A Matter of Trust: Skeptical Communication Between Coq and External Provers (Detailed Description)

Chantal Keller

Abstract

Most theorem provers are either hard to trust because they are large programs or hard to use due to a lack of automation. They are thus used by two different communities depending on the property one needs most: the ability to prove quickly a large amount of theorems or a high level of safety. My thesis studies a communication between these different kinds of tools, by asking for proof witnesses in addition to yes/no answers from automated provers, and checking them in safe provers. It offers both a way to safely combine proofs coming from all these provers in order to benefit from the advantages of all of them and guaranties to automated theorem provers.

Keywords: automated deduction; decision procedures; higher-order logic; type theory.

1 Introduction

1.1 Background

Theorem proving is torn between three *a priori* incompatible but desirable features: expressivity, safety, and automation. In a schematic way, at the three vertices of this triangle, we could respectively find:

- interactive theorem provers based on Type Theory: they are very expressive, rely on a rather small kernel but which requires a deep knowledge to get convinced of its soundness, with full automation for proof checking but almost none for proof writing;
- HOL-like interactive theorem provers: they rely on a very small kernel, with a good expressivity and some automated proof search; and
- automatic theorem provers, such as satisfiability provers (SAT for the propositional part or SMT modulo theories) or first-order provers: proof search is fully automatized, but at the cost of a very large untrustable code and of lots of encodings on the front end due to their lack of expressivity, encodings that may also obfuscate the properties we are proving and thus be hard to trust.

My thesis presents a step towards reducing this gap by studying, in theory and in practice, a communication between these three different kinds of tools that are all good at one of these criteria. The soundness of this communication is established *skeptically* [HT98]: we ask for the tool to justify *a posteriori* all its actions and formally check the justifications. This requires to check back any achievement of the external tool, but it is robust to changes in the tool: as long as justifications remain the same, the skeptical checker is still relevant.

1.2 General idea

My approach consists in benefiting from the expressivity and rather high level of safety of the Coq interactive theorem prover[1], based on the Calculus of Inductive Constructions, to

[1]Coq received the ACM SIGPLAN Programming Languages Software Award in 2013.

check and combine theorems coming from Coq and external provers based on Higher-Order and First-Order logics into this system.

It relies on a common language composed of:

- a deep embedding in the Calculus of Inductive Constructions of the *formulas* of the Higher-Order or First-Order logic, which are carefully translated into shallow terms; and
- a certificate format for the *proofs* of these formulas, in the shape of a list of commands that can be executed in a certified way in order to formally establish the validity of theorems.

It exploits two features of the Calculus of Inductive Constructions:

- its expressivity, that allows to embed the terms of Higher-Order logic either deeply through an inductive data-type or shallowly as a subset of Coq's terms; and
- its computational power, both to define the translation between the two levels of embeddings and to check certificates via *computational reflection*.

Intuitively, computational reflection relies on the inherent conversion of the Calculus of Inductive Constructions to replace potentially large proofs with computations. Section 3.2 explains it in more details in the case of a communication with external provers.

This study is put into practice for 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. For both of them, embeddings and translations from one level to another has been defined, as well as certificates formats and efficient certified checkers. Preprocessors ensure the modularity with respect to concrete proof witnesses. It ends 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.

1.3 Outline

The remaining of this description is organized as follows. I will first describe a very general framework to make Coq communicate with external provers, with two ingredients: an encoding of the language of the external provers into Coq (Section 2), and a certified checker with respect to this encoding that is at the heart of proving safety, importing theorems and building new automatic tactics (Section 3). I will then apply this general framework to two different kinds of provers. SMTCoq (Section 4) efficiently checks proofs coming from SAT and SMT solvers in order to improve their safety and Coq's automation. HOLLIGHTCOQ (Section 5) imports HOL Light theorems into Coq in order to allow formal developments that use the advantages of both provers. I will finally conclude in Section 6.

2 A common language between two provers

In order to check theorems from an external prover into Coq, this latter needs to understand the statements of these theorems. In this section, I present in a very general setting my approach to the representation of statements coming from external provers into Coq, which enjoys the property of keeping them intelligible and thus usable in combination with user defined Coq theorems.

Note that this section presents a very generic way to embed some subset of terms into Coq independently from any inference system or proof witnesses format.

2.1 Deep and shallow embeddings

To represent a logical framework A inside Coq, there are mainly two possible ways:

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

The relevant differences between the two options are summed up in Table 1. In order to combine in the end developments partially made in Coq and in A, the theorem statements must be naturally stated in Coq, in the shallow embedding. However, to efficiently check them using the certificate, in particular if we want to use computational reflection, we need to reason on the structure of terms, in the deep embedding. This deep embedding thus forms a *API between the two provers*.

	Deep	Shallow

Repr.	`Inductive typeD := ...`
	`Inductive termD := ...`
		`Definition typeS := Type`

Final object	`Inductive derivD := ...` A Coq proof that the theorem is provable in A	Coq proof term The theorem is proved in Coq in the fragment corresponding to A
Benefits	Access the structure of terms - induction - computational reflection	Concrete Coq terms
My usage	API to the external world	Coq theorems

Table 1: Deep and shallow embeddings

This observation is detailed in Figure 1. The theorem statements we want to obtain belong to the shallow representation of terms in Coq. However, to communicate with the external tool, we use a deep representation of the same terms both inside Coq for computational reflection and outside Coq for inputs and outputs. We thus need to be able to switch between the two representations: from deep to shallow, we define in Coq an *interpretation* function mapping the terms of the external prover into their Coq counterparts; and from shallow to deep, we *reify* terms directly at the Ocaml level. Reification can indeed be used as an oracle without compromising soundness, as explained in Section 2.3. Notice that I do not have a deep embedding of the inference rules: instead, I will prove correctness up to the interpretation function, as explained in Section 3.1.

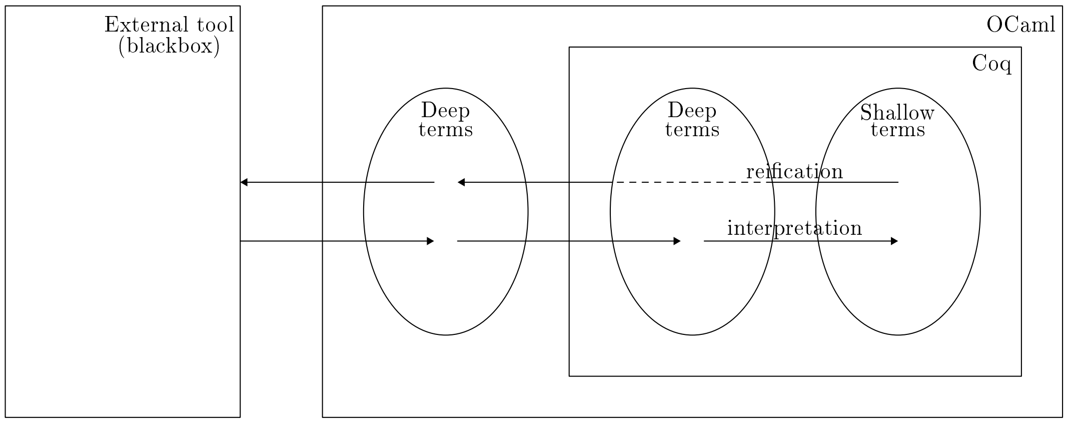


Figure 1: Interaction with external provers

2.2 From deep to shallow: interpretation

The interpretation function is a Coq program compiling the terms of the source language into their Coq counterparts. The difficulty is to type this program: deep terms have different types, thus their interpretations also have different Coq types. To handle this, I extended to the particular sets of terms used by the external provers I consider an idea originally given by Garillot and Werner as the compilation part of a Normalization by Evaluation function [GW07]. For a matter of clarity, I am going to explain this idea when the source language is the simply-typed λ-calculus.

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

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

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:

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

.

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:

$$\begin{array}{rcl} \mathrm{infer}(x^A) & = & A \\ \mathrm{infer}(\lambda x^A.u) & = & A \hookrightarrow \mathrm{infer}(u) \\ \mathrm{infer}(u\ v) & = & \left\{ \begin{array}{l} B \text{ if } \mathrm{infer}(u) = A \hookrightarrow B \text{ and } \mathrm{infer}(v) = A \\ \text{fails otherwise} \end{array} \right. \end{array} \tag{1}$$

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

which highlight the correspondence between the two.

2.3 From shallow to deep: reification

The role of the reification function can now be expressed in terms of the interpretation function. Its objective is to compute, from a Coq term s, a deep term d and an environment $\mathcal{I}$ such that

$$|d|_{\mathcal{I}} \text{ is convertible to } s$$

when they exist.

Stated as such, it becomes clearer that this function needs not be certified to keep soundness. Indeed, if it was to compute wrong d or $\mathcal{I}$, then $|d|_{\mathcal{I}}$ would not be convertible to s, but we would not prove something false. Nonetheless, we do not guarantee completeness, but this is already the case in a skeptical certification since we have no idea that the external prover will actually return a proof.

At the Ocaml level, it consists in syntactically inspecting Coq terms to reconstruct their structures when they belong to the set of terms we consider, which presents no theoretical difficulty.

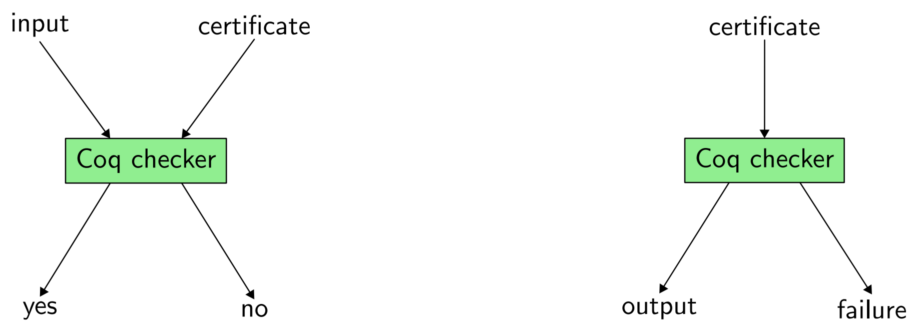
3 Interaction with external provers

Now that we defined an interface for terms of the external prover, the heart of the interaction consists in a *certified checker* for a *certificate format* given as a list of commands to execute in order to establish the validity of a given input. This checker can then be used independently to certify answers coming from the external solver as well as inside Coq to call external provers from Coq goals. We again present the main ideas in a very general setting.

3.1 A certified checker

Given a deeply embedded input formula ϕ and a certificate c supposed to establish the validity of ϕ, a checker is a Coq program that returns a Boolean, as illustrated by Figure 2(a). Its soundness is formulated with respect to the interpretation function, stating that whenever it returns "yes" on some formula ϕ, then the interpretation of ϕ is valid in any environment:

$$\forall \phi\ c, \text{checker}\ \phi\ c = \top \Rightarrow \forall\ \mathcal{I}, |\phi|_{\mathcal{I}}$$



(a) Standard certified checker

(b) Variant when the formula can be computed from the certificate

Figure 2: Two variants of certified checkers

A slight variant, presented in Figure 2(b), consists in computing the formula we want to prove from the certificate when possible. When the certificate is wrong, then the checker fails, which can be encoded in Coq by returning an `option` type. In this case, the soundness theorem, still stated with respect to the interpretation function, establishes that the checker only returns valid formulas when it does not fail:

$$\forall \phi\ c\ \mathcal{I}, |\text{checker}\ \phi\ c|_{\mathcal{I}}$$

Once again, we do not guarantee completeness, since certificates can be wrong: in this case, we just do not add anything to Coq.

3.2 Computational reflection

This is where *computational reflection* is put into practice. I am going to explain its operating principle for the first variant of the checker, but it works similarly for the second one. To establish some theorem t, it is sufficient to find by reification some ϕ and $\mathcal{I}$ such that $|\phi|_{\mathcal{I}}$ is convertible to t, and by calling an external prover some c such that:

$$\text{checker } \phi\ c \text{ reduces to } \top$$

A proof of t will thus simply be an application of the soundness theorem to ϕ, c, $\mathcal{I}$, and a proof that $\top = \top$, which is simply the reflexivity of $\top$.

The length of this proof is thus the length of the certificate, which is the smallest possible one in our setting. Moreover, it can be checked very efficiently if the reduction mechanism of the system is fast, which is the case for Coq with a reduction that uses an optimized call-by-value evaluation bytecode-based virtual machine [Gré03], and more recently a machine-based reduction available through the Ocaml's compiler [BDG11].

3.3 Three possible applications

Such a certified checker can be used for our three kinds of interactions.

First, the most straightforward application is to use it to check answers of untrusted provers, as illustrated by Figure 3(a). An external prover, applied to some problem, returns a proof witness. By parsing the problem and the proof witness, we can fed them to our certified checker, and expect a "yes" answer.

In this case, the trusting base consists of:

- the Coq's kernel;
- the input parser, to ensure that we indeed prove our problem: hence, it is important to make it simple so that we can be convinced that it has the shape of the identity; and
- the interpretation function: it should also be the canonical injection from deep terms to Coq terms, otherwise the soundness theorem does not have the intended meaning.

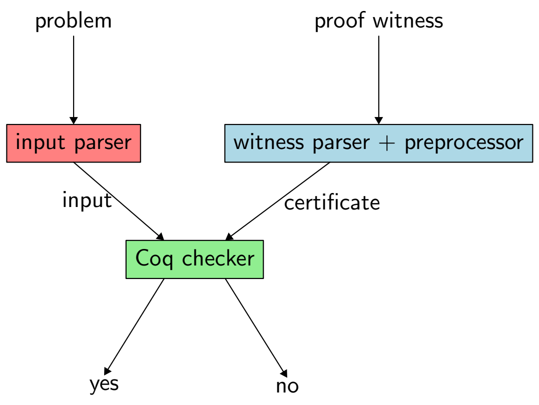
However, the proof witness parser does not need to be trusted: once again, if it is wrong, we just lose more completeness. It implies, as presented in the figure, that we can also preprocess the witnesses as much as we want. This is really interesting for *modularity*: it implies that we can plug *any* theorem prover, as long as we are able to preprocess its proofs witnesses into a common certificate format.

Finally, for this application, instead of running the checker inside Coq, we can *extract* it to run it in a more common programming language such as Ocaml.

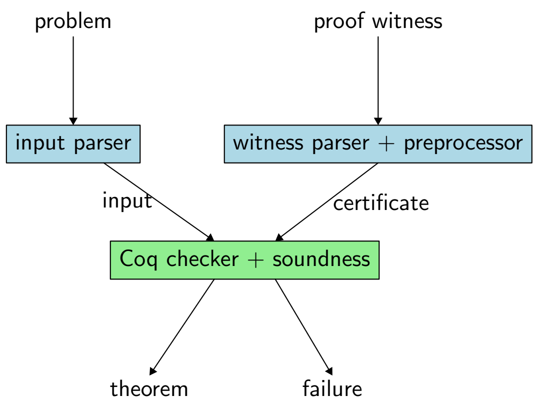
Second, another application of the checker we are interested in is to benefit in Coq from results that can be established more easily in external provers, because of their automation or their already existing libraries; it is illustrated by Figure 3(b). This time, instead of just returning a Boolean, it defines a new Coq constant which is the interpretation of the theorem validated by the soundness theorem on the proof witness, when this latter is correct.

In this case, the trusting base is only Coq's kernel: if something goes wrong during the process, then the importer is simply going to fail, and we will not add anything wrong to Coq. If the theorem is not the one the user intended, then he just cannot use it but it is still a valid theorem. Besides, the proof witness can be arbitrary preprocessed again, which is modular with respect to the external prover.

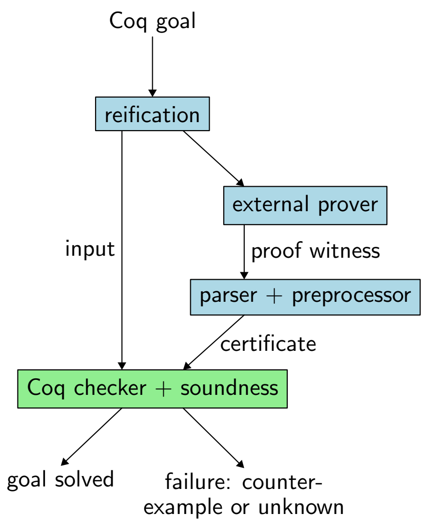
Finally, the last application of the checker is to provide automatic tactics by calling external automatic provers. This time, the problem does not come from outside but from a Coq goal through reification. This is a way to automatically prove Coq goals, but also to automatically find counter-examples if the external prover can return some, which can be really useful in a development. Here also, the trusting base is only Coq's kernel, and proof witnesses can be preprocessed.



(a) To check answers



(b) To import theorems



(c) Inside a tactic

Figure 3: Uses of a certified checker

I am now going to detail how I successfully applied this general setting to two different kinds of provers: SAT and SMT solvers, and interactive theorem provers based on Higher-Order Logic.

4 SMTCoq: cooperation with 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 above, 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]. In this section, I will explain how to apply our certification mechanism to these rising provers.

What is presented in this section has been implemented in a software called SMTCoq[2] which I still actively develop. It has been presented at two international conferences with peer-reviewed proceedings: the "First International Conference on Certified Programs and Proofs" (CPP'11) [AFG+11a] and the "International Workshop on Proof-Search in Axiomatic Theories and Type Theories" (PSATTT'11) [AFG+11b].

[2]SMTCoq is presented on this webpage: `http://cs.au.dk/~chkeller/Recherche/smtcoq.html`.

4.1 Certificates

SAT and SMT solvers are *satisfiability* solvers, which means that they check if a given formula is satisfiable or not, that is to say if there exists an assignment of the variables such that the interpretation of the formula is $\top$. To use these provers, we are mostly interested in unsatisfiability results, that are classically equivalent to provability.

In this case of unsatisfiability, SAT and SMT solvers can be easily instrumented to return *proofs in resolution and theory rules of the empty clause* [NOT06, BFT11]. I thus consider these as certificates.

I recall the basics: an *atom* a is an atomic formula in a combination of theories; a *literal* is an atom a or its negation $\neg a$; a *clause* is a disjunction of literals $l_1 \vee \cdots \vee l_n$; and a *formula in Conjunctive Normal Form* is a conjunction of clauses. The resolution rule, introduced by Robinson [Rob65], builds the clause $C \vee D$ from two clauses $a \vee C$ and $\neg a \vee D$, where no atom appears with one polarity in C and the other in D:

$$\frac{a \vee C \qquad \neg a \vee D}{C \vee D}$$

Certificates are trees in which nodes are either resolutions or *theory rules*, which are rules provable in one theory independently from the others and from clause reasoning. Examples of such rules are:

$$\frac{x_1 = y_1 \qquad \ldots \qquad x_n = y_n}{f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n)} \qquad\qquad \frac{}{\phi}$$

where ϕ is a tautology in linear integer arithmetic (also known as Presburger arithmetic). The first rule specifically states the congruence of equality with respect to function symbols, but we see in the second one that we can be really permissive as long as it does not mix theories.

4.2 An efficient and modular checker

This last point allows to build a very modular checker in terms of theories, as represented in Figure 4. A *main checker* dispatches the nodes of the certificate to the corresponding *small checkers*, which thus interact with each other only by means of the main checker. The correctness of the small checkers, stated in the same way as in Section 3.1, imply the correctness of the whole checker. It is thus really easy to add new small checkers: they just need to know about the deep embedding of the terms that concerns them, but not the other terms nor theories.

The efficiency of this checker relies on two ingredients:

- a careful choice in the deep embedding: terms are *maximally shared* to be compact as well as *stratified* to avoid some small checkers to be slowed down by too much information (for instance, the resolution checker does not need to look inside atoms); and
- the use of *efficient data-structures* to represent the intermediate objects appearing in the computation.

For small checkers, we either implemented dedicated ones, e.g. for congruence closure, or used existing decision procedures in Coq, e.g. Micromega [Bes06] for linear arithmetic.

4.3 Checking answers, importing theorems, and tactics

On top of this checker, we wrote preprocessors for the prof witnesses coming from the SAT solver ZChaff [FMM07] and the SMT solver veriT [BdODF09]. Following the mechanism presented in Section 3.3, we designed commands to check answers and import theorems as well as tactics to call external provers from Coq goals.

We evaluated these commands and tactics against:

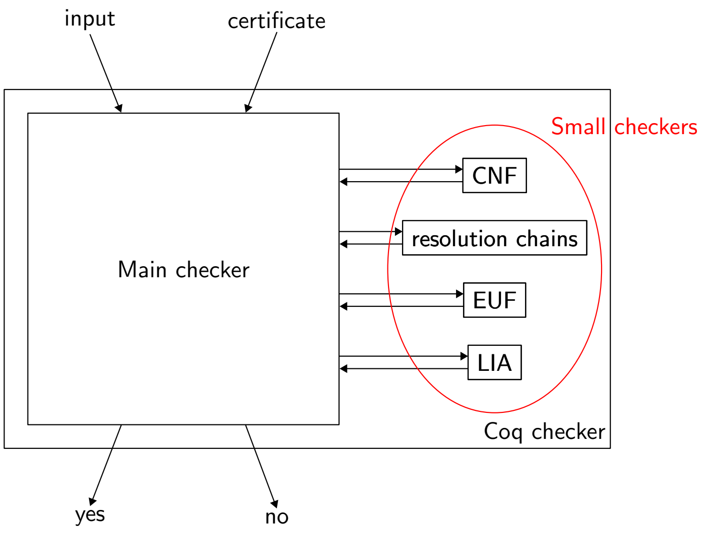


Figure 4: Architecture of the Coq checker

- proof reconstruction of SAT and SMT proof witnesses in the Isabelle/HOL interactive theorem prover [Web08, BW10]; and
- Ergo, a SMT solver written and certified in Coq [LC09].

The experiments show that SMTCoq is more efficient than these state-of-the-art techniques to certify SMT solvers. Qualitatively, SMTCoq is more expressive than Ergo, but less powerful than the Isabelle/HOL Sledgehammer tactic built among others on top of [Web08, BW10].

4.4 Future directions

I intend to spread SMTCoq. For this, I am improving it accordingly to users' demands. In particular, I am currently handling quantifiers: it will make the tactics really more usable if they can instantiate previously user-defined lemmas. To improve the tactics, I am also working on a certified encoding of Coq goals into the logic of SMT solvers. Finally, I work on encoding the reasoning of other kinds of automatic provers into my certificate format, in order to use SMTCoq as a back-end for a larger set of provers than SAT and SMT solvers [Kel13].

In the longer term, I would like to use this SAT power given to Coq to build new decision procedures based on bit blasting, a reduction of some kinds to problems to SAT.

5 HOLLIGHTCOQ: importing HOL Light into Coq

Cooperation between proof assistants is also 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[3] [HHM$^+$10], aiming at a formal proof of the Kepler Conjecture: some parts involve mathematical analysis, a domain widely explored by the HOL Light interactive theorem provers; others are proved by the check of certificates provided by oracles, which would be efficiently done by computational reflection in Coq. I will now explain how to apply my certification mechanism to allow formal developments in different proof assistants.

What is presented in this section has been implemented in a software called HOLLIGHTCOQ[4] which I still maintain. It has been presented at the international conference with

[3]The progress of this project is available at `http://code.google.com/p/flyspeck`.
[4]HOLLIGHTCOQ is presented on this webpage:

peer-reviewed proceedings “First International Conference on Interactive Theorem Proving” (ITP’11) [KW10].

5.1 Embeddings

Contrary to SMTCoq, the effort has been put on carefully defining embeddings that preserve the intelligibility of theorem statements, which is important if we want to develop formalization involving both HOL Light and Coq, rather than on efficiency, which is nonetheless still quite good since our generic approach ensures the use of computational reflection. This subsection describes HOL systems and is not specific to HOL Light.

Higher-Order Logic can be formulated in terms of a small bunch of rules over propositions stated as terms of the simply-typed λ-calculus with prenex polymorphism and user-defined term and type constants. Examples of these rules include:

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

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

I extended what I presented in Section 2.2 for the simply-typed λ-calculus to handle prenex polymorphism and user-defined term and type constants. The interesting point for intelligibility is the translation of these user-defined constants. Indeed, to interact with Coq theorems, we do not want to translate them into their HOL definitions, since most corresponding constants are not defined in the same way in Coq. My solution is to allow the user to translate them into any Coq term of the same type, as long as one can prove that the interpretation of the HOL definition of the constant is equivalent to its Coq mapping. It allows to finally obtain theorems as the user would have stated them directly in Coq.

5.2 Certificates

At the time when this work started, there was no standard for Higher-Order Logic certificates. I chose a format who is simply a skeleton of derivations in HOL: the nodes belong to an erased version of the small bunch of rules evoked above. On the same examples, this nodes become respectively $\textsc{refl}(t)$, $\textsc{assume}(p)$, $\textsc{beta}(\lambda x.t)$ and $\textsc{abs}(x,\mathrm{n})$ where n is inductively another node.

A simple checker for such certificates is rather straightforward. More interestingly, the soundness theorem associated to the checker establishes in Coq the correctness of this standard presentation of Higher-Order Logic, which is by itself an interesting formalization.

5.3 Importing theorems

In the case of HOL Light, we are mostly interested in the application of Figure 3(b) that imports theorems – there is no need to just check HOL Light proofs since HOL Light is already safer than Coq, and calling HOL Light from Coq goals would not give much automation.

On top of the checker, we use an implementation for HOL Light of our certificates called Proof recording [OS06]. Since HOL Light does not store proofs, it requires to first record them in order to be able to export them. I instrumented Proof recording to generate Coq files in my particular embedding.

With Proof recording, I was able to import in Coq HOL Light’s standard library as well as two non-trivial developments: a proof of consistency of HOL Light in HOL Light [Har06], and the library to reason about basic linear algebra. The time and memory consumption were

http://cs.au.dk/~chkeller/Recherche/hollightcoq.html. It is also distributed with HOL Light: http://www.cl.cam.ac.uk/~jrh13/hol-light.

too large for a everyday use, but not much a bottleneck in the objective of finally checking a development that involves both provers.

Qualitatively, the goal of obtaining understandable theorems is perfectly achieved. As an example, I can import HOL Light theorems on unary integers as Coq theorems on binary integers, and compose them with Coq theorems on binary integers from Coq's standard library. I thus obtain new properties that were not formalized before in any of the two provers.

5.4 Future directions

Now, a generic proof format for Higher-Order Logic called OpenTheory [Hur09] is being actively developed. Compared to Proof recording, it is very modular and thus likely to require less time and memory to be re-checked. I intend to switch to this format as well as to use more efficient data-structures, in order to make HOLLIGHTCOQ scale to larger developments, with the Flyspeck Project as a leading objective. Thanks to the modularity of my certifying framework, these switches do not require to change the embedding, which has already shown to achieve its goal.

6 Conclusion

In my thesis, I proposed an original framework to deal with the communication between provers in the broad sense with Coq. This framework relies on a careful choice of embeddings of the terms of the external prover into Coq. I described how this framework can be used to benefit from safety, automation and developments of these external provers.

I put it into practice for two different applications: SMTCoq, designed for efficiency and automation, and HOLLIGHTCOQ, designed for intelligibility. Both of them are modular and have shown to scale to non trivial certifications. They open new perspectives towards decision procedures and formal developments.

References

[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.

[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.

[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.

[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.

[BFT11] F. Besson, P. Fontaine, and L. Théry. A Flexible Proof Format for SMT: a Proposal. In *PxTP 2011: First International Workshop on Proof eXchange for*

Theorem Proving August 1, 2011 Affiliated with CADE 2011, 31 July-5 August 2011 Wrocław, Poland, pages 15–26, 2011.

[BW10] S. Böhme and T. Weber. Fast LCF-Style Proof Reconstruction for Z3. In Kaufmann and Paulson [KP10], pages 179–194.

[FMM07] Z. Fu, Y. Marhajan, and S. Malik. zChaff. *Research Web Page. Princeton University, USA,(March 2007)* `http://www.princeton.edu/~chaff/zchaff.html`, 2007.

[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.

[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 K. Schneider and J. Brandt, editors, *TPHOLs*, volume 4732 of *Lecture Notes in Computer Science*, pages 368–382. Springer, 2007.

[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.

[Kel13] Chantal Keller. Extended Resolution as Certificates for Propositional Logic. In Jasmin Christian Blanchette and Josef Urban, editors, *PxTP@CADE*, volume 14 of *EPiC Series*, pages 96–109. EasyChair, 2013.

[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.

[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.

[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.

[Rob65] John Alan Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM*, 12(1):23–41, 1965.

[Web08] T. Weber. *SAT-based Finite Model Generation for Higher-Order Logic.* PhD thesis, Institut für Informatik, Technische Universität München, Germany, April 2008.