states

To sum up, the main checker:

- defines a state containing the initial clauses;
- successively calls the small checkers on steps, which produces new clauses added to the state;
- checks that the last clause that was added to the state is empty.

In this process, it appears crucial to access and add clauses in the state efficiently. To do so, we rely on Coq's persistent arrays presented in Section 2.3.2: **a state is an array of clauses**. It is important to keep this array as small as possible, by re-using a cell as soon as the clause it holds is known to be not used anymore for further computations. This allocation will be computed upstream, by what we call the output preprocessor (see Section 6.1.1).

5.2.2 A piece of code

It must now be clear that, given a small checker for any possible step of certificates, the main checker is really easy to write and to prove correct. We illustrate this by giving the Coq code for the main checker.

We currently have small checkers for the following steps:

- `sc_res` for resolution;
- `sc_cnf` for CNF computation;
- `sc_euf` for congruence closure;
- `sc_lia` for linear integer arithmetic;
- `sc_spl` for simplifications (we will see in Section 5.3.8 that, in the actual proof witnesses returned by most SMT solvers, some simplification steps are omitted, which are handled by this small checker)

and the associated correctness proofs. We also have types for the corresponding steps (`step_res`, `step_cnf`, ...).

The `step` type is thus the following inductive:

```
Inductive step : Set :=
| Res : step_res
| Cnf : step_cnf
| Euf : step_euf
| Lia : step_lia
| Spl : step_spl.
```

and a step is checked by this straightforward function:

```
Definition step_checker (s:state) (st:step) : state :=
  match st with
    | Res st' ⇒ sc_res s st'
    | Cnf st' ⇒ sc_cnf s st'
    | Euf st' ⇒ sc_euf s st'
    | Lia st' ⇒ sc_lia s st'
    | Spl st' ⇒ sc_spl s st'
  end.
```

We now have all the ingredients to define the main checker. It takes three arguments: the initial state, containing the initial set of clauses; the certificate, which is an array of steps; and the index in the state where to look for the empty clause in the end of the process. It returns a Boolean. The correctness of this main checkers guarantees that when this Boolean is true, the initial set of clauses was unsatisfiable.

```
Definition main_checker
 (input:state) (certif: array step) (confl:int) : bool :=
  let s := Array.fold_left
    (fun s st ⇒ step_checker s st) input certif in
  is_false (get s confl).

Lemma main_checker_correct : forall input certif confl,
  checker input certif confl = true →
  forall 𝓘, ~(valid 𝓘 input).
(* Easy proof: application of the correctness of the small
   checkers *)
```

Adding a new small checker is thus straightforward. The only difficulty is to write the decision procedure, but integration is easy: we just have to make its internal representation of terms coincide with the one of states.

5.3 The small checkers

5.3.1 Representation of atoms, terms and formulas

For the moment, the logic we consider is QF_UFLIA, the quantifier free logic with uninterpreted functions and linear integer arithmetic. The underlying language is a multi-sorted first-order language without binders, but with term and type variables. This logical framework can be deeply and shallowly embedded in Coq, with a reflection between the two, as presented in **Chapter 3**.

We explained in Section 4.2.3 that certificates are stratified accordingly to the notion of SMT. The deep terms are accordingly stratified: *clauses* are built on top of *formulas* (which represent Boolean expressions), which are themselves built on top of *terms* and *atoms* of the theories. This stratification does not limit expressivity (as we will see in Section 5.3.6) and makes things easier and more efficient for small checkers: the resolution checker only needs to know about clauses, the CNF checker and some simplifications go down to the level of formulas and only theory checkers need to look at the structure of terms.

The deep terms are also maximally shared: every subexpressions of formulas and atoms are stored in an array and referred to by a number. It makes the terms smaller and the small checkers more efficient: it allows the work done on a given subterm be performed only once. This process is similar to hash-consing [FC06].

For a matter of clarity, we do not give the full deep embedding now, but refine it as we go along small checkers.

5.3.2 Resolution chains

The first small checker we present is a resolution checker. Given a list of clauses, it returns the clause obtained by successively resolving these clauses with one another. Before explaining its working, we detail the deep representation of literals and clauses, which play an important role in the efficiency.

Representation of literals and clauses

The efficient treatment of resolution chains requires a careful encoding of clauses and literals. First, propositional variables are encoded as 31-bit integers (we will see in the next section what they represent).

Since the resolution checker needs to deduce an implied clause even when the certificate is not correct (see Section 5.1), we need to have a deep clause whose interpretation is always true. This is done by having the convention that the location 0 represents the constant $\top$; hence, each environment $\mathcal{I}$ is now such that $\mathcal{I}(0) = \top$, and consequently the interpretation of the clause containing only this literal is always true.

Literals are either variables or their negation. They can be easily encoded from variables using the parity: the positive literal associated to i is $2i$ and the negative one is $2i + 1$. Their interpretation is thus:

$$|l|_{\mathcal{I}} = \begin{cases} \mathcal{I}(l/2) & \text{if } l \text{ is even} \\ \neg\mathcal{I}((l-1)/2) & \text{if } l \text{ is odd} \end{cases}$$

This representation is very efficient since parity check and division by two are very fast: they are directly performed by machine integer operations as explained in [AGST10].

To be able to represent all the literals, the number of variables is restricted to $2^{30} - 1$; but this is not a strong restriction since current solvers are far away from solving such problems in general.

Finally, clauses are represented as sorted lists of integers without redundancy.

Table 5.1 sums up our representation of terms so far.

Term	Representation
$\top$	0
variable	$x \in [\![1; 2^{30} - 1]\!]$
positive literal	$2x$
negative literal	$2x + 1$
clause	sorted list of literals

Table 5.1: Representation of terms from the point of view of the resolution checker

Example 5.1 *We represent the variable x by* 1*, y by* 2 *and z by* 3 *(remember that* 0 *represents* $\top$*). The clause $x \vee \bar{y} \vee z$ is represented by the list* $[2; 5; 6]$*:* 2 *is the positive literal associated* *to the variable* 1 *representing* x*,* 5 *the negative literal associated to* 2 *representing* y *and* 6 *the positive literal associated to* 3 *representing* z*.*

The small checker for resolution chains

A resolution step is a piece of certificate pointing to an array of clauses that can be successively resolved with one another. In our Coq representation, it consists in a data type containing two pieces of information:

- an array of integers representing the successive index in the state where to look for the clauses to resolve – this is the second argument of the constructor below;
- the cell of the state where to put the resulting clause (Section 6.1.1 explains how it is computed) – this is the first argument of the constructor below.

It is thus defined by:

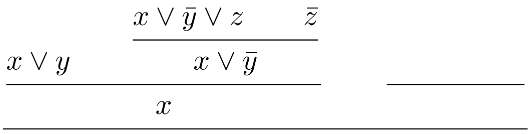
```
Inductive step_res : Type :=
| Res : int → array int → step_res.
```

The following example illustrates this encoding.

Example 5.2 *We consider the certificate of* **Example 4.5***. First, we encode the variables by 31-bits integers:* x *is represented by* 1*,* y *by* 2 *and* z *by* 3*. Second, we fill the initial state with the input* $\{x \vee y, x \vee \bar{y} \vee z, \bar{z}, \bar{x} \vee z\}$*, which can be represented by the array of four cells:*

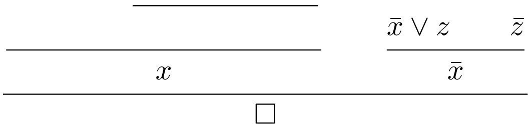
0	1	2	3
$x \vee y$	$x \vee \bar{y} \vee z$	$\bar{z}$	$\bar{x} \vee z$
`[2;4]`	`[2;5;6]`	`[7]`	`[3;6]`

The last object to encode is the certificate. It can be decomposed into the following two resolutions chains:



(1)

and



(2)

and thus the whole certificate is an array of two cells, one containing each resolution step:

(1)	(2)
`Res 0 [|1;2;0|]`	`Res 0 [|3;2;0|]`

The first step means that the main checker must call the resolution checker (since it is a resolution) successively on the clauses present in the cells 1*,* 2 *and* 0 *of the current state, and put the resulting clause in the cell indexed by* 0*. Its application thus updates the initial state into:*

0	1	2	3
x	$x \vee \bar{y} \vee z$	$\bar{z}$	$\bar{x} \vee z$
`[2]`	`[2;5;6]`	`[7]`	`[3;6]`

Similarly, the second step means that the main checker must call the resolution checker (since it is a resolution) successively on the clauses present in the cells 3*,* 2 *and* 0 *of the* ***current*** *state, and put the resulting clause in the cell indexed by* 0*. Its application thus updates the state into:*

0	1	2	3
$\square$	$x \vee \bar{y} \vee z$	$\bar{z}$	$\bar{x} \vee z$
`[]`	`[2;5;6]`	`[7]`	`[3;6]`

It remains for the main checker to verify that the clause at position 0 is indeed the empty clause.

Let us now detail the algorithm implemented by the small checker for resolution chains. The representation of clauses we chose allows to compute resolution linearly: we run through both lists until we find the resolution variable. This algorithm is very similar to the fusion of two sorted lists used in merge sort. If no resolution is possible, the checker returns the union of the two clauses; if more than one resolution are possible, the resolution computed is trivially true. It ensures the correctness of this (very simple) small checker.

5.3.3 CNF computation

With our previous small checker, proof witnesses for SAT problems in CNF can be checked in Coq. The next step is to be able to verify the transformation of a formula into an equisatisfiable formula in CNF. In refutation based on resolution, this is usually done using a technique proposed by Tseitin [Tse70]. This involves generating a new variable for every subterm of the formula; with these new variables, the CNF transformation is linear. It is this idea that we are going to implement in our setting.

Representation of Boolean formulas

In the previous section, we said that variables were represented by integers. Actually, those integers are the Tseitin variables associated to the input formula: each sub-formula is shared and put in an array, and an integer i represents the sub-formula contained in the i^{th} cell.

The Coq type that represent formulas is:

```
Inductive form : Type :=
| Fatom (_:int)
| Ftrue
| Ffalse
| Fand (_:array int)
| For  (_:array int)
| Fimp (_:array int)
| Fxor (_ _:int)
| Fiff (_ _:int)
| Fite (_ _ _:int)
| Fnot2 (_:int).
```

where connectives are either $\top$, $\bot$, conjunction, disjunction, implication, exclusive or, equivalence and Boolean branching (the **if ... then ... else ...** construction when the return type is a Boolean). The integers that some constructors take as arguments are literals: this encodes the negation so there is no constructor for it. However, double negation is explicit (this is the constructor Fnot2), in order to represent the formula $\neg\neg x$ faithfully. Note that the connectives Fand and For are n-ary operators which allows a more efficient subsequent computation. The base case of formulas are Fatom, in the perspective of the definition of terms and atoms in the next section; once more, these atoms are encoded by integers.

As we announced in Section 5.3.1, the formulas are maximally hashed. This is visible in the type defined above by the fact that it is not a proper recursion: we would expect the type of formulas to be a tree, but the nodes here take integers as arguments instead of formulas. This is because this type actually represents a formula only when it comes with an array containing all its sub-formulas, named t_form. In this case, the integers represent the literal associated to the cell of t_form where to look for the sub-formula.

This representation takes little memory since it implements maximal sharing. It also has the advantage to match exactly the notion of Tseitin variables, since any sub-formula is encoded by an integer. Finally, it also has the property that the equality between two formulas can be computed in constant time, by an integer comparison.

Table 5.2 sums up our representation of terms so far.

Term	Representation
atom	$x \in [\![0; 2^{31} - 1]\!]$
formula	t_form.[y] where y : int
positive literal	$2y$
negative literal	$2y + 1$
clause	sorted list of literals

Table 5.2: Representation of terms from the point of view of the CNF checker

To ensure the invariant on environment (that literal 0 is always valuated to $\top$), the first cell of the table must contain Ftrue.

We illustrate this encoding by the following example.

Example 5.3 *The formula* $\overline{(x \bigwedge y) \bigvee \overline{x \bigwedge y}}$ *can be encoded by the literal* 9 *using the formula table:*

t_form =	$\top$	x	y	$x \bigwedge y$	$(x \bigwedge y) \bigvee \overline{x \bigwedge y}$
	Ftrue	Fatom 0	Fatom 1	Fand [|2;4|]	For [|6;7|]

with the interpretation for atoms defined by $\mathcal{I}(0) = x$ *and* $\mathcal{I}(1) = y$*.*

We notice that the sub-formula $x \bigwedge y$ *appears twice in the formula but is shared in the table (at location 3): this representation allows maximal sharing.*

To ensure that our table does not contain infinite terms so that formulas can be interpreted, we preserve some well-formedness condition on the table: if a literal m appears in the formula stored at location n, we always have $m/2 < n$. With this condition, we can define the Boolean interpretation $|f|_{\mathcal{I}}$ recursively over the formula f, where $\mathcal{I}$ is the interpretation of the atoms.

In Coq, this condition could be stated using dependent types: a hash-table of variables could be a dependent pair composed of a an array and a proof that this array is well-formed. In this case, Coq's type system would have ensured that hash-tables are always well-formed.

Nonetheless, automatically constructing dependent pairs can be difficult, whereas it is not needed in our particular case of writing a Boolean checker: instead of returning `true` if the certificate if the proof of the input, it returns `true` if the hash-table is well-formed and the certificate is a proof of the input. Hence there is no need to state this condition at all, it will be part of the job of the checker to verify it.

The small checker for CNF transformation

Tseitin identifies around 40 generic tautology schemes that give the meaning of connectives [Tse70]. Instantiated with the variables appearing in a given formula F, it forms a set of clauses which is in CNF and is equisatisfiable with F. We are not going to give them all, since they are quite generic and the essence can be understood on the example of the conjunction:

Example 5.4 *Tseitin tautologies for the $\bigwedge$ connective are:*

$$\frac{}{\overline{x_1 \bigwedge \cdots \bigwedge x_n} \vee x_i}\ \textsc{AndNeg}_i \qquad \frac{}{(x_1 \bigwedge \cdots \bigwedge x_n) \vee \bar{x}_1 \vee \cdots \vee \bar{x}_n}\ \textsc{AndPos}$$

In addition to these tautologies, we can also consider direct implications, which are not sufficient to perform a full CNF transformation, but avoid unnecessary resolution steps when possible.

Example 5.5 *The implications corresponding to the tautologies of* **Example 5.4** *are:*

$$\frac{x_1 \bigwedge \cdots \bigwedge x_n}{x_i}\ \textsc{ImmAndPos}_i \qquad \frac{\overline{x_1 \bigwedge \cdots \bigwedge x_n}}{\bar{x}_1 \vee \cdots \vee \bar{x}_n}\ \textsc{ImmAndNeg}$$

The rules and their names are directly inspired by the proof witness specification of veriT.

The step for CNF transformation in the certificate is an algebraic data type containing a list of cases to check instantiations of these tautologies and implications:

```
Inductive step_cnf : Type :=
| AndNeg (pos:int) (_:int) (_:int)
| AndPos (pos:int) (_:int)
| ImmAndPos (pos:int) (_:int) (_:int)
| ImmAndNeg (pos:int) (_:int)
| ...
```

In each small step, `pos` is the index of the cell in the state in which the resulting clause is put. The other arguments are:

- for `AndNeg` (and similar rules for other connectives): the position of the formula $x_1 \bigwedge \cdots \bigwedge x_n$ in `t_form` and the index i of the projection;
- for `AndPos` (and similar rules for other connectives): the position of the formula $x_1 \bigwedge \cdots \bigwedge x_n$ in `t_form`;
- for `ImmAndPos` (and similar rules for other connectives): the position of the clause containing only the literal representing $x_1 \bigwedge \cdots \bigwedge x_n$ in the state and the index i of the projection;
- for `ImmAndNeg` (and similar rules for other connectives): the position of the clause containing only the literal representing $\neg(x_1 \bigwedge \cdots \bigwedge x_n)$ in the state.

We illustrate this encoding on an example.

Example 5.6 *The following certificate proves the unsatisfiability of $a \bigwedge \bar{a}$:*

$$\dfrac{\textsc{ImmAndPos}_1\ \dfrac{a \bigwedge \bar{a}}{a} \qquad \dfrac{a \bigwedge \bar{a}}{\bar{a}}\ \textsc{ImmAndPos}_2}{\Box}\ \textsc{Res}$$

.

First, we need to define the array of formulas `t_form`. We can choose the following representation if the variable a is encoded by 0:

$\top$	a	$a \bigwedge \bar{a}$
`Ftrue`	`Fatom 0`	`Fand [|2;3|]`

Second, we encode the initial state, which is an array of two cells filled with the input (the second cell will be used later):

0	1
$a \bigwedge \bar{a}$	-
`[4]`	-

Finally, we encode the certificate. It can be decomposed into three steps: the two CNF steps $\textsc{ImmAndPos}_1$ *and* $\textsc{ImmAndPos}_2$ *and the resolution step. It is thus the following array of three steps:*

(1)	(2)	(3)
`ImmAndPos 1 0 1`	`ImmAndPos 0 0 2`	`Res 0 [|0;1|]`

The step `ImmAndPos 1 0 1` is sent to the small checker for CNF, and means: "put at position 1 in the state, the clause obtained from the formula at position 0 in the current state, by taking the first projection". It thus updates the state into:

0	1
$a \bigwedge \bar{a}$	a
`[4]`	`[2]`

Similarly, the step `ImmAndPos 0 0 2` is then sent to the small checker for CNF. This time, it computes the second *projection, and put it at position 0 in the state, which results into:*

0	1
$\bar{a}$	a
`[3]`	`[2]`

Finally, the step `Res 0 [|0;1|]` *is sent to the small checker for resolution, which will result into (as we saw in the previous section):*

0	1
□	a
`[]`	`[2]`

The algorithm implemented by the small checker is straightforward: it looks at the head symbol of the given sub-formula, and computes the corresponding tautology or implication if possible – otherwise, it returns the clause `[0]` which is always true. In the implementation, the type `step_cnf` is less verbous that what we presented here, since similar steps for different connectives can be grouped together and the small checker can distinguish them regarding the head connective of the sub-formula taken as an argument.

Remark 3 *The combination of resolution with small steps for CNF transformation is a variant of the* extended resolution *[Tse70] proof system.*

5.3.4 Theories

Representation of terms and atoms

To handle the theories, we now need to refine the representation of terms and atoms. The difference with the terms we manipulated so far is that they are not only Booleans anymore, but can have other simple types. So we also need to be able to represent the types and the interpretation function for terms will rely on deep types as presented in **Chapter 3**.

The deep terms are those of the theories currently handled by our checker: congruence closure and linear integer arithmetic. We thus have three kinds of base types: uninterpreted – which are indexed by machine integers – integers (corresponding to $\mathbb{Z}$) and Booleans:

```
Inductive type :=
| Tindex : int → type
| TZ : type
| Tbool : type.
```

There are possibly different uninterpreted types: this allows to deal with `Coq` terms containing many different types not handled by the SMT solvers we consider, like user defined or higher-order types, without losing the information that they are actually different.

We can build types of a given arity on top of this, represented as a curryfied codomain and a domain:

```
Definition ftype := list type * type.
```

Since functions are always fully applied in this language, there is no need for a function type, so this is a simplification of **Chapter 3**.

Like formulas, terms are maximally hashed and come with an array of sub-terms `t_atom`. They are (possibly empty in the case of constants) applications of either interpreted (`Aop`) or uninterpreted (`Aapp`, indexed by machine integers) functions:

```
Inductive atom : Type :=
| Aop  (_ : op)  (_: list int)
| Aapp (_ : int) (_: list int).
```

where interpreted operators are:

```
Inductive op : Type :=
| Zcst (_ : Z) | Zle | Zlt | Zplus | ...
| Eq (_ : type).
```

where `Z` is the Coq type of binary integers. Once more, the arguments of functions are machine integers encoding the index of the corresponding sub-atom in `t_atom`. Note that the constructor `Zcst` embedding the constants of $\mathbb{Z}$ takes a binary integer as an argument – this does not have harmful effects on efficiency since no operation is performed on them.

We illustrate this encoding by the following example.

Example 5.7 *We consider the formula $f(x) < 1 \bigvee g(y+1) < 1$, where the variable x has an uninterpreted type. We encode the variable x by 0, y by 1, f by 2 and g by 3. The formula then can be represented by the literal 6 with the following tables (written from left to right then top to bottom):*

`t_atom` =

x	y	1
`Aapp 0 []`	`Aapp 1 []`	`Aop (Zcst 1)[]`

$f(x)$	$y+1$	$f(x) < 1$
`Aapp 2 [0]`	`Aop Zplus [1;2]`	`Aop Zlt [3;2]`

$g(y+1)$	$g(y+1) < 1$
`Aapp 3 [4]`	`Aop Zlt [6;2]`

`t_form` =

$\top$	$f(x) < 1$	$g(y+1) < 1$	$f(x) < 1 \bigvee g(y+1) < 1$
`Ftrue`	`Fatom 5`	`Fatom 7`	`For [|2;4|]`

Table 5.2 sums up our final representation of terms.

Term	Representation
variable	$x \in [\![0; 2^{31} - 1]\!]$
atom	`t_atom.[z]` where `z : int`
formula	`t_form.[y]` where `y : int`
positive literal	$2y$
negative literal	$2y+1$
clause	sorted list of literals

Table 5.3: Representation of terms from the point of view of the theory checkers

The small checker for congruence closure

The congruence closure checker has three possible steps, corresponding to instantiating one of these tautologies:

- transitivity: $x_1 \neq x_2 \vee \cdots \vee x_{n-1} \neq x_n \vee x_1 = x_n$
- function congruence: $x_1 \neq y_1 \vee \cdots \vee x_n \neq y_n \vee f\ x_1\ \ldots\ x_n = f\ y_1\ \ldots\ y_n$
- predicate congruence: $x_1 \neq y_1 \vee \cdots \vee x_n \neq y_n \vee \neg P\ x_1\ \ldots\ x_n \vee P\ y_1\ \ldots\ y_n$

In these clauses, the equality is understood up to symmetry and reflexivity of equality. For instance, $y \neq x \vee f\ x\ z = f\ y\ z$ is a valid instantiation of the function congruence rule, instead of the expected $x \neq y \vee z \neq z \vee f\ x\ z = f\ y\ z$. At small cost for the checker, it gives more compact certificates, since there is no step for reflexivity nor symmetry and it also avoids consequently resolutions.

The Coq type for step is thus simply:

```
Inductive step_euf : Type :=
| EqTr (pos:int) (_:clause)
| EqCgr (pos:int) (_:clause)
| EqCgrP (pos:int) (_:clause).
```

where the first argument is – as before – the index of the state where to put the resulting clause, and the second argument is the resulting clause obtained by the instantiation of the corresponding rule.

This format is inspired by veriT's proof format for congruence closure. For SMT solvers that do not give such details, we will need to find them when preprocessing the proof witness.

Example 5.8 *The certificate proving the unsatisfiability of $\{x = y, z = y, g(x) \neq g(z)\}$ given at the top of Figure 5.2 can be encoded using the array of atoms*

t_atom =

x	y
Aapp 0 []	Aapp 1 []

z	$x = y$
Aapp 2 []	Aop (Eq (Tindex 0))[0;1]

$z = y$	$x = z$
Aop (Eq (Tindex 0))[2;1]	Aop (Eq (Tindex 0))[0;2]

$g(x)$	$g(z)$
Aapp 3 [0]	Aapp 3 [2]

$g(x) = g(z)$
Aop (Eq (Tindex 0))[6;7]

the array of formulas

t_form =

$\top$	$x = y$	$z = y$	$x = z$	$g(x) = g(z)$
Ftrue	Fatom 3	Fatom 4	Fatom 5	Fatom 8

and the initial state:

0	1	2	3
$x = y$	$z = y$	$g(x) \neq g(z)$	-
[2]	[4]	[9]	-

as:

(1)	(2)	(3)	(4)
`EqTr 3 [3;5;6]`	`Res 0 [|0;3;1|]`	`EqCgr 3 [7;8]`	`Res 0 [|0;3;2|]`

We detail only step (1)*: the main checker sends this step to the small checker for congruence closure, which checks that the clause represented by* `[3;5;6]` *(which is* $x \neq y \vee y \neq z \vee x = z$*) is a valid instantiation of the transitivity rule. Since it is, it puts it at position* 3 *in the state, giving:*

0	1	2	3
$x = y$	$z = y$	$g(x) \neq g(z)$	$x \neq y \vee y \neq z \vee x = z$
`[2]`	`[4]`	`[9]`	`[3;5;6]`

Implementing a checker for these certificates is straightforward. For instance, to check the transitivity rule, we compare all the intermediate terms, possibly twice if there is a silent application of symmetry. This is done efficiently since the comparison between terms can be done in constant time: here **sharing is absolutely crucial**.

The small checker for linear integer arithmetic

Coq has already three decision procedures that suit linear integer arithmetic: Omega, a solver for problems in Presburger Arithmetic (see Chapter 20 of [Tea11]); ROmega, a variant of Omega with reflexive traces; and Micromega, a set of tactics for solving arithmetics goals over ordered rings [Bes06]. After making informal experiments on arithmetic goals coming from SMT benchmarks, it appeared that the `lia` tactic of Micromega was much faster than `omega` and `romega`, and efficient enough to be integrated into our SMT checker. As a result, we decided to use it instead of having detailed certificates, contrary to congruence closure.

This decision procedure also has a design which is really suited to be used as a small checker. Like SMTCoq, it calls an external solver that produces a certificate. It is associated to a Coq checker for this certificate. It is thus easily integrated in three steps:

- we translate our terms into Micromega's representation;
- the external solver is called upstream from the checker (during preprocessing, see Section 6.1.1);
- the Micromega Coq checker is called for each arithmetic step.

Thus, for the linear arithmetic checker, there is only one step, that contains the resulting clause and the Micromega certificate:

```
Inductive step_lia : Type :=
| Lia (pos:int) (cl:clause) (c:list ZMicromega.ZArithProof)
```

We give an example:

Example 5.9 *The certificate proving the unsatisfiability of* $\{x \leqslant 1, 1 \leqslant x, g(x) \neq g(1)\}$ *given at the bottom of Figure 5.2 can be encoded using the array of atoms*

A certificate proving the unsatisfiability of $\{x = y, z = y, g(x) \neq g(z)\}$ is:

$$\textsc{Res}\,\dfrac{\textsc{Res}\,\dfrac{\textsc{EqCgr}\,\dfrac{}{x \neq z \vee g(x) = g(z)} \qquad \dfrac{z = y \qquad \dfrac{x = y \qquad \dfrac{}{x \neq y \vee z \neq y \vee x = z}\,\textsc{EqTr}}{z \neq y \vee x = z}\,\textsc{Res}}{x = z}\,\textsc{Res}}{g(x) = g(z)} \qquad g(x) \neq g(z)}{\Box}$$

A certificate proving the unsatisfiability of $\{x \leqslant 1, 1 \leqslant x, g(x) \neq g(1)\}$ is:

$$\dfrac{\textsc{Res}\,\dfrac{\textsc{EqCgr}\,\dfrac{}{x \neq 1 \vee g(x) = g(1)} \qquad g(x) \neq g(1)}{x \neq 1} \qquad \textsc{Res}\,\dfrac{\dfrac{\dfrac{}{x = 1 \vee \overline{x \leqslant 1} \vee \overline{1 \leqslant x}}\,\textsc{Lia} \qquad x \leqslant 1}{x = 1 \vee \overline{1 \leqslant x}}\,\textsc{Res} \qquad 1 \leqslant x}{x = 1}}{\Box}\,\textsc{Res}$$

Figure 5.2: Examples of certificates for congruence closure and linear integer arithmetic

x	1	$x = 1$
`Aapp 0 []`	`Aop (Zcst 1)[]`	`Aop (Eq TZ)[0;1]`

`t_atom` =

$g(x)$	$g(1)$	$g(x) = g(1)$
`Aapp 1 [0]`	`Aapp 1 [1]`	`Aop (Eq (Tindex 0))[3;4]`

$x \leqslant 1$	$1 \leqslant x$
`Aop Zle [0;1]`	`Aop Zle [1;0]`

the array of formulas

`t_form` =

$\top$	$x = 1$	$g(x) = g(1)$	$x \leqslant 1$	$1 \leqslant x$
`Ftrue`	`Fatom 2`	`Fatom 5`	`Fatom 6`	`Fatom 7`

and the initial state

0	1	2	3
$x \leqslant 1$	$1 \leqslant x$	$g(x) \neq g(1)$	-
`[6]`	`[8]`	`[5]`	-

as:

(1)	(2)	(3)	(4)
`EqCgr 3 [3;4]`	`Res 2 [|2;3|]`	`Lia 3 [2;7;9] c`	`Res 0 [|3;0;1;2|]`

The certificate `c` *appearing in the* `Lia` *step is pre-computed by the external solver of Micromega. When the main checker reaches step* (3)*, it gives it to the LIA checker, which itself calls the Micromega checker to be sure that* `c` *actually proves the clause* `[2;7;9]`*; if it is the case, it puts this clause at position* 3 *in the state.*

5.3.5 Interpretation of terms

Terms (that is to say clauses, formulas and atoms) are interpreted as presented in Section 3.2, in particular using the trick of the intermediate function returning the deep type of the term. We have two environments:

- ρ_T for type variables, that return a Coq type;
- ρ_V for atom variables, that return both a type A and a Coq term whose type is $[A]_{\rho_T}$.

The only difference with Section 3.2 is that, as terms are hashed instead of being inductively defined, the recursion in the interpretation function must be unfolded.

5.3.6 Remark on expressivity

The terms are stratified between atoms and terms of the theories, formulas and clauses. This simplifies the small checkers that do not need to know about formulas or atoms (and make them more efficient). Compared to the current logic we implement (called QF_UFLIA in the SMT-LIB, for Quantifier Free with Unintepreted Functions and Linear Integer Arithmetic logic), the only kind of terms we cannot represent is branching with a type different from Boolean. For instance, `if x = 3 then y+2 else y-2` cannot be written directly in our representation of terms.

However, general branching can be encoded by adding a new variable representing its result. On the example above, it consists in replacing each occurrence of `if x = 3 then y+2 else y-2` by a new variable `z` and adding the clause `if x = 3 then z = y+2 else z = y-2`.

Using this trick, we do not lose expressivity compared to QF_UFLIA.

5.3.7 Remark on modularity about theories

Finally, to handle a new theory, it is sufficient to augment the type `op` with the operators of this new theory, to consequently extend the interpretation with them and to give the small checker together with a proof of its correctness.

We can even go further when the theory do not share symbols with other theories: we have to change neither `op` nor its interpretation. Indeed, as formulas appearing in the certificate are pure, the atoms of the new theory can appear as variables in the `t_atom` array and only the new small checker needs to access their contents (for instance, using another array of hashed terms). The environment for atom variables ρ_V will then be the interpretation function of the new theory.

5.3.8 Towards concrete proof witnesses: a small checker for silent simplifications

So far, we presented a checker for certificates in the logic QF_UFLIA. However, this is not sufficient to handle proof witnesses coming from actual SMT solvers, even for this logic: formulas are likely to be silently "simplified", in various ways – arithmetic operations may be simplified (eg. $2 + x + 3$ is transformed into $x + 5$), associativity and commutativity may be applied, n-ary operations may be flattened.... Unjustified simplifications most often appear at the beginning: the formula for which a proof witness is output is not the original formula. This is of course a major problem when integrating SMT solvers into Coq through proof witnesses; we propose a solution, implemented as a small checker, which appears to work correctly on most tests.

The problem posed by the simplification process can be stated as follows: we are given two terms and we need to prove that their interpretations are equivalent. The major issue is that we do not know at all the nature of the simplifications and to which theories they belong. We followed two approaches.

The first approach is a *syntactical* one: we perform a simultaneous descent in the two terms and check equivalences at each level. This can be used to check simplifications associated to associativity and commutativity, erasure of double negation, simple rewriting of arithmetic equations (eg. $a < b \equiv b > a \equiv \neg a \geqslant b$)... It is limited to very simple simplifications, but one major advantage is that it mixes various theories.

To check more complicated simplifications, we also follow a *semantical* approach: we send the equivalence between the two terms to one of the small checkers. The choice of this small checker is done by looking at the head symbol of the terms. It can perform non trivial simplifications, as for instance $3 + x + y + 7 + x = 2x + 10 + y$. The drawback is that it does not mix theories; if we have simplifications with different theories at the same time, we would need to decompose into many steps of simplifications.

Hence, the type of simplification steps is:

```
Inductive step_spl : Type :=
| SplSyntactic (pos:int) (orig:int) (res:int)
| SplArith (pos:int) (orig:int) (res:int) (l:list
   ZMicromega.ZArithProof)
| ...
```

As usual, the constructors take as a first argument the position in the state where the resulting clause should be put. The next two arguments are the position in the state of the initial clause and the literal that must be equivalent. `SplArith` takes also a Micromega certificate.

We give an example of the syntactic approach:

Example 5.10 *This certificate proving the unsatisfiability of* $\{x \geqslant 7 \bigwedge y \leqslant -4, x < 2\}$*:*

$$\cfrac{\cfrac{x \geqslant 7 \bigwedge y \leqslant -4}{x \geqslant 7}\;\text{CNF} \qquad \cfrac{\cfrac{}{\overline{x \geqslant 7} \vee x \geqslant 2}\;\text{LIA} \quad \cfrac{x < 2}{\overline{x \geqslant 2}}\;\textsc{Simp}}{\overline{x \geqslant 7}}\;\textsc{Reso}}{\square}\;\textsc{Reso}$$

can be encoded using the array of atoms

t_atom =

x	y	-4
`Aapp 0 []`	`Aapp 1 []`	`Aop (Zcst (-4))[]`

2	7	$x < 2$
`Aop (Zcst 2)[]`	`Aop (Zcst 7)[]`	`Aop Zlt [0;3]`

$x \geqslant 2$	$x \geqslant 7$	$y \leqslant -4$
`Aop Zge [0;3]`	`Aop Zge [0;4]`	`Aop Zle [1;2]`

the array of formulas

t_form =

$\top$	$x < 2$	$x \geqslant 2$	$x \geqslant 7$	$y \leqslant -4$	$x \geqslant 7 \bigwedge y \leqslant -4$
`Ftrue`	`Fatom 5`	`Fatom 6`	`Fatom 7`	`Fatom 8`	`Fand [|6;8|]`

and the initial state

0	1	2
$x \geqslant 7 \bigwedge y \leqslant -4$	$x < 2$	-
`[10]`	`[2]`	-

as:

(1)	(2)
`ImmAndPos 0 0 0`	`SplSyntactic 1 1 5`

(3)	(4)
`Lia 2 [7;4] c`	`Res 0 [|2;1;0|]`

The fact that we can handle silent simplifications using a small checker that fits in our model emphasizes that it is well suited to check certificates in an extendable way.

Chapter 6

SMTCoq: certified checker and tactics

Around the checker presented in the previous chapter, we have developed a set of straightforward applications to make its use possible and, we hope, easy.

The first aim of this work is to have a certified checker for SAT/SMT answers. It can be used both in Coq and in general programming languages.

The second aim is to enjoy the power of SAT and SMT inside Coq. To achieve this, we have designed tactics that, given a Coq goal, call external provers and check their answers. We will explain the possibilities offered by these tactics and how they could be improved.

At the present time, these two applications can handle two provers: the SAT solver ZChaff and the SMT solver veriT. We will explain what has to be done to deal with other provers. We also make clear the parts that need to be changed to add new theories than the one presented in the previous chapter.

6.1 Certified checker

The most straightforward application is the direct use of the Coq checker to check SAT/SMT answers. This corresponds to the diagram given in Figure 6.1. To check the answers given by SAT and SMT solvers, we need three components:

- the Coq checker presented in the previous chapter, which is certified (green color);
- parsers for SAT and SMT inputs (DIMACS or SMT-LIB), written in Ocaml. They are not certified and thus belong to the trusted base (red color). The code of the SMT-LIB parser is taken from the Alt-Ergo SMT solver, written by Sylvain Conchon, Evelyne Contejean, Stephane Lescuyer, Mohamed Iguernelala and Alain Mebsout;
- parsers and preprocessors for SAT and SMT answers, written in Ocaml. Their main role is to transform concrete proof witnesses coming from all the different provers into certificates in our common language. They are not certified, but this has no impact on soundness (blue color): if they go wrong, in the end the checker will fail. (Nonetheless, it would have an impact on completeness, if we were interested in this aspect.)

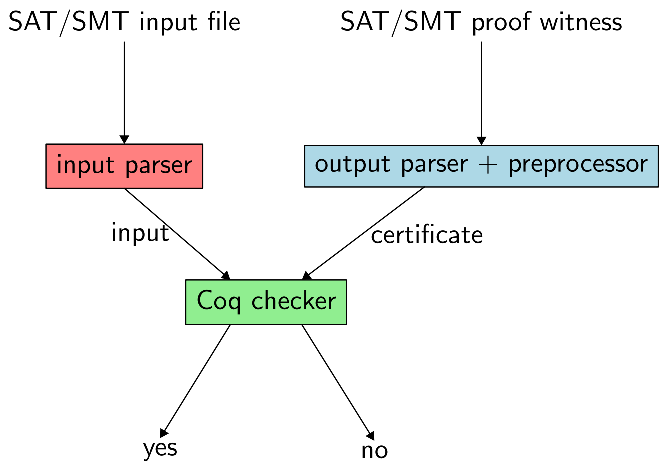


Figure 6.1: Architecture of the vernacular commands of SMTCoq

This architecture emphasizes the first aspect of the modularity of SMTCoq (as presented in the beginning of **Chapter 5**): to add a new SAT or SMT solver, one has only to write a new Ocaml preprocessor, but no Coq code nor proofs. The difficulty of writing such a preprocessor depends on the distance between the proof witnesses given by the new prover and our certificates. As we said, we currently have preprocessors for ZChaff and the quantifier free part of veriT, whose proof witnesses are easy to translate into certificates. A preliminary work investigated the preprocessing of Z3 proof witnesses, which are rather different [Gil12].

We first detail the role and the implementation of the preprocessors, before explaining the different uses that can be done from this architecture.

6.1.1 Output preprocessors

There are three main actions that a preprocessor should do:

- translate the terms used to state and solve the problem in our representation (see Section 5.3.4);
- translate the proof witness into a certificate;
- eventually perform some optimizations on the certificate.

To be efficient, the generation of terms and certificates should be done at the ML level. That is to say, the preprocessors do not generate a Coq source file, but rather define Coq terms using the Ocaml representation (the `constr` type presented in Section 3.3.2) and communicate with proof scripts through vernacular commands and tactics (see below). This avoids the Coq terms to be parsed and is also more natural for the user's point of view, since it is interactive.

Generation of terms

We give a simple and modular presentation of SAT and SMT terms. We keep the same stratification as before: we have two module interfaces, one for atoms (called `ATOM`) and the other for formulas (called `FORM`). We give one functorial implementation of the interface for formulas:

```
module Make (Atom:ATOM) : FORM with type hatom = Atom.t
```

and, currently, two implementations of ATOM, one for SAT solvers (for which atoms are only variables) and one for SMT solvers (with the representation of atoms detailed in Section 5.3.4). This way, to add new theories, one has only to refine the SMT implementation of ATOM, but everything else is unchanged. There is nothing to do to add new provers.

Let us now enter into details. The ATOM interface is:

```
module type ATOM = sig
  type t
  val equal : t → t → bool
  ...
  val to_coq : t → constr * constr
  ...
end
```

It is the data of a type, some functions to work with this type (like equality) and functions to export atoms into Coq terms. Notice that we do not impose anything on the type of atoms: in this way, any combination of theories is an implementation of this interface.

On the contrary, since formulas are implemented once and for all, we impose the type of formulas in the FORM interface:

```
module type FORM = sig
  type hatom
  type t
  type pform =
    | Fatom of hatom
    | Ftrue
    | Ffalse
    | Fand of t array
    | ...

  val equal : t → t → bool
  ...

  val get : pform → t
  ...

  val to_coq : t → constr * constr
  ...
end
```

This interface features three types: hatom is the type of atoms, t is the internal type of hashed formulas and pform is the type of nodes of the formulas. We have different kind of functions to work with this:

- standard functions, like equality testing;
- functions to generate hashed formulas from their expression;
- functions to export hashed formulas and sub-formulas arrays into Coq terms.

The SAT implementation of ATOM is very straightforward, since atoms are only variables. Concerning the Make functor and the SMT implementation of ATOM, we need to compute maximal sharing and store sub-terms in arrays. This is an adaptation of hash-consing [FC06].

Generation of certificates

The type of certificates is mostly a tree, whose nodes correspond to steps of the small checkers:

```
type 'hform step =
  | Resolution of 'hform clause list
  | AndNeg of 'hform clause * int
  | AndPos of 'hform clause
  ...
  | EqTr of 'hform list
  ...
  | Lia of 'hform list * Certificate.Mc.zArithProof list
  ...

and 'hform clause = {
            id    : int;
   mutable kind   : 'hform step;
   mutable pos    : int option;
   mutable used   : int;
   mutable prev   : 'hform clause option;
   mutable next   : 'hform clause option;
           value  : 'hform list option
}
```

It is parameterized by a type 'hform of formulas, which is meant to be instantiated with some implementation of FORM.t. The steps are the straightforward enumeration of the steps of the small checkers. In the case of linear integer arithmetic, we provide the certificate given by Micromega.

It relies on an intermediate record type for clauses which, in addition to allowing sharing, contains information to facilitate cell allocation (and optimizations, as we will see in the next subsection). Indeed, as we explained in Section 5.2, in the Coq certificate the intermediate clauses are going to be stored in an array, so we need to allocate clauses in this array when exporting certificates to Coq. This is simply computed by a linear traversal of certificates and a greedy allocation, which seems to produce an efficient allocation in practice. The pos field stores this allocation.

As we previously explained, to add a new prover, one has to translate the new proof witnesses into this type. Coq exportation, with clauses allocation, can be entirely reused.

To add a new theory, it is sufficient to add the corresponding steps to this type and accordingly change the exportation functions.

Optimizations

Reducing the length of proofs by resolution is an active research area [Cot10, FMP11]. Some optimizations are already implemented in SAT and SMT solvers.

We implemented very simple improvements to reduce the length of certificates:

- the removal of unused clauses;
- the sharing of common prefixes of resolution chains.

The clause type presented in the above section has a field name used which is used to record the number of uses of each intermediate clause, in order to be able to remove unused clauses very easily.

These optimizations transform certificates and thus can be totally reused when adding new provers. They must be extended when adding new theories.

6.1.2 A Coq checker

Vernacular commands

The first way to use SMTCoq as a proof checker is through two sets of Coq vernacular commands.

The first set of commands call the checker on given input file and proof witness: Zchaff_Checker and Verit_Checker, one for each prover. Their goal is to be able to check SAT and SMT answers inside Coq. For instance, the command Verit_Checker "file.smt2""file.log". returns a Boolean; if it is true, we are sure that file.log is a proof of file.smt2.

We also implemented two commands that allow to import SAT/SMT theorems inside Coq: Zchaff_Theorem and Verit_Theorem. For instance, the command Verit_Theorem theo "file.smt2""file.log". defines a new Coq term theo whose type is the problem posed by file.smt2, if file.log is correct (otherwise it fails). This could be useful if one wants to import, in a Coq development, different results that can be established by SAT or SMT solvers.

The performance of the first two vernacular commands are evaluated in Section 7.1.

Direct use

If one wants to use our checker in a large development, it can also be called directly, either at the Coq or at the Ocaml level. The main difficulty is that it requires to manipulate our representation of terms, but this should be made easier by the effort we put in making the code simple and functorial.

6.1.3 An extracted checker

One useful aspect of computational reflection is that our Coq certified checker can be extracted to a more general programming language. The extraction mechanism of Coq can produce Ocaml or Haskell code from our checker. It gives SAT and SMT users the possibility to use this certified tool without installing Coq, but in a general purpose language.

Regarding reliability, it adds the extraction mechanism of Coq to the trusted base. Currently, this mechanism has been shown to be sound on paper [Let04] and is being formally certified as well [Glo09].

We did not evaluate the performance of our extracted checker. However, the Ocaml extracted code compiled into native code should perform the same as our Coq vernacular commands, since we already use the native version of Coq with efficient data structures.

6.2 Tactics

A more interesting application for Coq users is to build tactics on top of this checker: it gives automation without compromising soundness. We added two tactics to Coq: `zchaff` and `verit`, that call the corresponding provers.

6.2.1 Practical use

Both tactics apply to goals of the form **`forall`** `x`$_1$ ... `x`$_n$`,` `b`$_1$ `=` `b`$_2$ where `b`$_1$ and `b`$_2$ are two Coq terms of type `bool`. The tactics either solve the goal, or fail.

`zchaff` solves goals that involve only Boolean reasoning. `verit` combines Boolean reasoning, equality and linear arithmetic. It thus can be seen as a combination of the tactics `tauto`, `lia` and a part of `congruence`, applied to Boolean goals. It does not fully implement `congruence`, since there is no special reasoning about inductive data types (we ignore injectivity and discrimination of constructors).

Users might be disconcerted by the use of Booleans instead of propositions, which are more common in Coq. We work with Booleans because we want to enjoy the power of SAT and SMT without requiring classical logic. The use of SMTCoq jointly with the Ssreflect plugin would help working with Booleans.

6.2.2 Architecture

An overview of the architecture of both tactics is given in Figure 6.2. Like on Figure 6.1, green indicates the certified part and blue shows the parts that are not certified, but without jeopardizing soundness. Here, the whole process can be trusted without compromising Coq: if something goes wrong during reification, solver, parser or preprocessor, the tactic will fail and the possibly wrong goal will not be solved. If it solves the goal, we are guarantied that it is correct.

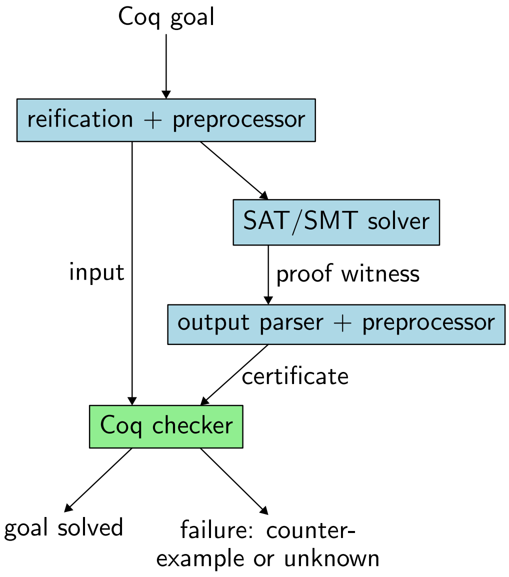

Coq goal
reification + preprocessor
SAT/SMT solver
input
proof witness
output parser + preprocessor
certificate
Coq checker
goal solved
failure: counter-example or unknown

Figure 6.2: Architecture of the tactics of SMTCoq

The parser, second preprocessor and checker have been presented in **Chapter 5** and at the beginning of this chapter. The new step is reification and first preprocessing.

6.2.3 Reification and first preprocessing

As we saw in Section 3.3, reification is performed by an Ocaml program that computes the deep representation of a Coq term t. The deep representation was given throughout **Chapter 5**.

The reification must compute:

- an integer, representing some literal l;
- an atom table and a formula table;
- two environments ρ_T and ρ_V

such that $|l|_{\rho_T,\rho_V} \equiv_c t$.

Once the goal is reified, it could be directly sent to the chosen automatic solver. However, the prover will be able to solve it only if it is formulated in a way that it understands: for instance, SAT solvers expect problems in CNF. Since we do not want Coq users to be aware of the working of the automatic solvers, we prefer to preprocess the goal before sending it to the prover. Of course, each step of preprocessing must be justified in the final certificate.

First, we compute the Tseitin CNF transformation of goals given to ZChaff. This is justified using the already existing steps for the small checker for CNF. We also apply some syntactical simplifications, like flattening the associative logical connectives, which are treated more efficiently by SAT and SMT solvers. This is justified by the certificates for simplifications.

Many other improvements could be implemented in this preprocessing. In particular, a future work is to improve expressivity by encoding some aspects peculiar to Coq or higher-order into first-order logic, like polymorphism and inductive definitions. This could be done by merging with the dp tactic [AF06], that encodes lots of Coq features into first-order logic before calling SMT solvers, but do not check their answers. It relies on the back-end of the Why program verification tool [FM07].

6.2.4 Modus operandi

We now explain how satisfiability solvers can be used to solve provability problems. It relies on the observation that $\forall \vec{x}, b_1 = b_2$ is provable if and only if $b_1 \neq b_2$ is unsatisfiable.

Figure 6.3 details the modus operandi of the tactics. Given the goal $\forall \vec{x}, b_1 = b_2$, we first reify the Coq term $b_1 \neq b_2$ into a deep term. This deep term is written in an input file for the chosen prover (DIMACS for SAT solvers and SMT-LIB for SMT solvers). Three possibilities may occur:

- in the ideal case, the prover answers that the problem is unsatisfiable and gives a correct proof witness of this unsatisfiability. In that case, the proof witness is translated into a certificate and the goal can be solved by an application of the correctness of the checker (see below);
- the prover may also return that the problem is satisfiable and give an assignment of the variables. In that case, the tactics fails, but gives an informative error message by providing the counter example given by the assignment;

- otherwise, the solver might not know or give a wrong proof witness, in which case the tactic fails without the possibility to help the user.

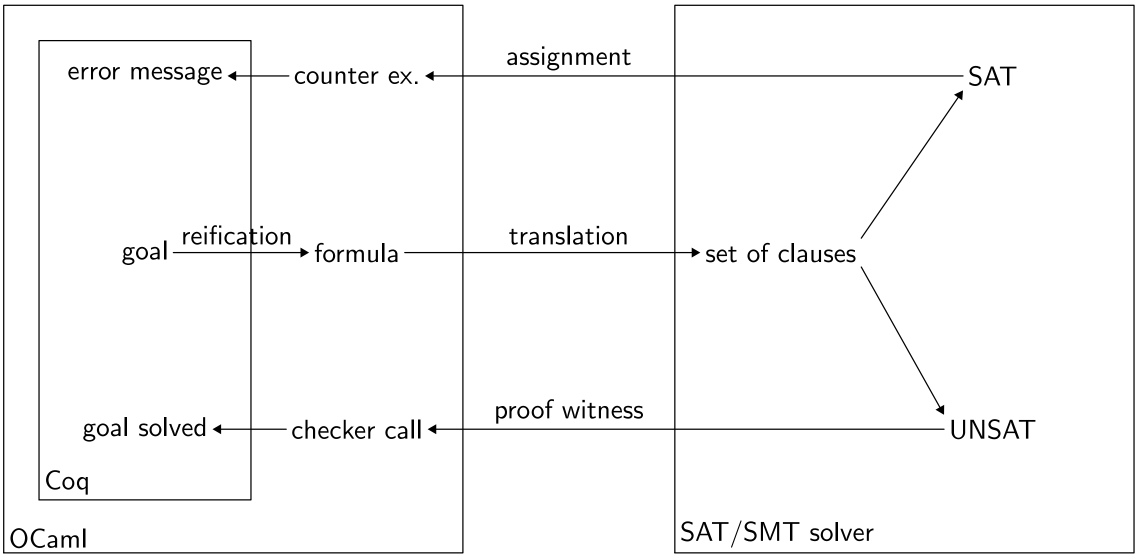


Figure 6.3: Modus operandi of the tactics

Let us observe what happens in the ideal case. We recall the statement of the correctness of the checker:

```
Lemma main_checker_correct : forall input certif confl,
  checker input certif confl = true →
  forall 𝓘, ~(valid 𝓘 input).
```

We proved the following corollary:

```
Corollary main_checker_correct_eq : forall l l1 l2 certif confl,
  is_equiv l l1 l2 = true →
  checker l certif confl = true →
  forall 𝓘, |l1|_𝓘 = |l2|_𝓘.
```

where `is_equiv` checks that `l` is a literal corresponding to the equivalence between the literals `l1` and `l2`.

This corollary directly applies to our goals: a proof of **forall** $x_1 \ldots x_n$, $b_1 = b_2$ is a term of the form **fun** $x_1 \ldots x_n \Rightarrow$ `main_checker_correct_eq l l1 l2 certif confl (refl_equal true) (refl_equal true)` $\mathcal{I}$ where `l`, `l1`, `l2` and $\mathcal{I}$ are terms and valuation generated by reification and `certif` and `confl` are the certificates preprocessed from the prover's proof witness. The most important point here is to notice that the proofs of `is_equiv l l1 l2 = true` and `checker l certif confl = true` are just reflexivity: since `l`, `l1`, `l2`, `certif` and `confl` are closed terms, Coq's type checker can *compute* the result of the applications of `is_equiv` and `checker`. This is the essence of **computational reflection**. Since computation does not appear inside proofs, the length of the final proof is the length of the certificate.

To be efficient, especially since we use arrays and machine integers, this computation is performed using the native computation in Coq.

Chapter 7

Evaluation of SMTCoq

SMTCoq was designed, among other things, to be efficient both as a proof checker and as a Coq tactic. In this chapter, we try to evaluate this claim against two different approaches: the verification of ZChaff proof witnesses [Web08] and Z3 proof witnesses [BW10] in Isabelle/HOL, and Ergo in Coq [LC09]. We also qualitatively highlight the differences with existing tools. The quantitative experiments we will present here all have been conducted in 2011 on an Intel Quad Core processor with 2.66GHz and 4Gb RAM, running Linux.

7.1 Evaluation of the checker

7.1.1 Qualitatively

Several SAT and SMT checkers have been integrated in LCF style interactive theorem provers including checkers for CVC Lite in HOL Light [MBG06], MiniSat and ZChaff in Isabelle/HOL [Web08], haRVey in Isabelle/HOL [FMM$^+$06], Z3 in HOL and Isabelle/HOL [BW10], SAT solvers in Coq [DFMS10]. To our knowledge, no SAT nor SMT checker have been certified by other means than interactive theorem provers.

We observe that a new checker has been written for every prover. SMTCoq tends to avoid this thanks to its modularity: it can handle both SAT and SMT solvers and great care has been taken to make it easily extendable. It should be small work to write preprocessors for our certificates compared to proving the correctness of a new checker (this seems to be confirmed by the preliminary preprocessor for Z3 proof witnesses [Gil12]).

Moreover, the implementation of SMTCoq in a proof assistant based on Type Theory makes it extractable as a certified checker. This should favor its deployment among the users of SAT and SMT solvers, even when they are not familiar with Coq.

7.1.2 Performance

We warn that fairly evaluating the performances of interactive theorem prover like Coq and Isabelle (which run in languages that use garbage collection) in applications that call external tools is a very difficult task and a comparison between both is even more risky, even if we took care to conduct the experiments in the best conditions as possible.

Resolution

We first look at the performance of the combination of the main checker with the small checker for resolution chains, which is always widely used since resolution is central in both SAT and SMT certificates. We took a set of 151 unsatisfiable industrial benchmarks from SAT Race'06 and '08, which range from 300 to 2.3 million variables and from $1,800$ to 8.9 million clauses. On each benchmark:

- we run ZChaff 2007.3.12 (with proof witness generation) on these benchmarks with a timeout of 300 seconds;
- when ZChaff succeeded, we run SMTCoq (using the vernacular command `Zchaff_Checker`, thus with preprocessing and in native-coq (not extracted)) on the proof witness with a timeout of 300 seconds.

Table 7.1 presents the number of benchmarks solved by ZChaff and, among them, the number of proof witnesses successfully checked by SMTCoq. The proof witnesses given by ZChaff have size from 41Kb to 205Mb. The times are the mean of the times for the 79 benchmarks on which ZChaff succeeded, in seconds.

ZChaff			SMTCoq		
#	%	Time	#	%	Time
79	52	70.2	79	52	21.5

Table 7.1: Standalone evaluation of the resolution checker

ZChaff was able to solve 79 benchmarks in the given time, which represents 52%. We first notice that SMTCoq was able to check all the proof witnesses given by ZChaff. We also observe that SMTCoq was much faster to check the certificate than ZChaff to compute it.

As a conclusion, ZChaff proof witnesses can be efficiently verified by SMTCoq, at small time cost. It does not add a bottleneck, since it is faster than ZChaff.

We want to make a comparison with the state-of-the-art Isabelle/HOL checker for ZChaff proof witnesses written by Alwen Tiu and Tjark Weber [Web08]. We take the same benchmarks as before and, when ZChaff succeeded, we run the Isabelle/HOL checker on the proof witness with a timeout of 300 seconds. We use Isabelle 2009-1 (running with Poly/ML 5.2).

Table 7.2 extends the previous table with the Isabelle/HOL results we obtained. Now, the times are the mean of the times for the 57 benchmarks on which the Isabelle/HOL checker succeeded, in seconds.

ZChaff			SMTCoq			Isabelle/HOL checker		
#	%	Time	#	%	Time	#	%	Time
79	52	51.9	79	52	17.5	57	38	100.

Table 7.2: Comparison with Isabelle/HOL resolution checker

It appears on these experiments that Isabelle/HOL can check less proof witnesses in the given time than SMTCoq and takes more time (and even more time than ZChaff). However, these results have to be taken with great care, since they disagree with the experiments conducted by Tjark Weber [Web08], which showed that the Isabelle/HOL checker was faster than ZChaff. A discussion with Tjark Weber is in progress to understand these differences. Possible explanations are:

- we do not conducted the experiments the right way, since we are not familiar with Isabelle/HOL;
- ZChaff and Isabelle/HOL might have evolved since Weber's experiment was conducted (in 2008);
- our set of benchmark is different and larger than the one taken in [Web08].

All the results for resolution are summarized in Figure 7.1, which presents the number of benchmarks solved through time. The steepening of the Coq curve enhances the efficiency of SMTCoq: the time curve is of course exponential since the length of certificates grows exponentially, but the exponential behavior starts later than for ZChaff and Isabelle/HOL.

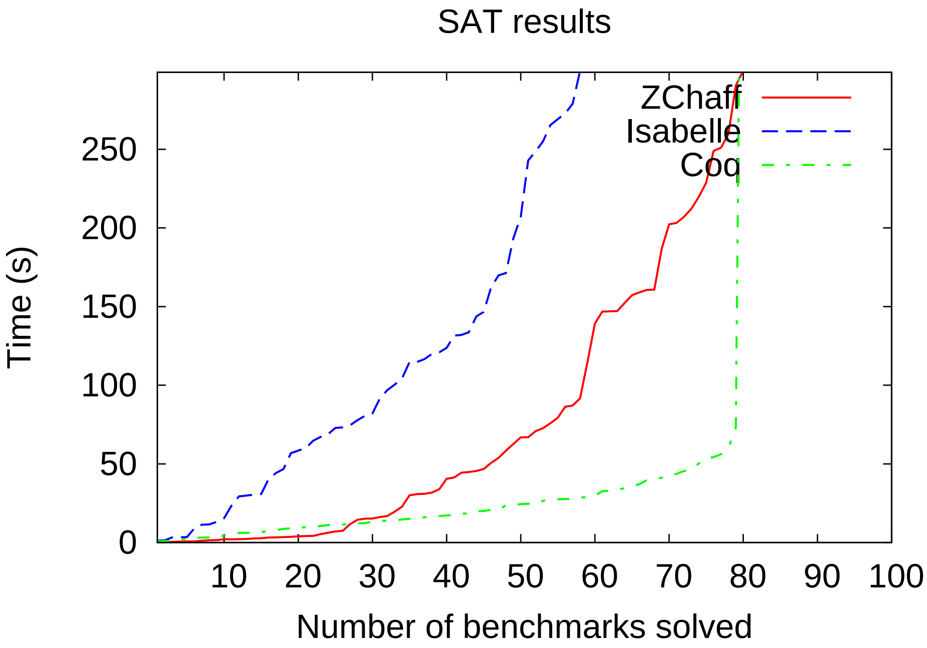


Figure 7.1: Evaluation of the resolution checker

Whole checker

We now evaluate the whole SMTCoq checker. We took a database of 2377 unsatisfiable industrial benchmark from the SMT-LIB[1] for theories QF_UF (congruence closure), QF_IDL (difference logic, a subset of linear integer arithmetic with polynomial decision procedures) and QF_LIA (linear integer arithmetic). On each benchmark:

- we run the development version of veriT (with proof witness generation) on these benchmarks with a timeout of 300 seconds;
- when veriT succeeded, we run SMTCoq (using the vernacular command `Verit_Checker`, thus with preprocessing and in native-coq (not extracted)) on the proof witness with a timeout of 300 seconds.

Table 7.3 presents the number of benchmarks solved by veriT and; among them, the number of proof witnesses successfully checked by SMTCoq. The times are the mean of the times for the benchmarks on which both veriT and SMTCoq succeeded, in seconds.

Benchmarks		veriT			SMTCoq		
Logic	#	#	%	Time	#	%	Time
QF_UF	1852	1816	98	7.2	1804	97	1.5
QF_IDL	409	368	90	11.1	349	85	54.3
QF_LIA	116	98	84	15.5	98	84	4.2

Table 7.3: Standalone evaluation of SMTCoq

veriT solved a large set of benchmarks in rather short time; unsolved benchmarks are mainly due to timeouts or to unknown answer (since veriT was not complete for QF_LIA when these experiments where conducted). SMTCoq was able to check most of the certificates, faster than veriT produced them for logics QF_UF and QF_LIA, but slower for logic QF_IDL. This might be explained by the fact that we do not use a special small checker for this logic but the LIA small checker; as a consequence, we do not benefit from fast decision procedures known for QF_IDL.

As a conclusion, veriT proof witnesses can be efficiently verified by SMTCoq, most of the time at small time cost. It would be an interesting improvement to write in Coq a decision procedure for difference logic.

It is even more difficult to compare SMTCoq with other tools, since none can check veriT proof witnesses. Nonetheless, we can try to compare the combination of veriT and SMTCoq with the combination of Z3 and the Isabelle/HOL checker [BW10] written by Sascha Böhme and Tjark Weber. We take the same benchmarks as before:

- we run Z3 2.19 (with proof witness generation) on these benchmarks with a timeout of 300 seconds;
- when Z3 succeeded, we run the Isabelle/HOL checker on the proof witness with a timeout of 300 seconds.

[1] The SMT-LIB benchmarks are available at `http://smtlib.org`.

Table 7.4 extends the previous table with the Z3 and Isabelle/HOL results we obtained. Now, the times are the mean of the times for the benchmarks on which the **four** tools succeeded (whereas for Table 7.3 it was the (most numerous) benchmarks on which veriT and SMTCoq succeeded), in seconds.

Benchmarks		veriT			SMTCoq		
Logic	#	#	%	Time	#	%	Time
QF_UF	1852	1816	98	6.5	1804	97	1.4
QF_IDL	409	368	90	6.3	349	85	37.8
QF_LIA	116	98	84	11.6	98	84	3.1

Benchmarks		Z3			Isabelle/HOL checker		
Logic	#	#	%	Time	#	%	Time
QF_UF	1852	1834	99	2.5	1775	96	25.8
QF_IDL	409	402	98	0.6	190	46	55.2
QF_LIA	116	107	92	0.7	96	83	46.6

Benchmarks		veriT + SMTCoq			Z3 + Isabelle/HOL		
Logic	#	#	%	Time	#	%	Time
QF_UF	1852	1804	97	7.9	1775	96	28.3
QF_IDL	409	349	85	44.1	190	46	55.8
QF_LIA	116	98	84	14.7	96	83	47.3

Table 7.4: Comparison with Isabelle/HOL checker

The top table presents individual results. We can observe that Z3 is faster than veriT, but as we said, we cannot say anything about SMTCoq and Isabelle/HOL, since they do not check the same proof witness.

However, we can compare the combination of veriT and SMTCoq with the combination of Z3 and Isabelle/HOL as *a posteriori* certified solvers. This is presented in the bottom table. The combination veriT + SMTCoq appears to solve more goals and to be faster than the combination Z3 + veriT. This may be explained by the fact that the Isabelle/HOL checker has more work to do than SMTCoq, since Z3 proof witnesses lack more information than veriT's. This should encourage the generalization of our certificates: they are easier to check with a small generation additional cost.

The results for QF_IDL are summarized in Figure 7.2, which presents the number of benchmarks solved through time.

7.2 Evaluation of the tactics

7.2.1 Qualitatively

Many interactive theorem provers implement automatic tactics using the skeptical approach. The external prover can be either a different tool we do not control, like SMTCoq, or a tool done on purpose, like the external provers of the Coq Micromega set of tactics or of the

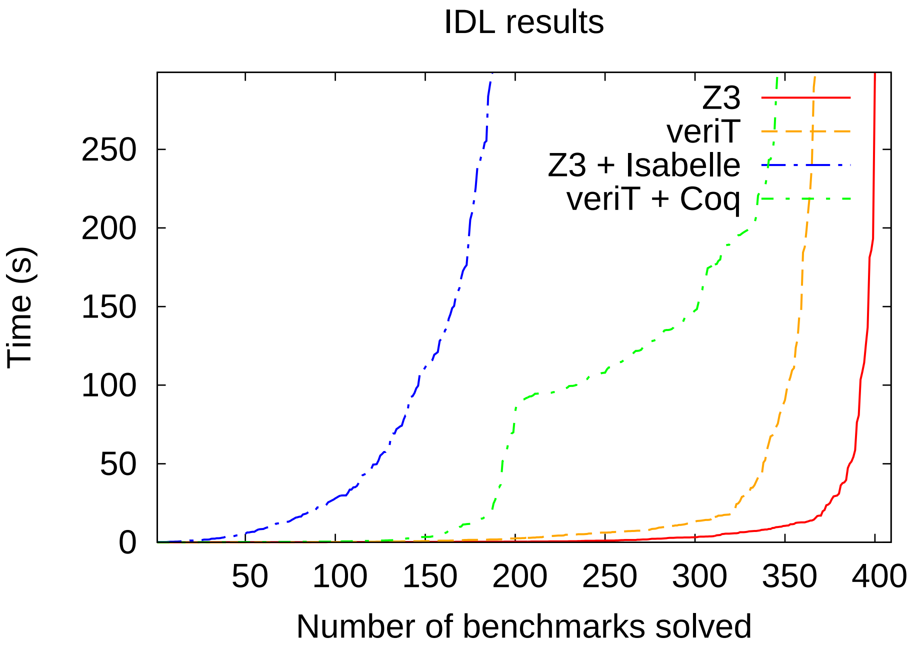


Figure 7.2: Evaluation of SMTCoq

Isabelle/HOL `blast` (a tableau prover) and `metis` (a resolution prover) tactics. The most impressive approach is Sledgehammer [BBP11], a Isabelle/HOL utility that combines calls to different external solvers. Most of these solvers are used as oracles for the internal prover `metis`, and ZChaff and Z3 proof witnesses are directly reconstructed, as explained above. Sledgehammer is far more powerful than our tactics currently, especially since it does not only send the goal to the prover, but also a well chosen bunch of lemmas from Isabelle/HOL standard library. We could not do this until we handle quantifier instantiation.

The autarkic approach is often considered as being unrealistic considering the complexity of modern automatic solvers. Nonetheless, if the objective is to have a certified automated prover for a proof assistant rather than a very efficient prover for industrial benchmarks, it must be conceivable to implement it directly in the proof assistant. This approach was followed for the SMT solver Ergo implemented and proved correct in Coq [LC09]. Contrary to SMTCoq, it currently cannot handle integer inequalities, but it is complete for the logic QF_UFLIA without them (while SMTCoq cannot be complete without guaranties on the external tools it uses).

7.2.2 Performance

We can compare the performance of our `zchaff` and `verit` tactics with the reflexive tactics `dplln` and `cc` of Ergo. To do so, we use the same formulas that are presented in Section 11.2

of [LC09]:

- for SAT:
  - the pigeon hole problems (see **Example 4.2**);
  - the de Bruijn formulas: $deb_n = \forall x_0, \dots, x_{2n}, (x_{2n} \leftrightarrow x_0) \vee \bigvee_{i=0}^{2n-1} (x_i \leftrightarrow x_{i+1})$
- for EUF:
  - the formulas $FP(n, m, k) = \forall f x, f^n(x) = x \rightarrow f^m(x) = x \rightarrow f^k(x) = x$ which are true for any n, m, k such that k is a multiple of $\gcd(n, m)$
  - the formulas $D_n =$
    $$\forall f, \left(\bigwedge_{i=0}^{n-1} (x_i = y_i \wedge y_i = f(x_{i+1})) \vee (x_i = z_i \wedge z_i = f(x_{i+1})) \right) \rightarrow x_0 = f^n(x_n)$$

Table 7.5 presents the time taken by the tactics, in seconds.

	dplln	zchaff
H_7	28.0	0.2
H_8	262.7	1.2
H_9	-	1.6
H_{10}	-	6.7

	dplln	zchaff
deb_{700}	111.5	0.8
deb_{800}	147.9	1.0
deb_{900}	201.6	1.2
deb_{1000}	260.4	1.5

	cc	verit
$F(13, 5, 8)$	0.5	0.1
$F(25, 13, 1)$	1.3	0.1
$F(25, 15, 5)$	0.5	0.2
$F(25, 24, 24)$	16.9	0.1

	cc	verit
D_5	2.3	0.3
D_8	24.9	1.1
D_{10}	118.7	2.2
D_{15}	-	45.7

Table 7.5: Comparison between Ergo and SMTCoq

We see that our zchaff and verit tactics here clearly outperform dplln and cc. This is not surprising since ZChaff and veriT have more efficient algorithms than Ergo. Note it may be difficult to change Ergo's algorithm since it would involve redoing many correctness proofs; the certificate approach is more flexible here. If we store proof witnesses, zchaff and verit would get faster at rather small storage cost: in our examples, the largest proof witness is 41Mb large for D_{15}.

Regarding other existing Coq tactics, zchaff is far faster than tauto and verit is similar to congruence. However, these latter ones do not solve the same goals, since verit can solve goals including congruence and propositional reasoning, while congruence can deal with inductive data-types.

7.3 About data structures

In this section, we evaluate the importance of native data structures in terms of efficiency. We implemented a small variant of SMTCoq: the only difference is that the state of the main checker (see Section 5.2.1) is not a native array anymore, but a **finite map using AVL trees** (implemented in the Coq library `FMapAVL`). Other structures using native arrays or 31-bit integers are unchanged. We compare it with the standard SMTCoq, on the SAT benchmarks of Section 7.1.2.

Table 7.6 sums up the results. The times are the mean of the times for the 79 benchmarks on which the three tools succeeded, in seconds.

ZChaff			SMTCoq			SMTCoq +`FMapAVL`		
#	%	Time	#	%	Time	#	%	Time
79	52	70.2	79	52	21.5	79	52	79.3

Table 7.6: Evaluation of native data structures

We observe that the variant of SMTCoq was far less efficient than the standard SMTCoq: it took between 3 and 4 more time to check the certificates, which is also a little longer than for ZChaff to produce them. This is a large loss in efficiency, but it remains affordable. Since the representation of states is the place where native data structures theoretically play the more important role, SMTCoq would be sustainable without them at all, but with bad performances.

As a conclusion, at the cost of extending a bit Coq's kernel by a process rather well understood [AGST10], we can significantly improve the efficiency of Coq programs involving lots of computations, in particular accesses and replacements in some structure.

Chapter 8

Future directions

8.1 Spreading SMTCoq

Version 1.0 of SMTCoq was released in 2012 as free software (under the CeCILL-C license) and thus can be freely used to certify ZChaff and veriT answers, or inside other Coq developments. For instance, the prototype implementation of a checker for SAT certificates [AGST10] that was the prelude to SMTCoq has been used to implement a Nelson-Oppen theory combiner in Coq [BCP11].

SMTCoq is now ready to a confrontation with applications external to the Coq or the SAT and SMT community and collaborations with potential users are beneficial to fully understand their requirements. Collaborations that are currently under inspection include:

- a certification of the SMT calls performed by the Liquid Type checker [RKJ08], a type checker for an ML programming language with refined types;
- the use of the tactics in a Coq formalization of some conjectures in chess by Predrag Janicic and Marko Malicović, which involve many easy-to-automate proofs combining propositional reasoning with linear integer arithmetic;
- an integration in the Why3 platform [BFMP11] for deductive program verification, which delegates proof obligations coming from annotated programs to a wide range of provers, in particular SMT solvers.

Considering all these possible applications, we thought about many improvements that have to be brought to SMTCoq to be confronted to the real world.

8.1.1 Generic certificates

We argued that interfacing SMTCoq with a new proof-producing prover only requires to write an Ocaml preprocessor translating the proof witnesses of this new prover into our certificates. This is particularly interesting for solvers developers, which do not need to write Coq code to integrate their tools in SMTCoq.

Nonetheless, we think that it would be really profitable if the SMT community could agree on an output format, in the same way that they currently agree on the input format. The certificates used here correspond to a proposal from the ANR DeCert initiative for a common output format. The work presented in this thesis show that this format has an appropriate level of details to be checked really efficiently in Coq at a small generation cost (see Section 7.1.2). We thus expect other provers to implement it as well.

A longer term perspective is to understand how SMT proof witnesses can be included in more general purpose certificates, like the broad spectrum certificates proposed by Miller [Mil11] or the input format of dedukti [BCH12].

8.1.2 Quantifiers

We also argued that SMTCoq could easily be instrumented with decision procedures for new decidable first-order theories. Nonetheless, in addition to decidable theories, many applications would benefit from *quantifier instantiation*, for instance:

- it is a common way to define new interpreted symbols: universally quantified axioms provide the semantics to these symbols and have to be instantiated to obtain the final proof (this appears often in program verification for instance);
- the Coq tactics could rely on previously proved lemmas which are universally quantified, which is absolutely needed in even small Coq developments.

Handling quantifiers (even only for lemma instantiation) in SMTCoq do require much efforts. The main current limitation is that quantifiers cannot be defined in the set of Booleans, so we need to switch to an interpretation of formulas into the sort of propositions.

It implies that we have to handle classical logic, which is widely used in CNF computation (see Section 5.3.3 and [LC09]). The simplest way to do this would be to assume the axiom of excluded middle. We could also use a translation from classical to intuitionistic logic, as proposed in Ergo [LC09]. However, we can avoid this in many cases of SMT solving: most of the time, the excluded middle is not applied to a Boolean variable, but to a proposition in some theory in which excluded middle is constructively valid (like linear arithmetic).

8.1.3 Tactics

We just explained how handling quantifier instantiation would increase the power of the tactics. The side effect of switching to the sort **`Prop`** would be profitable as well: we are currently limited to goals of the form $b_1 = b_2$ where b_1 and b_2 are Booleans, whereas it is far more common to use propositions in Coq.

The tactics can be improved in many other ways.

First, we can enhance speed efficiency, by storing the certificates. Currently, when interpreting twice the same proof script, everything is computed again. If we store the certificates, we would avoid doing proof search and proof preprocessing twice, which can be very important, especially for proof search. This is for instance done by the Micromega set of tactics.

To make the tactics more useful, we could also consider the possibility to simplify the goal instead of solving it or fail. For instance, this could be very useful to integrate theories for which we do not have decision procedures in Coq: we could just simplify the goal and let the user prove the theory lemmas that cannot be proved automatically in Coq.

Finally, we do not do much encoding of Coq goals before sending them to the automatic provers, which means that the quantifier-free parts of the goals must be direct instances of the

QF_UFLIA logic to be solved by the SMT solver. To be usable, the tactics should encode inductive definitions and some higher-order aspects of Coq into first-order logic, before sending the problem to the automatic solver. This aspect fits in an integration with the Why3 platform, which already handles such encodings, but without formal certification.

8.2 Application to a decision procedure for machine integers

Many decision procedures are based on a SAT encoding of the initial problem: hardware verification and bounded model checking [BCC+99], propositional planning [RGPS08], sudoku [Web05].... Enjoying the power of SAT in Coq allows to write decision procedures for this kind of problems that are reducible to SAT.

In this section, we discuss about the theoretical foundations for a Coq decision procedure for 31-bit integers based on a SAT reduction. Bounded unsigned integers are isomorphic to bit-vectors, whose theory can be encoded in SAT using *bit-blasting* (also called *flattening* [KS08]). It relies on a *binary encoding* of integers: we associate 31 Boolean variables to each integer, representing its digits, and translate the common operators into a set of constraints that the Boolean variables must satisfy.

Example 8.1 *For each integer variable x, we associate $(x_i)_{i\in[\![0,30]\!]}$ Boolean variables (x_0 represents the least significant bit). We can encode the bitwise disjunction as follows:*

$$x = y|z \text{ is encoded as } (x_0 \Leftrightarrow y_0||z_0) \wedge \cdots \wedge (x_{30} \Leftrightarrow y_{30}||z_{30})$$

Constraints associated to bitwise operators are straightforward. For arithmetic operators, usual encodings follow their implementation as *circuits* [KS08], which is experimentally affordable for modern SAT solvers.

A decision procedure for machine integers in Coq would be really useful, since modulo arithmetic proofs often involve burdening extra-work to show that we do not exceed the capacity limitation. There are mainly two possibilities to implement such a decision procedure relying on SMTCoq: either by using SMTCoq as a black-box, or by a deeper integration in the checker.

8.2.1 Using SMTCoq as a black-box

A decision procedure relying on SMTCoq as a black-box works in two steps:

1. encode the problem into a set of SAT constraints using a dedicated tool (this encoding must be certified);

2. use SMTCoq to call ZChaff on the obtained problem.

This method presents the advantage not to require any knowledge about SMTCoq. However, as we took great care to be modular, we think that a deeper integration is possible with only a high level understanding of SMTCoq and is likely to be more efficient.

8.2.2 Deeper integration in SMTCoq

We noticed that step 1, that encodes the problem in SAT, relies on mechanical operations that can be certified using a small checker as presented in Section 5.3. It is indeed rather similar to CNF computation (see Section 5.3.3): we can maximally hash the atoms of this theory and define a bunch of certificates for each operator on 31-bit integers.

Example 8.2 *The clausal tautologies associated with the $y|z$ subterm are:*

$$\frac{}{\neg(y|z)_i \vee y_i \vee z_i}\ \textsc{BorNeg}_i \qquad \frac{}{(y|z)_i \vee \neg y_i}\ \textsc{BorPos1}_i \qquad \frac{}{(y|z)_i \vee \neg z_i}\ \textsc{BorPos2}_i$$

We can also define the corresponding implications:

$$\frac{(y|z)_i}{y_i \vee z_i}\ \textsc{ImmBorPos}_i \qquad \frac{\neg(y|z)_i}{\neg y_i}\ \textsc{ImmBorNeg1}_i \qquad \frac{\neg(y|z)_i}{\neg z_i}\ \textsc{ImmBorNeg2}_i$$

This is an example of a theory checker that can been added without changing the representation of atoms, as mentioned in Section 5.3.7.

8.2.3 Perspectives

Future work consists in implementing this small checker and use it to define a tactic solving goals involving modulo arithmetic on 31-bit integers. We should be careful to develop a generic framework that can be re-used for other decision procedures based on SAT encodings.

Part III

Cooperation with interactive theorem provers: importing HOL Light into Coq

Cooperation between proof assistants is a hot topic, as their number is increasing while their logical frameworks diverge. Even if they have similar theoretical expressivities, they are not practically suited for the same formalizations. An illustration is the Flyspeck Project[1] [HHM$^+$10], aiming at a formal proof of the Kepler Conjecture: some parts involve mathematical analysis, a domain widely explored by HOL Light; others are proved by the check of certificates provided by oracles, which would be efficiently done by computational reflection in Coq.

To translate proofs from one formalism to another, most approaches define an intermediate language for proofs, in the objective that we first translate the source language into this one, which is then translated into the target language. If the intermediate language is generic enough, the process is modular; current propositions often highly mimic the source language, though. A recent work [Mil11] proposes theoretical broad spectrum intermediate certificates, in a long term cooperation perspective; but there is no implementation for the moment.

Another approach, followed in particular by the dedukti project [BCH12], is to make provers communicate inside a back-end universal checker. An interest is that the dedicated checker can be really small and thus the trusted base also.

In this part of the thesis, we present a translation of proofs from HOL Light into Coq, using an intermediate proof format close to natural deduction in Higher-Order Logic. This work is built upon a model of Higher-Order Logic inside Coq and a *good* translation of HOL Light statements, in the sense that they remain intelligible and can be incorporated into further developments made in Coq.

After presenting the paradigms of HOL theorem provers and two possible formats of certificates for Higher-Order Logic (**Chapter 9**), we implement the model of HOL in Coq (**Chapter 10**): a deep embedding (Section 10.3) and an interpretation into Coq (Section 10.4). Upon this generic model, we implemented HOLLIGHTCOQ, an importer of HOL Light theorems into Coq (**Chapter 11**). **Chapter 12** evaluates this importer, both in terms of the quality of the obtained theorems (Section 12.1) and of efficiency (Section 12.2). We finally discuss possible improvements and perspectives (**Chapter 13**).

[1] The progress of this project is available at `http://code.google.com/p/flyspeck`.

Chapter 9

HOL-like theorem provers

9.1 Philosophy and presentation

Like all proof assistants, HOL-like theorem provers (like HOL Light, Isabelle/HOL and HOL4) implement an expressive logical framework in which theorems can be constructed only by the combination of low-level well-understood rules. The way they are guaranteed as well as the logical framework differ from Coq.

9.1.1 Correctness

The kernel of a HOL prover is an abstract data type for theorems, together with a few inference rules allowing to build objects of this abstract type. The typing system of the language in which it is implemented thus guarantees that theorems can only be built at runtime using the few inference rules that were given in the kernel. As a consequence, each session of such a prover starts by evaluating from scratch the definition of all the theorems (it may be possible to save the prover's state to avoid this step, but this is just for a matter of convenience: in the end everything must be checked from the beginning).

A consequence is that there is no actual need for proof terms in HOL provers: when a theorem is built using inference rules, the only important thing to recall is that it actually has the abstract data type of theorems, but we can forget how it has been proved. Contrary to provers based on Type Theory, what matters is the existence of a proof, but we do not care about its low-level content.

9.1.2 Logical framework

HOL-like theorem provers implement the following version of Higher-Order Logic. The language of terms is the simply-typed λ-calculus with prenex polymorphism, with some distinguished constants $c^A \in \mathcal{C}$ (we will explain later what these constants may be):

$$\begin{array}{rcl} A, B & \triangleq & \text{bool} \mid A \to B \mid \alpha \\ t, u & \triangleq & x^A \mid \lambda x^A.t \mid t\ u \mid c^A \end{array}$$

A proposition is a λ-term of type bool. A theorem is a pair made up of a finite set of propositions Γ and a proposition ϕ:

$$\Gamma \vdash \phi$$

obtained by applying some inference rules presented in a natural deduction style.

There are mainly two presentations for the inference rules [LS88], depending on the choice of the distinguished constants: an intentional presentation of Higher-Order Logic based on implication and universal quantification

$$\mathcal{C} = \{\Rightarrow^{\text{bool}\to\text{bool}\to\text{bool}}, \forall^{(\alpha\to\text{bool})\to\text{bool}}\}$$

(implemented for instance in Isabelle/HOL) and an extensional presentation based on equality

$$\mathcal{C} = \{=^{\alpha\to\alpha\to\text{bool}}\}$$

(implemented for instance in HOL Light). For each presentation, it is possible to define the standard logical connectives as shortcuts for some λ-terms and to derive their introduction and elimination rules from the existing rules.

We are more interested in the second presentation, since it is the one used by HOL Light and by proof certificates for HOL (see Section 9.2). The inference rules and definitions of connectives are given in [Har06], we recall six of them on Figure 9.1.

$$\textsc{refl}\ \frac{}{\vdash t =_A t}\ \vdash t : A \qquad \textsc{assume}\ \frac{}{\{p\} \vdash p}\ \vdash p : \text{bool}$$

$$\textsc{beta}\ \frac{}{\vdash (\lambda x.t)\ x = t} \qquad \textsc{abs}\ \frac{\Gamma \vdash s = t}{\Gamma \vdash (\lambda x.s) = (\lambda x.t)}\ x \notin \text{FV}(\Gamma)$$

$$\textsc{mk\_comb}\ \frac{\Gamma \vdash s = t \qquad \Delta \vdash u = v}{\Gamma \cup \Delta \vdash s(u) = t(v)} \qquad \textsc{eq\_mp}\ \frac{\Gamma \vdash p \Leftrightarrow q \qquad \Delta \vdash p}{\Gamma \cup \Delta \vdash q}$$

Figure 9.1: Six inference rules of the Higher-Order Logic

For all the rules, we adopt the same notations as in [Har06]. We give an example of a proof tree in this logical framework.

Example 9.1 *A HOL Light proof of symmetry of equality is:*

$$\textsc{eq\_mp}\ \dfrac{\textsc{mk\_comb}\ \dfrac{\textsc{mk\_comb}\ \dfrac{\textsc{refl}\ \dfrac{}{\vdash (\bullet = \bullet) = (\bullet = \bullet)} \qquad \dfrac{}{y = x \vdash y = x}\ \textsc{assume}}{y = x \vdash (y = \bullet) = (x = \bullet)} \qquad \dfrac{}{\vdash y = y}\ \textsc{refl}}{y = x \vdash (y = y) = (x = y)} \qquad \dfrac{}{\vdash y = y}\ \textsc{refl}}{y = x \vdash x = y}$$

It entails from the inference rules that **terms are not dynamic**: **there is no conversion rule** and β-equivalent terms can only be *proved* equal, as we can see in the BETA rule. There is no computational reflection in HOL (contrary to Coq and other systems based on Martin-Löf Type Theory). From our point of view in this part, where we want to embed a HOL system in Coq, this is rather an advantage: we do not have to write a normalizer for the terms of this language, which would have been very difficult since its termination is not obvious.

In addition to the inference rules, most HOL systems, in particular HOL Light, add to the set of constants the Hilbert operator $\varepsilon^{(\alpha\to\text{bool})\to\alpha}$ and two axioms:

- an axiom asserting that ε is a choice operator: $\vdash Px \Rightarrow P(\varepsilon(P))$;
- an axiom of eta-conversion $\vdash (\lambda x.t\ x) = t$.

This logical framework is non-constructive in at least two aspects. First, we can derive *propositional extensionality*, stating that two equivalent propositions are equal, and *functional extensionality*, which are not derivable in intuitionistic logic. Second, the axiom asserting that it ε is a choice operator makes the system fundamentally classical: in presence of extensionality, this axiom is equivalent to the Axiom of Choice [Dia75].

9.1.3 Constants definitions

Since there is no dynamics, the definition of new terms and types must also be handled differently than in systems with computation like Coq.

Terms definitions are just shortcuts for some λ-terms. Defining $c = t$ where c is some fresh name and $\vdash t : A$ has no free term variables, adds c^A to the set of constants and the new inference rule

$$\frac{}{\vdash c = t}$$

Once again, a constant is provably equal to its definition, but there is no convertibility.

The types that can be defined are non-empty subtypes of existing types, using the schema of specification. If A is a type and $P : A \to \mathrm{bool}$ is a predicate satisfied by at least one term t, then we can define the subtype B (which must be a fresh name) of the terms of A satisfying P:

$$B = \{x : A | P\ x\}$$

The type A may contain type variables, in which case they are considered as arguments of B. Such a definition adds the new type name B to the system, as well as new term constants $\mathrm{rep} : B \to A$ and $\mathrm{abs} : A \to B$ without computational content and the new inference rules:

$$\frac{}{\vdash \mathrm{abs}\ (\mathrm{rep}\ a) = a} \qquad \frac{}{\vdash P\ r = (\mathrm{rep}\ (\mathrm{abs}\ r) = r)}$$

The term rep can be understood as the canonical injection from B to A and abs as the injection from A to B whose behavior is specified only for terms satisfying P.

P needs to be satisfied at least for one term in order to guarantee that **all types are non-empty**. This is fundamental in order to make the ε operator a total function.

Some implementations of HOL also offer the possibility to define inductive data types. In HOL Light, these definitions are just particular cases of the general types definitions presented here, so they are handled at this low level.

We will see that the absence of computation in definitions are again also welcome when formalizing Higher-Order Logic in Coq, since we can freely translate constants into the desired Coq terms to get intelligible statements.

9.1.4 High level rules and tactics

Sections 9.1.2 and 9.1.3 presented the HOL Light kernel. On top of it, more complex rules are derived, standard definition mechanisms are implemented, as well as a large set of tactics, from simple to fully automatized. The important point is that these high level constructions are only combinations of the base rules, definitions and axioms. In consequence, they can be traced back to the kernel primitives, which guarantees correctness and offers the possibility to instrument the proof assistant in order to generate low level certificates, even if there are no built-in proof objects.

9.2 Proof certificates for HOL provers

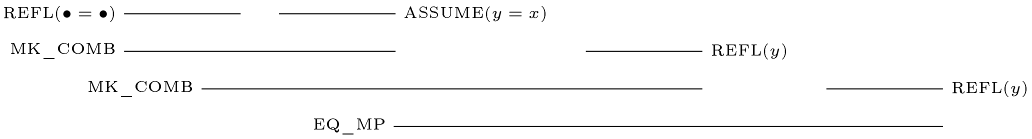
Since a first proposal for HOL by Denney [Den00], different proof formats have been imagined. The more widespread are Proof recording by Obua [OS06] and more recently OpenTheory by Hurd [Hur09]. We are going to present and compare them, illustrated by the proof of symmetry of equality given in **Example 9.1**.

9.2.1 Proof recording

Proof recording [OS06] was introduced to import HOL4 and HOL Light into Isabelle/HOL and has been implemented for both provers[1]. It is currently not developed anymore.

Proof recording records the **skeleton of the proof in natural deduction**. It is a tree of HOL inference rules, but labeled only with the names of the rules and some of their arguments and not the intermediate theorems produced at each step nor the side conditions (like the freshness condition of the ABS rule of Figure 9.1). This is sufficient to automatically build the complete derivation when it exists, by inductively decorating the skeleton with the statements.

Example 9.2 *The skeleton of the proof of* **Example 9.1** *is:*



These proofs are too large to be exported as such. Proof recording introduces two mechanisms to reduce the size of proofs.

The first one is sharing, at the levels of types, terms and theorems: types and terms appearing in a theorem are maximally hashed; concerning theorems, common subproofs are recorded as intermediate lemmas. This drastically reduces the length of certificates, but at the cost of modularity: since the intermediate lemmas are created regarding the whole development, it generates so much dependencies that the certificates coming from different parts of the development cannot be checked separately.

The second (less crucial) improvement for the size of proofs is the possibility to consider more inference rules than the ten rules given by the kernel, in particular the introduction and elimination rules of connectives. Since these rules are more familiar than the kernel rules, they

[1] Proof recording for HOL Light is freely distributed with HOL Light at http://code.google.com/p/hol-light.

are more likely to be used; Obua noticed in consequence that it was a waste of memory to expand them.

In practice, the proofs are exported as XML files, one for each theorem (defined by the user or stemming from sharing), divided into four parts:

- the hashed list of types appearing in the theorem (subtypes are referred to by their locations in the list), enclosed by the `tylist` tag;
- the hashed list of terms appearing in the theorem (subterms are referred to by their locations in the list), enclosed by the `tmlist` tag;
- the skeleton itself, possibly referring to other files by their names;
- the conclusion of the theorem, enclosed by the `tmi` tag.

Example 9.3 *The XML file generated by the proof of* **Example 9.1** *is:*

```
<proof>
  <!-- List of types -->
  <tylist i="4">
    <tyv n="A"/>
    <tyc n="bool"/>
    <tya><tyc n="fun"/><tyi i="0"/><tyi i="1"/></tya>
    <tya><tyc n="fun"/><tyi i="0"/><tyi i="2"/></tya>
  </tylist>
  <!-- List of terms -->
  <tmlist i="7">
    <tmc n="=" t="3"/>
    <tmv n="y" t="0"/>
    <tma f="0" a="1"/>
    <tmv n="x" t="0"/>
    <tma f="2" a="3"/>
    <tma f="0" a="3"/>
    <tma f="5" a="1"/>
  </tmlist>
  <!-- Skeleton -->
  <peqmp>
    <pcomb>
      <pcomb>
        <prefl i="0"/>
        <phyp i="4"/>
      </pcomb>
      <prefl i="1"/>
    </pcomb>
    <prefl i="1"/>
  </peqmp>
  <!-- Conclusion -->
  <tmi i="6"/>
</proof>
```

9.2.2 OpenTheory

The goal of OpenTheory [Hur09] is to share proofs between HOL-like provers and to build a standard library of formalizations done in these provers. An implementation exists for HOL Light[2] and the OpenTheory website[3] makes built libraries available. This is recent software and still in development.

Each library is presented as a package, containing:

- a *theory file*: a text file describing package information (name, dependencies from other packages, ...) and a logical theory $\Gamma \rhd \Delta$ constructed from proof article files and other theory packages, meaning that the theorems in Δ logically derive from the assumptions in Γ (free variables are implicitly universally quantified);
- some *proof article files*: the actual certificates proving $\Gamma \rhd \Delta$.

The package presentation is really meant to be modular and dependencies are clearly established.

Articles contain **commands intended to be processed by a stack-based virtual machine** to build types, terms and theorems: for instance, the command `refl` pops a term t from the stack and pushes the theorem $\vdash t = t$. To provide sharing, the machine uses a **dictionary** containing the terms and theorems that are used multiple times (filled by the command `def`), which can then be referred to by integers (using the command `ref`). The machine finally outputs the two sets Γ and Δ, that are filled respectively by the commands `axiom` and `thm`. A complete description of the commands can be found online[4].

Example 9.4 *Above is the article file generated by the proof of* **Example 9.1**. *When processing it, in the end, we obtain* $\Gamma = \emptyset$ *and* $\Delta = \{y = x \vdash x = y\}$.

```
"="
const
0
def
"→"
typeOp
1
def
"A"
varType
2
def
1
ref
2
ref
"bool"
typeOp
3
def
nil
opType
4
def
nil
cons
cons
opType
5
def
nil
cons
cons
opType
6
def
constTerm
7
def
refl
8
def
7
ref
"y"
2
ref
var
9
def
varTerm
10
def
appTerm
11
def
"x"
2
ref
var
12
def
varTerm
13
def
appTerm
14
def
```

[2] The OpenTheory recorder for HOL Light is available at `http://src.gilith.com/hol-light.html`.

[3] The OpenTheory project is available at `http://www.gilith.com/research/opentheory`.

[4] A description of the OpenTheory article file format is available at `http://www.gilith.com/research/opentheory/article.html`.

```
assume
15
def
appThm
16
def
10
ref
refl
17
def
appThm
18
def
17
ref
eqMp
19
def
14
ref
nil
cons
7
ref
13
ref
appTerm
20
def
10
ref
appTerm
21
def
thm
```

9.2.3 Comparison

The two formats are similar in terms of lengths of certificates. It is a known fact that HOL proofs are likely to be rather huge, since no computation occur inside them.

However, OpenTheory seems more adapted to post-checking: in particular, its modularity allows to check different theories independently from one another. The data structures involved (stacks, arrays) might also be more compact and efficient than the ones involved in Proof recording (trees). Finally, OpenTheory is still being developed and improved, as opposed to Proof recording. The drawback is that the OpenTheory format is still subject to small changes, which implies to change OpenTheory checkers as well.

The development presented in **Chapters 11 and 12** started in 2009, before OpenTheory became a spread standard. The checker is thus based on Proof recording, but the embedding of Higher-Order Logic in Coq is generic (**Chapter 10**). We discuss in Section 13.2 how we could possibly switch to OpenTheory and the possible consequences on performance.

Chapter 10

A model of HOL in Coq

The main perspective of this work is to mix theorems coming from HOL-like theorem provers, in particular HOL Light, with Coq proofs. An important consequence is that theorem statements should not be "obfuscated": they should be written as one would have done directly in Coq, in particular in a *shallow embedding*. However, Wiedijk [Wie07] observed that translating directly HOL theorems into a shallow encoding was difficult to automate, while giving awkward statements. To solve this crucial problem, we propose to use a *deep embedding* as an intermediate step, which will be interpreted into a shallow one to obtain Coq theorems in the end, using the techniques presented in **Chapter 3**. This idea was suggested by Carlos Simpson. The key point is to obtain intelligible statements through a **careful translation of HOL constants**.

Using this now familiar translation process will allow us to check HOL Light's certificates using computational reflection in the next chapter and thus efficiently deal with huge proofs.

This time we do not only embed the **underlying language** of HOL but also its **inference rules**. In addition to clearly establishing the HOL framework, this has two purposes:

- a theoretical one: by proving that the inference rules are correct with respect to the interpretation, we establish in Coq the **correctness of the HOL kernel**;
- a practical one: it is a simple way to handle the **sharing** between the proofs (an aspect which was absent in **Part II**).

As we explained in the previous chapter, the underlying language is a simply-typed λ-calculus with prenex polymorphism. We embed it using a locally-nameless representation, which corresponds exactly to our illustration of deep and shallow embeddings in **Chapter 3**. The additional features are the constants, either distinguished – easily translated to their Coq counterparts – or user defined – which must be translated carefully in order to obtain intelligible Coq theorems, as we will detail in Section 10.2. We need also to understand how to deal with the classical aspects of HOL (see Section 10.1).

Since HOL terms are static, we do not need to implement a normalizer – which would have been difficult in Coq, which requires to be convinced that all functions terminate. Nonetheless, we cannot avoid to implement substitutions and variable freshness (Sections 10.3.2 and 10.3.3), used in some HOL inference rules (Section 10.3.4).

We finally interpret types and terms into Coq and show the adequacy of derivations with respect to this interpretation (Section 10.4).

The model presented in this chapter is generic and does not depend on a particular format for certificates.

10.1 Classical logic

We described a Higher-Order Logic which is fundamentally extensional and classical: we can derive the Axiom of Choice and the extensionality of functions and of propositions. It entails that, to embed this language into Coq, it is not sufficient to use Coq Booleans instead of propositions. Instead, we have three possibilities:

- add these three axioms to Coq; or
- use some translation transforming the theorem statements into some others provable in Coq's logic; or
- try to recognize intuitionistic and intentional facts and import only these.

Since our main objective is to obtain intelligible statements in order to communicate with Coq terms, we choose the first solution: the second would obfuscate a lot the statements and the third one is very difficult to initiate at the cost that some developments cannot be imported. Adding these three axioms does not break consistency and is completely satisfactory for our main application: if parts of a formal developments are written in HOL Light, it means that we assume classical logic, even if other parts are written in Coq.

Practically, adding the three axioms to Coq is done by importing the `FunctionalExtensionality` and `ClassicalEpsilon` libraries (which are parts of the standard distribution of Coq) and assuming propositional extensionality. These axioms have been shown as consistent with the logic of Coq.

10.2 Constant definitions

The more straightforward way to handle types and terms definitions in the importation would be to add the same definitions to Coq. For instance, when a new term constant c is defined with content t, we could add in Coq a new constant hol_c with content $|t|_{\mathcal{I}}$.

This approach does not fulfill our objective to obtain intelligible statements: it would duplicate already existing Coq objects into new ones nobody knows about, possibly with other names and definitions. For instance, for any proposition A, $\neg A$ would be translated into:

```
((fun f ⇒ f A′ (fun p ⇒ p = (fun _ ⇒ ((fun p ⇒ p) = (fun p ⇒
   p))))) = (fun f ⇒ f ((fun p ⇒ p) = (fun p ⇒ p)) ((fun p ⇒ p)
   = (fun p ⇒ p)))) = A′
```

where `A′` is the translation of A, which is completely unreadable, whereas we simply expect `∼A′`!

To avoid this pitfall, our solution is to somehow forget about the HOL definition of constants and allow the user to give custom translations, provided that for every constant, its type is the translation of the type of the constant. With this requirement, we obtain readable statements and $\neg$ is indeed translated into $\sim$. We can even go further: this approach gives the possibility to translate for instance HOL Light unary integers into Coq binary integers, which may be interesting (see Section 12.1). This method is safe if the user also gives a proof that the custom translation he gave is equivalent to the HOL definition.

The definitions of constants thus do not appear in the deep embedding, but only their names: they are embedded like variables. The difference appears during the translation: variables are translated using some environment and constants using another environment fulfilled with the Coq terms given by the user. We will see in the next chapter how the user practically gives custom translations.

10.3 Deep embedding

We are now ready to define the deep embedding, before giving its interpretation. The deep embedding for types and terms is similar to the one presented in Section 3.2, with additional constructors for constants, either distinguished or defined by the user. For efficiency reasons, variables (and constants) are not represented using nat, but binary natural numbers positive. We recall that constants' definitions are not given in the deep embedding.

10.3.1 Types, terms and sets of terms

A previously defined type is accessed through its name and the list of its arguments (its definition is used only when interpreting). Lists of types are mutually inductively defined with types, to ease the generation of Coq induction principles.

```
Inductive type : Type :=
| TVar : positive → type                    (* Type variables *)
| TDef : positive → list_type → type
             (* User defined constants, possibly applied *)
| Bool : type
                                  (* The type of propositions *)
| Arrow : type → type → type
                                      (* The type of functions *)

with list_type : Type :=                    (* Lists of types *)
| Tnil : list_type
| Tcons : type → list_type → list_type.
```

The distinguished term constants are the ε operator and equality.

```
Inductive term : Type :=
| Dbr : nat → term                         (* Bound variables *)
| Var : positive → type → term             (* Named variables *)
| Equ : type → term                                (* Equality *)
| Eps : type → term                          (* Hilbert epsilon *)
| Def : positive → type → term  (* User defined constants *)
| App : term → term → term                      (* Application *)
| Abs : type → term → term.                      (* Abstraction *)
```

Two named variables with the same name but with different types are considered different and similarly for user defined constants.

We lighten the notations of function type and equality:

```
Notation "A '⟶' B" := (Arrow A B).
Notation equ := (fun A a b ⇒ App (App (Equ A) a) b).
```

Since HOL judgments are relations between a finite set of propositions and one proposition, we also define `hyp_set`, the type of finite sets whose elements have type `term`, built as an instance of the `FSets` functorial library. It automatically generates standard set operations, like the union of two `hyp_sets` (written `hyp_union`) or the empty `hyp_set` (written `hyp_empty`).

We also define the type inference function and the `wt` relation of Section 3.2.2.

10.3.2 Substitutions

Type substitutions straightforwardly replace a type variable with a type inside a term. There are two kinds of term substitutions in a locally nameless STLC [ACP+08]:

- *named substitutions* that replace a named variable with a term;
- *abstraction opening* and *variable closing*, which respectively instantiate a bound variable with a term and replace a named variable with a bound variable.

The implementation of these substitutions is detailed in [ACP+08] and is rather easy in a locally nameless representation: we avoid both capture variable that would occur in a named representation and the lift needed in a de Bruijn representation. Here we consider *parallel* type and named substitutions, whose definition directly entails from pointwise substitutions. In consequence, we only give the prototype of the Coq functions.

Parallel type substitutions

Parallel type substitutions are defined as a list of pairs containing the name of a variable and the type by which it is substituted:

```
Definition substitution_idt := list (positive * type).
```

The `subst_idt` function applies it to a term:

```
Definition subst_idt : term → substitution_idt → term.
```

Its extension to sets of terms is called `hyp_subst_idt`.

Parallel named substitutions

Parallel named substitutions are defined as a list of pairs containing a variable (its name and its type) and the term by which it is substituted:

```
Definition substitution_idv := list ((positive * type) * term).
```

To be well defined, a named substitution should only substitute a variable x^A by a closed term of type A. This is verified by the `wf_substitution_idv` function:

```
Definition wf_substitution_idv : substitution_idv → bool.
```

The `subst_idv` function applies a named substitution to a term:

```
Definition subst_idv : term → substitution_idv → term.
```

Its extension to sets of terms is called `hyp_subst_idv`.

Abstraction opening and variable closing

The prototypes of abstraction opening and variable closing are:

```
Definition close : term → positive → type → term.
Definition open : term → term → term.
```

`close t x A` abstracts x^A in the term t. `open t u` replaces in t every de Bruijn index 0 with u.

10.3.3 Variable freshness

We also define a Boolean function checking if some variable x^X does not appear free in the term t:

```
Definition is_not_free (x: idV) (X: type) (t: term) : bool.
```

Its extension to sets of terms is called `hyp_is_not_free`.

10.3.4 Derivations

HOL judgments $\Gamma \vdash \phi$ can be deeply embedded as an inductive data type relating the set of propositions Γ to the proposition ϕ:

```
Inductive deriv : hyp_set → term → Prop :=
| Drefl : forall t A, wt nil t A → deriv hyp_empty (heq A t t)
| Dassume : forall t, wt nil t Bool → deriv (hyp_singl t) t
| Dbeta' : forall t u X A, wt (X::nil) t A → wt nil u X → deriv
   hyp_empty (heq A (App (Abs X t) u) (open t u))
| Dabs' : forall h u v (L: list positive) A X, (forall x,
    (forallb (fun y ⇒ y != x) L) → deriv h (heq A (open u (Var x
    X)) (open v (Var x X)))) → deriv h (heq (X ⟶A) (Abs X u)
    (Abs X v))
| ... (* Other HOL rules and axioms except INST and INST_TYPE *)
| Dweak : forall h1 h2 t, hyp_subset h1 h2 → deriv h1 t → deriv
   h2 t.
```

This type does not exactly model the HOL logical framework. First-order rules, like REFL and ASSUME, are faithfully encoded: we just need to type check some terms to ensure that we derive only well-typed propositions. However, rules involving abstraction, like BETA and ABS, are not straightforward. One would rather expect:

```
| Dbeta : forall x t X A, wt nil t A → deriv hyp_empty (heq A
    (App (Abs X (close t x X)) (Var x X)) t)
| Dabs : forall h x u v A X, hyp_is_not_free x X h → deriv h
    (heq A u v) → deriv h (heq (X ⟶A) (Abs X (close u x X))
    (Abs X (close v x X)))
```

We prefer the first formulation since it confers `deriv` a **stronger induction principle**, which is required to prove the adequacy with the semantics in Section 10.4.2. In particular, the cofinite quantification [ACP$^+$08] over any list not containing x in the `Dabs'` rule will supply as many fresh variables as we need. The cofinite definitions are equivalent to the "paper presentation" of HOL rules [ACP$^+$08].

The second difference is that we added a weakening rule. It allows us to finally derive Dbeta and Dabs from Dbeta' and Dabs'. In this context, the substitution rules are also derivable [ACP+08]:

```
Lemma Dinsttype : forall h c S, deriv h c → deriv
   (hyp_subst_idt h S) (subst_idt c S).
Lemma Dinst : forall h c s, wf_substitution_idv s → deriv h c →
   deriv (hyp_subst_idv h s) (subst_idv c s).
```

which is why they do not appear in the definition of the inductive deriv.

This data type is extremely wordy and cannot be reasonably considered as proof certificates. The Proof recording format corresponds to the skeleton of this type, in which all the information that can be computed during checking do not appear, as we will see in Section 11.1.

10.4 Interpretation

10.4.1 Types, terms and sets of terms

Chapter 3 explained how to write Coq functions interp_type and interp_aux interpreting the types and the terms of this language. In the particular embedding of HOL presented in this chapter, we dispose of four environments:

- two environments $\mathtt{I}_V : \mathtt{E}_V$ and $\mathtt{i}_v : \mathtt{e}_V$ respectively interpret type and term variables;
- two others $\mathtt{I}_C : \mathtt{E}_C$ and $\mathtt{i}_c : \mathtt{e}_c$ respectively interpret type and term constants, by associating to them the Coq terms specified by the user (see Section 10.2).

Bool is translated into **Prop**, as explained in Section 10.1. Equ is translated into Coq's standard equality (which behaves as HOL Light's equality since we added classical axioms) and Eps is translated into the ε operator given in the library ClassicalEpsilon.

We must take care of the fact that all the types are non-empty to faithfully translate the ε operator. This is done by interpreting a HOL type not as a Coq type, but as a dependent pair containing a Coq type and one element of this type:

```
Record type_translation : Type := mkTT
  { ttrans :> Type;
    tinhab : ttrans }.
```

The type of interp_type and interp_aux are thus:

```
Definition interp_type : E_V → E_C → type → type_translation.
Definition interp_aux : forall (I_V:E_V) (I_C:E_C),
  e_V → e_C → context → term →
  option {A:type & interp_context g → interp_type I_V I_C A}.
```

Propositions are locally closed terms of type Bool. They can be interpreted directly to Coq propositions by the following function:

```
Definition has_sem I_V I_C i_V i_C (t: term) : Prop :=
  match interp_aux I_V I_C i_V i_C nil t with
    | Some (existT Bool b) ⇒ b (interp_nil I_V I_C i_V i_C)
    | _ ⇒ False
  end.
```

which is a variant of the interp function interpreting only propositions. It can be extended to sets of propositions, which are interpreted as the conjunction of all their elements:

```
Definition hyp_sem I_V I_C i_V i_C (h: hyp_set) :=
  hyp_For_all (has_sem I_V I_C i_V i_C) h.
```

10.4.2 Adequacy of derivations

Using the previous lemma, we can finally establish the adequacy of the derivations with respect to the interpretation:

```
Theorem deriv_interp : forall (h: hyp_set) (t: term),
  deriv h t → forall I_V I_C i_V i_C,
  hyp_sem I_V I_C i_V i_C h → has_sem I_V I_C i_V i_C t.
```

The proof of this theorem is an induction on the proof of `deriv h t` and relies on:

- the definition of the interpretation: for instance, the adequacy of the EQ_MP rule is trivial since $|p \Leftrightarrow q|_{\mathcal{I}} \equiv |p|_{\mathcal{I}} \leftrightarrow |q|_{\mathcal{I}}$;
- the adequacy of the `close` and `open` substitutions.

Remark 4 *The conclusion of the adequacy theorem* is *the shallow Coq version of the HOL theorem $h \vdash t$. In consequence, to obtain intelligible Coq theorems in the end, we just apply the adequacy theorem to a proof of* `deriv h t`*, obtained by computational reflection as we will see in the next chapter, and the environments.*

Chapter 11

HOLLIGHTCOQ: Coq theorems built from HOL Light proofs

Using the model of the previous chapter, we implemented HOLLIGHTCOQ, an automatic importer of HOL Light theorems into Coq based on Proof recording certificates. Figure 11.1 depicts its architecture.

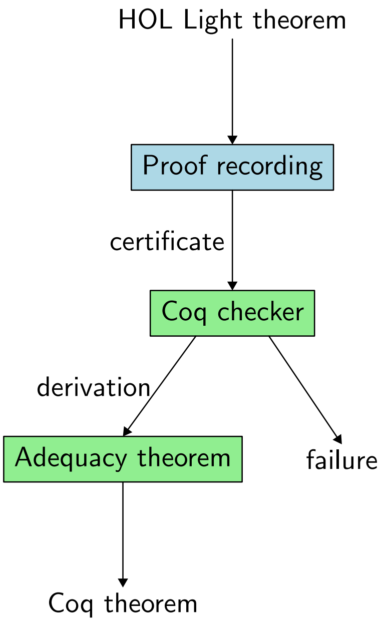


Figure 11.1: Architecture of HOLLIGHTCOQ

Proof recording was instrumented to generate Coq files (instead of XML files) containing:

- a bunch of lemmas whose statements are of the form `deriv h t` for some closed `h` and `t`, and whose proof consists in an application of the checker of Section 11.2 to a certificate in the format presented in Section 11.1;
- the environments to interpret variables and constants (see Section 11.4).

The user can then apply the adequacy theorem to some automatically generated lemmas and the environments to establish the desired theorems (see **Remark 4**).

The variant of Proof recording directly generating Coq files is distributed with HOL Light source code[1]. HOLLIGHTCOQ is freely available[2].

After presenting the Coq version of Proof recording certificates (Section 11.1) and how they can be translated into correct derivations when possible (Section 11.2), we describe the Coq files that are generated by HOLLIGHTCOQ (Sections 11.3 and 11.4).

11.1 Coq version of Proof recording certificates

The certificates are a purified version of HOL Light derivations, containing only the skeleton:

```
Inductive proof : Type :=
| Prefl : term → proof
| Passume : term → proof
| Pbeta : positive → type → term → proof
| Pabs : proof → positive → type → proof
| ...  (* Other HOL Light rules *)
| Poracle : forall h t, deriv h t → proof.
```

The additional Poracle constructor refers to results that were previously established, by the user or by sharing.

11.2 Transformation into derivations

This structure is too loosely to guarantee the correctness: some objects of type proof do not correspond to actual proofs. For instance, Passume (Dbr 0) is not valid since Dbr 0 is not locally closed. It is however sufficient to reconstruct a complete derivation when it exists, by making the necessary verifications. This is the job of the proof2deriv function:

```
Fixpoint proof2deriv (p: proof) : option (hyp_set * term) :=
  match p with
    (* rule REFL *)
    | Prefl t ⇒
      match infer nil t with
        | Some A ⇒ Some (hyp_empty, equ A t t)
        | None ⇒ None
      end

    (* rule ASSUME *)
    | Passume t ⇒
      match infer nil t with
        | Some Bool ⇒ Some (hyp_singl t, t)
        | _ ⇒ None
      end
```

[1] HOL Light source code is available at http://code.google.com/p/hol-light.

[2] HOLLIGHTCOQ is freely available at http://www.lix.polytechnique.fr/~keller/Recherche/hollightcoq.html.

```
    (* rule BETA *)
    | Pbeta x X t ⇒
      match infer nil t with
        | Some A ⇒ Some (hyp_empty,
          equ A (App (Abs X (close t x X)) (Var x X)) t)
        | None ⇒ None
      end

    (* rule ABS *)
    | Pabs q x X ⇒
      match proof2deriv q with
        | Some (App (App (Equ A) t1) t2, t) ⇒
          if hyp_is_not_free x X h then
            Some (h, equ (X⟶A) (Abs X (close t1 x X)) (Abs X
                (close t2 x X)))
          else
            None
        | None ⇒ None
      end

    (* Other HOL Light rules *)
    | ...
  end.
```

which is proved to be correct when it actually returns a judgment:

```
Lemma proof2deriv_correct : forall p,
  match proof2deriv p with
    | Some (h,t) ⇒ deriv h t
    | None ⇒ True
  end.
```

11.3 Generation of lemmas

The generated Coq files are divided into two parts:

- the first part contains the definition of types and terms appearing in the lemmas, using maximal sharing like in the original Proof recording;
- the remaining (and largest part) contains, for each lemma $h \vdash t$: the definition of the set h, the definition of t and the lemma of type `deriv h t` whose proof relies on `proof2deriv_correct`.

Example 11.1 *The file generated by the Coq exportation of* **Example 9.1** *looks like:*

```
Definition hollight_type_0 := Bool.
Definition hollight_term_0 := Equ hollight_type_0.
Definition hollight_term_1 := Dbr 0.
Definition hollight_term_2 := App hollight_term_0
   hollight_term_1.
Definition hollight_term_3 := App hollight_term_2
   hollight_term_1.
Definition hollight_term_4 := Abs hollight_type_0
   hollight_term_3.
...   (* Many other HOL types and terms *)


Definition hollight_0_h := nil.

Definition hollight_0_t := hollight_term_106.

Lemma hollight_0_lemma : deriv hollight_0_h hollight_0_t.
Proof.
  vm_cast_no_check (proof2deriv_correct (Prefl
     hollight_term_105)).
Qed.


Definition hollight_SYM_h := (cons hollight_term_103 nil).

Definition hollight_SYM_t := hollight_term_107.

Lemma hollight_SYM_lemma : deriv hollight_SYM_h hollight_SYM_t.
Proof.
  vm_cast_no_check (proof2deriv_correct (Peqmp (Pmk_comb
     (Pmk_comb (Prefl hollight_term_101) (Passume
     hollight_term_103)) (Poracle hollight_0_lemma)) (Poracle
     hollight_0_lemma))).
Qed.
```

The tactic `vm_cast_no_check t` *solves the current goal without any verification, but tells the system to use the VM reduction when verifying that* `t` *is indeed the proof of the goal during the* **Qed** *step. The lemmas are proved by computational reflection and the length of the proof is the length of the* Proof recording *certificate.*

We observe that proofs that are used many times can be proved only once, and referred to using the `Poracle` *constructor.*

When exporting non trivial developments, we cannot generate one single Coq file, which would be too large to be fed to Coq's compiler. We studied the possibility to generate smaller files as independent as possible from one another, but the sharing performed by Proof recording makes it impossible to check different parts independently (see Section 9.2.1). We thus decided to linearly generate the lemmas such that dependencies are respected, with the empirical heuristic of one hundred lemmas a Coq file. We will see in the next chapter that this absence of modularity is a drawback for memory consumption, but it is unavoidable without at least changing the sharing paradigm.

11.4 Generation of environments

An additional Coq file is generated to define the interpretation environments, in particular using the text file given by the user to interpret constants.

11.4.1 Variables

Conventionally, the meaning of free variables in theorems is to be implicitly universally quantified. To reproduce this in Coq, we define a Coq parameter for each term and type variable, which will be the interpretation of the HOL Light variable.

An improvement would be to use the section mechanism of Coq to avoid the addition of top-level parameters.

11.4.2 Constants

We explained in Section 10.2 how to safely interpret constants, by translating them to Coq terms of the same type which can be proved to be equivalent to the original definition. This is not completely implemented yet in the current version of HOLLIGHTCOQ: the user must give well-typed custom interpretation of constants, but axioms are added to justify the equivalence. We discuss in Section 13.1.1 how to faithfully implement constant translation without adding axioms.

The interpretation for constants is given in a text file containing pairs of lines: the first line is the HOL Light name of the constant, and the second line is a Coq term which is its interpretation. If the constant is a type, the Coq term must have type `type_translation` (see Section 10.4.1).

Example 11.2 *In HOL Light, the unit type is written 1 and its element is written `one`. To translate them into their Coq counterparts (respectively `unit` and `tt`), the file must contain:*

```
...
1
mkTT unit tt
one
tt
...
```

We provide such a file to translate HOL Light's standard library.

Chapter 12

Evaluation of HOLLIGHTCOQ

We evaluate HOLLIGHTCOQ on two aspects that were at the heart of this work:

- the possibility to make HOL Light and Coq theorems interact inside Coq;
- the efficiency of the whole process (lemmas generation and compilation), in terms of time and memory.

These two goals must be achieved to check in Coq the current formalization of the Flyspeck project.

The quantitative experiments have been conducted in 2010 on a virtual machine that is installed on a DELL server PowerEdge 2950, with 2 processors Intel Xeon E5405 (quad-core, 2GHz) and 8 Gb RAM, with CentOS (64 bits).

12.1 Qualitative interaction between HOL Light and Coq

The care we took when interpreting constants offers the possibility to obtain intelligible Coq theorems that can easily interact with other theorems proved by hand in Coq.

We give an example from integer arithmetic. Coq and Ssreflect provide vast libraries about arithmetic for unary integers, but not for binary integers. We propose to import a HOL Light arithmetic theorem, to interpret it in the set of Coq binary integers and to prove a corollary that would have required efforts using only Coq libraries.

We import the theorem MOD_EQ_0 from HOL Light, mapping HOL Light's constants as stated in Table 12.1. HOLLIGHTCOQ automatically generates source files containing the deep version of theorem MOD_EQ_0, called hollight_MOD_EQ_0_lemma. Applying it to the adequacy theorem, we can prove the shallow version of MOD_EQ_0 in Coq:

```
Theorem MOD_EQ_0 : forall x x0 : N, x0 <> 0 →
  x0 | x = (exists a : N, x = a * x0).
Proof.
  vm_cast_no_check (deriv_interp hollight_MOD_EQ_0_lemma I_V I_C
    i_V i_C (hyp_sem_empty I_V I_C i_V i_C)).
Qed.
```

where a | b is the usual notation to say "a divides b":

Environments	$\mathtt{I}_C$	$\mathtt{i}_C$			
HOL Light	num	+	*	DIV	MOD
Coq	N	Nplus	Nmult	Ndiv	Nmod

Table 12.1: Mapping HOL Light definitions about integers into their Coq binary representations

```
Notation "a|b" := (Nmod b a = 0).
```

We can combine it with Coq theorems like Nmult_assoc to prove the following corollary:

```
Lemma div_mult : forall a b, a <> 0 → a | b → forall k, a | k*b.
```

The proof is only five lines long.

Here, the possibility to interpret HOL Light integers as Coq *binary* integers augmented Coq with a new theorem, which would not have been the case if HOL Light integers were translated into nat once and for all.

12.2 Performance

We tested the time and memory performance of HOLLIGHTCOQ on three HOL Light developments:

- HOL Light's standard library (what is loaded by default when launching HOL Light);
- a proof of consistency and soundness of HOL Light in HOL Light [Har06], distributed with HOL Light in the Model directory;
- the elementary linear algebra tools developed in Multivariate/vectors.ml.

The results are summed up in Table 12.2. For each benchmark, we report the number of theorems that were exported, the number of lemmas generated by sharing, the time to interpret theorems and record their proofs in HOL Light, the time to export theorems to Coq, the time of compilation in Coq, the size of the generated Coq files, the maximal virtual memory used by Ocaml during exportation and the maximal virtual memory used by Coq during compilation.

Bench.	Number		Time			Memory		
	Theorems	Lemmas	Rec.	Exp.	Comp.	H.D.D.	Ocaml	Coq
Stdlib	1,726	195,317	2 min 30	6 min 30	10h	218 Mb	1.8 Gb	4.5 Gb
Model	2,121	322,428	6 min 30	29 min	44h	372 Mb	5.0 Gb	7.6 Gb
Vectors	2,606	338,087	6 min 30	21 min	39h	329 Mb	3.0 Gb	7.5 Gb

Table 12.2: Quantitative evaluation of HOLLIGHTCOQ

12.2.1 Time and memory in Coq

As we explained, the sharing of Proof recording has the major drawback to lead to a complete blow-up of the number of exported statements, as the first two columns of Table 12.2 attest. When the generated Coq files are compiled, all these statements need to be kept in memory because all the theorems depends on one another, which explains why Coq's compiler requires so much virtual memory. Note that the proofs (and thus the certificates) are not kept in memory, only the statements.

It has also an impact on the time of Coq's compilation: it thus cannot be less than quadratic in the number of Coq files, since compiling the $(n+1)^{\text{th}}$ file imports files 1 to n.

The other operations that are expensive in time are:

- the parsing of the proof objects;
- the evaluation of the computationally reflexive proofs.

The first item could be avoided if we do not generate intermediate Coq files, as we will discuss in Section 13.1.2. Concerning this last item, it is important to notice that Coq's virtual machine can run such a big computation. This is another example showing that computational reflection is appropriate to import large proofs in Coq.

As long as it remains reasonable, the compilation time is not as restrictive as memory consumption, since the incoming theorems have to be compiled once and for all. Moreover, it requires far less human work to automatically export some theorems and compile it with our tool than to prove them from scratch in Coq.

Memory is a large limitation for users though, since they need to import in memory all the Coq files even to use only the last theorem. It would be convenient to be able not to load the intermediary lemmas, but it does not seem possible with our present proof objects implementation.

In other words: this kind of sharing limits the memory consumed by *proof objects*, but the resulting number of *statements* then becomes a problem.

12.2.2 Memory in Ocaml

The virtual memory needed to record and export HOL Light's theorem is also rather huge. The fact that proofs are kept is not the only limiting factor: during exportation, we create big hash-tables to perform hash-consing and to remember theorem statements. If we keep the present proof format, we definitely would have to reduce the extra-objects we construct for exportation.

12.3 Conclusion of the experiments

The current version of HOL Light already allows to import non trivial HOL Light developments into Coq intelligible theorems, but at a virtual memory cost which corresponds to the limit affordable by nowadays computers. Larger developments, in particular Flyspeck, cannot be quantitatively handled.

Hence the next step is to switch to a more compact proof format. We think in particular of OpenTheory [Hur09], whose format seems well adapted to an efficient proof checking by computational reflection like the one used by SMTCoq, as we will discuss in Section 13.2. It is important to notice that changing the format of certificates only affects what was presented in **Chapter 11**, since we took care to design generic model and interpretation of Higher-Order Logic in Coq in **Chapter 10**.

Chapter 13

Future directions

We first propose possible improvements that are independent from the format of certificates, before discussing the work required to switch to OpenTheory certificates.

13.1 Improvements of the current version of HOLLIGHTCOQ

13.1.1 Generation of constants environments

The idea to let the user prove the equivalence between its interpretation of constants and HOL Light's definitions must be implemented to avoid adding axioms when importing HOL Light theorems into Coq (see Sections 10.2 and 11.4.2). To be able to do this, we need to understand how these equivalence proofs, which are stated in shallow terms, will be integrated in the whole system.

Our proposition consists in lifting Coq propositions that correspond to the interpretation of HOL Light terms at the deep level. A possible implementation would be to add a constructor to the inductive `deriv`:

```
Inductive deriv I_V I_C i_V i_C : hyp_set → term → Prop :=
| ...    (* The same constructors as before *)
| Dshallow : forall t, has_sem I_V I_C i_V i_C t → deriv nil t.
```

The non minor drawback that appears is that `deriv` now requires additional parameters: the environments used for interpretation. The adequacy theorem remains valid, but the environments must be known in the derivation:

```
Theorem deriv_interp :
  forall (h: hyp_set) (t: term) I_V I_C i_V i_C,
  deriv I_V I_C i_V i_C h t → hyp_sem I_V I_C i_V i_C h →
  has_sem I_V I_C i_V i_C t.
```

There is not a clear distinction between the syntactically generated lemmas and their interpretations in Coq anymore. However, it should not affect computation: the `proof2deriv` function still does not need to know about the interpretation. This is what makes us think this would be a reliable solution to safely handle constants.

13.1.2 Removal of intermediate files

The fact that HOLLIGHTCOQ generates Coq files that are then compiled by Coq is an important loss of efficiency: it is space consuming and the proofs need to by parsed by Coq. Since HOL Light and Coq are both written in Ocaml, it would be easy to have a "synchronized" communication: Coq terms can be directly generated by Proof recording at the Ocaml level using the method described in 3.3.2 and implemented in SMTCoq.

13.1.3 Working with deep embeddings in Coq

A longer term perspective is to establish a generic way to define deep embeddings of logical frameworks or programming languages into proof assistants like Coq. Each time, the same work has to be redone to define standard function on terms like substitutions or variable freshness. A first step towards genericity could be to provide for Coq a tool similar to Cαml[1] for Ocaml, that allows to manipulate deep terms modulo α-conversion in a rather concise style and automatically defines substitution functions.

13.2 Switching to OpenTheory

OpenTheory proof certificates correspond to the state-of-the-art in terms of certificates dedicated to proofs in Higher-Order Logic, and as such are good candidates to improve HOLLIGHTCOQ's efficiency, modularity and expressivity (since certificates can be generated by other HOL-like provers than HOL Light). Holide[2] uses this format to import HOL Light and HOL4 theorems into dedukti [BCH12].

13.2.1 Light approach

Since we took care to be modular, a simple approach to switch to OpenTheory consists in writing an Ocaml preprocessor that translates an OpenTheory certificate into a bunch of Proof recording certificates (since a single OpenTheory witness potentially establishes several theorems).

It would already bring the OpenTheory modularity to HOLLIGHTCOQ: each library can be exported and checked separately; the dependencies are handled by checking that the assumptions of one library are established by the conclusions of another one. We would also benefit from the fact that OpenTheory recorders have been implemented for other provers than HOL Light.

13.2.2 Deeper approach

To also benefit from the efficiency inherent to the OpenTheory format, we need to change Coq's checker as well (but the embedding of Higher-Order Logic can remain the same). In this approach, OpenTheory certificates can be handled in Coq in a similar way than Proof recording certificates:

- define a Coq representation of the certificates (similar to `proof`);

[1] Cαml is available at `http://cristal.inria.fr/~fpottier/alphaCaml`.

[2] Holide is available at `https://www.rocq.inria.fr/deducteam/Holide/index.html`.

- define a Coq function that reconstructs the logical theory $\Gamma \rhd \Delta$ from a certificate (similar to `proof2deriv`);
- prove its correctness (similar to `proof2deriv_correct`).

The adequacy theorem then must be generalized to a logical theory $\Gamma \rhd \Delta$ instead of only one derivation, but this is easy. The difficult part is the transformation of `proof2deriv` and `proof2deriv_correct`: if these function and proof were straightforward to define for Proof recording, since the certificates were close to the derivations, their implementations require more work for OpenTheory. In particular, `proof2deriv` for Proof recording was a simple traversal of an inductive data type, whereas its OpenTheory variant consists in a stack-based machine relying on a dictionary.

Even if it is more difficult to establish the correctness of a stack-based virtual machine, `proof2deriv` is likely to be more efficient than before: the data structures that are involved can be encoded using native arrays, since the maximal lengths of the stack and the dictionary can be preprocessed.

Conclusion

This thesis explored the communication between Coq and external provers, namely SAT and SMT solvers and interactive theorem provers based on Higher-Order Logic, in the perspective of a safe and efficient cooperation that uses the strengths of the different tools. The communication is based on a careful embedding of the involved logical frameworks in Coq, that allows at the same time automation, sharing and intelligibility, and computational reflection, which efficiently and automatically proves concrete theorems in the end.

This work ended up in two implementations: SMTCoq, which carries out efficient certified *a posteriori* checkers for SAT and SMT solvers and automated Coq tactics calling these solvers, and HOLLIGHTCOQ, an importer and checker for HOL Light theorems into Coq.

This work clearly establishes that a cooperation between Coq and external tools through proof witnesses is possible: the expressivity and the computational power of Coq together with its user interface makes it an excellent system to model logical frameworks, implement certificates checkers and establish their correctness, with high guarantees of safety. They also make clear that it is profitable for all the tools: Coq gains in automation and external provers can benefit from safety and computation.

Indubitably, SMTCoq and HOLLIGHTCOQ must be improved in many ways to be fully usable. Nonetheless, this thesis establishes a general methodology for a communication between proof assistants and external provers, principally based on two ingredients: computation and certificates.

Computation in proof assistants

Computation in Coq is employed for two different purposes:

- computational reflection is used to check that certificates actually prove deeply embedded theorems;
- the interpretation of the deeply embedded theorems into Coq terms is performed by a Coq function.

It is the combination of the two that give an efficient communication in which Coq has a global “understanding” of the other tools. It also offers the possibility to extract the different components, in particular the checkers.

Other developments use computation in Coq to formalize properties about programs that can actually be executed [Gon08, Ler09] and extracted [Ler09, Dar09, Bar99].

Undoubtedly, computation enlarges the proof assistant and in particular its trusted base: its correctness relies on the strong normalization of the reduction, which is particularly nontrivial in presence of fixpoints and coinduction. Nonetheless, it confers a high flexibility and a large set of possibilities to the interactive theorem prover, of which only a small part was made visible in this thesis.

Cooperation through certificates

In the early days of computers, the most spread image of the future of computer installations was simple terminals all over the world connected to one single omniscient computer, as suggested by Grosh's law and Isaac Asimov and Douglas Adams novels. The reality appeared to be rather different, with billions of small computers distributed over all kinds of systems, some of them being general and others being dedicated to a single task and thus pretty good at this task (this phenomenon is called *downsizing* in France).

When talking about integrating a prover into another one, the first model rather corresponds to an autarkic approach: one single tool learns a new way of proving theorems in order to be self-contained. Conversely, the skeptical approach seems more suited to a distributed world and can benefit from the experience of the dedicated tools in particular domains. In this representation, certificates correspond to protocols required for participants to understand each other.

The work presented here established that the distributed model was reliable and robust for theorem proving. At the cost of completeness and as long as *a posteriori* certification is sufficient, a cooperation through certificates is adaptable enough to exploit the advantages of all the participants, once a good balance between generation and checking has been found. It is also less affected by minor changes. We presented all along the documents other works based on proof witnesses, sometimes with different approaches.

However, even if we took care to make the largest part of the implementations independent from the choice of certificates, they directed nontrivial fragments of the developments and changing their formats implies to change these fragments as well. Besides, since we considered two different kinds of theorem provers, we noticed that the certificates are rather different in spite of the fact that they are both based on a similar first-order framework. A perspective would be to find more general certificates and a starting step may be to unify SMTCoq and HOLLIGHTCOQ as most as possible. In a different approach, we can also imagine generating at the same time formats for certificates and dedicated checkers for them at a meta-level.

Bibliography

[ACHA90] Stuart F. Allen, Robert L. Constable, Douglas J. Howe, and William E. Aitken. The Semantics of Reflected Proof. In *LICS*, pages 95–105. IEEE Computer Society, 1990.

[ACP$^+$08] B.E. Aydemir, A. Charguéraud, B.C. Pierce, R. Pollack, and S. Weirich. Engineering formal metatheory. In G.C. Necula and P. Wadler, editors, *POPL*, pages 3–15. ACM, 2008.

[AF06] N. Ayache and J.-C. Filliâtre. Combining the Coq proof assistant with first-order decision procedures. Unpublished notes, March 2006.

[AFG$^+$11a] Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses. In Jouannaud and Shao [JS11], pages 135–150.

[AFG$^+$11b] Mickaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Wener. Verifying SAT and SMT in Coq for a fully automated decision procedure. In *PSATTT - International Workshop on Proof-Search in Axiomatic Theories and Type Theories - 2011*, Wroclaw, Pologne, 2011. Germain Faure, Stéphane Lengrand, Assia Mahboubi.

[AGST10] M. Armand, B. Grégoire, A. Spiwack, and L. Théry. Extending Coq with Imperative Features and Its Application to SAT Verification. In Kaufmann and Paulson [KP10], pages 83–98.

[ASM06] Fadi A. Aloul, Karem A. Sakallah, and Igor L. Markov. Efficient Symmetry Breaking for Boolean Satisfiability. *IEEE Trans. Computers*, 55(5):549–558, 2006.

[Bak91] Henry G. Baker. Shallow binding makes functional arrays fast. *SIGPLAN Notices*, 26(8):145–147, 1991.

[Bar99] B. Barras. *Auto-validation d'un système de preuves avec familles inductives.* PhD thesis, Université Paris 7 – Denis Diderot, 1999.

[BB09] B. Brummayer and A. Biere. Fuzzing and Delta-Debugging SMT Solvers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, pages 1–5. ACM, 2009.

[BBP11] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. Extending Sledgehammer with SMT Solvers. In Bjørner and Sofronie-Stokkermans [BSS11], pages 116–130.

[BCC$^+$99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Masahiro Fujita, and Yunshan Zhu. Symbolic model checking using sat procedures instead of bdds. In *DAC*, pages 317–320, 1999.

[BCH12] Mathieu Boespflug, Quentin Carbonneaux, and Olivier Hermant. The $\lambda\Pi$-calculus modulo as a universal proof language. In *Proof Exchange for Theorem Proving—Second International Workshop, PxTP 2012*, pages 28–43, 2012.

[BCP11] Frédéric Besson, Pierre-Emmanuel Cornilleau, and David Pichardie. Modular SMT Proofs for Fast Reflexive Checking Inside Coq. In Jouannaud and Shao [JS11], pages 151–166.

[BDG11] Mathieu Boespflug, Maxime Dénès, and Benjamin Grégoire. Full Reduction at Full Throttle. In Jouannaud and Shao [JS11], pages 362–377.

[BdODF09] T. Bouton, D.C.B. de Oliveira, D. Déharbe, and P. Fontaine. veriT: An Open, Trustable and Efficient SMT-Solver. In R. A. Schmidt, editor, *CADE*, volume 5663 of *Lecture Notes in Computer Science*, pages 151–156. Springer, 2009.

[Ben06] Nick Benton. Machine Obstructed Proof. In *Workshop on Mechanizing Metatheory*, volume 89, 2006.

[Bes06] F. Besson. Fast Reflexive Arithmetic Tactics the Linear Case and Beyond. In Thorsten Altenkirch and Conor McBride, editors, *TYPES*, volume 4502 of *Lecture Notes in Computer Science*, pages 48–62. Springer, 2006.

[BFMP11] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd Your Herd of Provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011.

[BGG$^+$92] Richard J. Boulton, Andrew D. Gordon, Michael J. C. Gordon, John Harrison, John Herbert, and John Van Tassel. Experience with embedding hardware description languages in hol. In Victoria Stavridou, Thomas F. Melham, and Raymond T. Boute, editors, *TPCD*, volume A-10 of *IFIP Transactions*, pages 129–156. North-Holland, 1992.

[BJP10] Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. Parametricity and dependent types. In Paul Hudak and Stephanie Weirich, editors, *ICFP*, pages 345–356. ACM, 2010.

[BMZ05] Alfredo Braunstein, Marc Mézard, and Riccardo Zecchina. Survey propagation: An algorithm for satisfiability. *Random Struct. Algorithms*, 27(2):201–226, 2005.

[Bry86] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[BS91] U. Berger and H. Schwichtenberg. An Inverse of the Evaluation Functional for Typed lambda-calculus. In *LICS*, pages 203–211. IEEE Computer Society, 1991.

[BSS11] Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors. *Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings*, volume 6803 of *Lecture Notes in Computer Science*. Springer, 2011.

[BST10] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, volume 13, 2010.

[BT07] Clark Barrett and Cesare Tinelli. CVC3. In Damm and Hermanns [DH07], pages 298–302.

[BW10] S. Böhme and T. Weber. Fast LCF-Style Proof Reconstruction for Z3. In Kaufmann and Paulson [KP10], pages 179–194.

[CK05] Donald Chai and Andreas Kuehlmann. A fast pseudo-Boolean constraint solver. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 24(3):305–317, 2005.

[Coo71] Stephen A. Cook. The Complexity of Theorem-Proving Procedures. In Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman, editors, *STOC*, pages 151–158. ACM, 1971.

[Cot10] Scott Cotton. Two Techniques for Minimizing Resolution Proofs. In Ofer Strichman and Stefan Szeider, editors, *SAT*, volume 6175 of *Lecture Notes in Computer Science*, pages 306–312. Springer, 2010.

[CTVW04] E. Clarke, M. Talupur, H. Veith, and D. Wang. Sat based predicate abstraction for hardware verification. In *Theory and Applications of Satisfiability Testing*, pages 267–268. Springer, 2004.

[Dar09] Z. Dargaye. *Vérification formelle d'un compilateur pour langages fonctionnels.* PhD thesis, Université Paris 7 Diderot, July 2009.

[dB70] N. G. de Bruijn. The mathematical language AUTOMATH, its usage, and some of its extensions. In *Symposium on automatic demonstration*, pages 29–61. Springer, 1970.

[Dén] M. Dénès. Coq with imperative data structures. `https://github.com/maximedenes/native-coq`.

[Den00] E. Denney. A prototype proof translator from hol to coq. In M. Aagaard and J. Harrison, editors, *TPHOLs*, volume 1869 of *Lecture Notes in Computer Science*, pages 108–125. Springer, 2000.

[DFMS10] Ashish Darbari, Bernd Fischer, and João Marques-Silva. Industrial-Strength Certified SAT Solving through Verified SAT Proof Checking. In Ana Cavalcanti, David Déharbe, Marie-Claude Gaudel, and Jim Woodcock, editors, *ICTAC*, volume 6255 of *Lecture Notes in Computer Science*, pages 260–274. Springer, 2010.

[DH07] Werner Damm and Holger Hermanns, editors. *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*. Springer, 2007.

[Dia75] R. Diaconescu. Axiom of Choice and Complementation. In *Proceedings of the American Mathematical Society*, volume 51, pages 176–178. American Mathematical Society, 1975.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960.

[DT10] Nachum Dershowitz and Iddo Tzameret. Complexity of propositional proofs under a promise. *ACM Trans. Comput. Log.*, 11(3), 2010.

[FC06] J.-C. Filliâtre and S. Conchon. Type-safe modular hash-consing. In Andrew Kennedy and François Pottier, editors, *ML*, pages 12–19. ACM, 2006.

[FM07] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In Damm and Hermanns [DH07], pages 173–177.

[FMM$^+$06] P. Fontaine, J.-Y. Marion, S. Merz, L.P. Nieto, and A.F. Tiu. Expressiveness + Automation + Soundness: Towards Combining SMT Solvers and Interactive Proof Assistants. In H. Hermanns and J. Palsberg, editors, *TACAS*, volume 3920 of *Lecture Notes in Computer Science*, pages 167–181. Springer, 2006.

[FMP11] Pascal Fontaine, Stephan Merz, and Bruno Woltzenlogel Paleo. Compression of Propositional Resolution Proofs via Partial Regularization. In Bjørner and Sofronie-Stokkermans [BSS11], pages 237–251.

[FS06] U. Furbach and N. Shankar, editors. *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4130 of *Lecture Notes in Computer Science*. Springer, 2006.

[Gil12] G. Gilbert. Intégration de Z3 en Coq. Internship report, 2012.

[GLJ93] Andrew J. Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *FPCA*, pages 223–232, 1993.

[Glo09] Stéphane Glondu. Extraction certifiée dans coq-en-coq. In Alan Schmitt, editor, *JFLA*, volume 7.2 of *Studia Informatica Universalis*, pages 383–410, 2009.

[GM08] G. Gonthier and A. Mahboubi. A small scale reflection extension for the Coq system. *Rapport de recherche INRIA*, 2008.

[GMR$^+$07] Georges Gonthier, Assia Mahboubi, Laurence Rideau, Enrico Tassi, and Laurent Théry. A modular formalisation of finite group theory. In Schneider and Brandt [SB07], pages 86–101.

[Gon08] G. Gonthier. Formal Proof – The Four-Color Theorem. *Notices of the AMS*, 55(11):1382–1393, 2008.

[GOS$^+$12] Vijay Ganesh, Charles W. O'Donnell, Mate Soos, Srinivas Devadas, Martin C. Rinard, and Armando Solar-Lezama. Lynx: A Programmatic SAT Solver for the RNA-Folding Problem. In Alessandro Cimatti and Roberto Sebastiani, editors, *SAT*, volume 7317 of *Lecture Notes in Computer Science*, pages 143–156. Springer, 2012.

[Gré03] B. Grégoire. *Compilation des termes de preuves: un (nouveau) mariage entre Coq et Ocaml.* Thése de doctorat, spécialité informatique, Université Paris 7, École Polytechnique, France, December 2003.

[GW07] F. Garillot and B. Werner. Simple types in type theory: Deep and shallow encodings. In Schneider and Brandt [SB07], pages 368–382.

[Har06] J. Harrison. Towards self-verification of HOL Light. In Furbach and Shankar [FS06], pages 177–191.

[HHM$^+$10] Thomas C. Hales, John Harrison, Sean McLaughlin, Tobias Nipkow, Steven Obua, and Roland Zumkeller. A Revision of the Proof of the Kepler Conjecture. *Discrete & Computational Geometry*, 44(1):1–34, 2010.

[HT98] J. Harrison and L. Théry. A Sceptic's Approach to Combining HOL and Maple. *Journal of Automated Reasoning*, 21(3):279–294, 1998.

[Hur09] Joe Hurd. OpenTheory: Package management for higher order logic theories. In Gabriel Dos Reis and Laurent Théry, editors, *PLMMS '09: Proceedings of the ACM SIGSAM 2009 International Workshop on Programming Languages for Mechanized Mathematics Systems*, pages 31–37. ACM, August 2009.

[JS11] Jean-Pierre Jouannaud and Zhong Shao, editors. *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings*, volume 7086 of *Lecture Notes in Computer Science*. Springer, 2011.

[KL12] Chantal Keller and Marc Lasson. Parametricity in an Impredicative Sort. In Patrick Cégielski and Arnaud Durand, editors, *CSL*, volume 16 of *LIPIcs*, pages 381–395. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.

[KMPS10] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Comfusy: A tool for complete functional synthesis. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 430–433. Springer, 2010.

[KP10] M. Kaufmann and L. 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*. Springer, 2010.

[KS08] D. Kroening and O. Strichman. *Decision Procedures: an Algorithmic Point of View*. Springer-Verlag New York Inc, http://www.decision-procedures.org, 2008.

[KW10] Chantal Keller and Benjamin Werner. Importing hol light into coq. In Kaufmann and Paulson [KP10], pages 307–322.

[LC09] Stéphane Lescuyer and Sylvain Conchon. Improving coq propositional reasoning using a lazy cnf conversion scheme. In Silvio Ghilardi and Roberto Sebastiani, editors, *FroCos*, volume 5749 of *Lecture Notes in Computer Science*, pages 287–303. Springer, 2009.

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.

[Let04] P. Letouzey. *Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq*. PhD thesis, Université Paris-Sud, Juillet 2004.

[LS88] J. Lambek and P.J. Scott. *Introduction to higher order categorical logic*, volume 7. Cambridge Univ Pr, 1988.

[MBG06] S. McLaughlin, C. Barrett, and Y. Ge. Cooperating Theorem Provers: A Case Study Combining HOL-Light and CVC Lite. *Electr. Notes Theor. Comput. Sci.*, 144(2):43–51, 2006.

[Mil72] R. Milner. Logic for Computable Functions: Description of a Machine Implementation. Technical report, Stanford University, 1972.

[Mil11] Dale Miller. A Proposal for Broad Spectrum Proof Certificates. In Jouannaud and Shao [JS11], pages 54–69.

[MT12] Sebastian Müller and Iddo Tzameret. Short Propositional Refutations for Dense Random 3CNF Formulas. In *LICS*, pages 501–510. IEEE, 2012.

[NO79] Greg Nelson and Derek C. Oppen. Simplification by Cooperating Decision Procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.

[NOT06] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(). *J. ACM*, 53(6):937–977, 2006.

[OS06] S. Obua and S. Skalberg. Importing HOL into Isabelle/HOL. In Furbach and Shankar [FS06], pages 298–302.

[Rey83] John C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.

[RGPS08] Nathan Robinson, Charles Gretton, Duc Nghia Pham, and Abdul Sattar. A compact and efficient sat encoding for planning. In Jussi Rintanen, Bernhard Nebel, J. Christopher Beck, and Eric A. Hansen, editors, *ICAPS*, pages 296–303. AAAI, 2008.

[RKJ08] P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In R. Gupta and S. P. Amarasinghe, editors, *PLDI*, pages 159–169. ACM, 2008.

[Rob65] John Alan Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM*, 12(1):23–41, 1965.

[SB07] K. Schneider and J. Brandt, editors. *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings*, volume 4732 of *Lecture Notes in Computer Science*. Springer, 2007.

[Spi06] A. Spiwack. Ajouter des entiers machine à Coq. Master's thesis, Ens Cachan, sep 2006.

[Tea11] The Coq Development Team. The Coq proof assistant: reference manual. *Rapport technique - INRIA*, 2011.

[Tse70] G. Tseitin. On the Complexity of Proofs in Propositional Logics. In *Seminars in Mathematics*, volume 8, pages 466–483, 1970.

[Wad89] Philip Wadler. Theorems for free! In *FPCA*, pages 347–359, 1989.

[Web05] T. Weber. A SAT-based Sudoku solver. In *LPAR*, pages 11–15, 2005.

[Web08] T. Weber. *SAT-based Finite Model Generation for Higher-Order Logic*. PhD thesis, Institut für Informatik, Technische Universität München, Germany, April 2008.

[Wie07] F. Wiedijk. Encoding the hol light logic in coq. *Unpublished notes*, 2007.