Zenon: an Extensible Automated Theorem Prover Producing Checkable Proofs

Richard Bonichon[1], David Delahaye[2], and Damien Doligez[3]

[1] LIP6/Paris 6, Paris, France,
Richard.Bonichon@lip6.fr
[2] CEDRIC/CNAM, Paris, France,
David.Delahaye@cnam.fr
[3] INRIA, Rocquencourt, France,
Damien.Doligez@inria.fr

Abstract. We present Zenon, an automated theorem prover for first order classical logic (with equality), based on the tableau method. Zenon is intended to be the dedicated prover of the Focal environment, an object-oriented algebraic specification and proof system, which is able to produce OCaml code for execution and Coq code for certification. Zenon can directly generate Coq proofs (proof scripts or proof terms), which can be reinserted in the Coq specifications produced by Focal. Zenon can also be extended, which makes specific (and possibly local) automation possible in Focal.

1 Introduction

Theorem proving is generally separated into two distinct domains: automated theorem proving and interactive theorem proving. Even if these two domains are obviously connected, it seems that in practice, they have little interaction. Actually, the motivations are quite different: automated theorem proving focuses on heuristic concerns (complexity, efficiency, ...) to solve well-identified problems, whereas interactive theorem proving is more concerned with providing means (essentially tools) to achieve proofs of theorems. As a consequence, in automated theorem proving, it is quite difficult to produce formal proofs and in general, the corresponding tools only generate proof traces, which can be seen as abstractions of formal proofs and cannot be directly translated into machine checkable proofs. In this way, we can understand how complicated it is to integrate automated theorem proving features into interactive theorem provers, which tend to suffer from a certain lack of automation. Over the past ten years, some experiments have aimed to make these two kinds of theorem proving activities interact, such as between Gandalf and HOL by J. Hurd [5], between Otter and ACL2 by W. McCune and O. Shumsky [8], between Bliksem and Coq by M. Bezem, D. Hendriks and H. de Nivelle [2], or more recently between E, SPASS, Vampire and Isabelle by L. C. Paulson and K. W. Susanto [9]. However, these examples of integration are not fully satisfactory, since the design of the corresponding automated theorem provers is clearly separated from the automation that could be required by the respective interactive theorem provers. In particular, it is impossible to extend the automated theorem prover to manage a very specific and local need for automation.

In this paper, we present Zenon, an automated theorem prover for first order classical logic (with equality), based on the tableau method. Zenon is not supposed to be only another general-purpose automated theorem prover, but is designed to be the reasoning support mechanism of the Focal [15] environment, initially conceived by T. Hardin and R. Rioboo. Focal is a language in which it is possible to build applications step by step, going from abstract specifications to concrete implementations. These different structures are combined using inheritance and parameterization, inspired by object-oriented programming; in addition, each of these structures is equipped with a carrier set, providing a typical algebraic specification flavor. Moreover, in this language, there is a clean separation between the activities of programming and proving. In particular, the compiler is able to produce OCaml [14] code for execution and Coq [13] code for certification. In this compilation scheme, Zenon is involved in the certification part, between the specification level and the generated Coq implementation. Zenon is intended to be the prover of Focal, whereas Coq is used only as a proof checker to ensure the soundness of the final output.

Beyond the automation itself, Zenon brings an effective help to the design of Focal. First, Zenon uses the tableau method. Even though, these days, this method is generally considered as not very efficient (compared to resolution, for example), it has the advantage of being very appropriate for building formal proofs. In this way, Zenon has a low-level format of proofs, which is very close to a sequent calculus. From this low-level format, Zenon can directly produce proofs for Coq (it could be easily done for other proof assistants). This feature can be seen as a guarantee of soundness for the implementation of Zenon, but it is also essential to Focal, where the Coq proofs produced by Zenon are reinserted in the Coq specifications generated by the Focal compiler and fully verified by Coq. In addition, Zenon is also able to produce proof terms for Coq (using its Curry-Howard isomorphism capability), so that Zenon verifies the De Bruijn criterion [1], i.e. it generates proof terms that can be checked independently by a relatively small and easily hand checked algorithm. This means that it is possible to verify Zenon's proofs without Coq, using another tool that would implement *only* the type-checking of Coq. Second, Zenon can be easily extended and this is directly related to the use of the tableau method, which is also very appropriate to handle additional rules. Thanks to this feature, it is possible to manage specific (and possibly local) needs of automation in Focal, such as arithmetic, induction, etc.

The paper is organized as follows: in Section 2, we give the rules of the search method used by Zenon, as well as the format of the generated proofs (in this part, we also point out some specific implementation techniques, such as the use of non-destructive rules and pruning, the management of lemmas or the extension mechanism); in Section 3, we describe the intermediate proof format produced by translating from the proof search rules; in Section 4, we give the translation from this intermediate format to Coq proofs; in Section 5, we provide some examples of use, coming from the TPTP library but also from Focal applications.

2 MLproof

The MLproof inference rules (Figures 1 and 2) are used by Zenon to search for a proof. These rules are applied with the normal tableau method: starting from the negation of the goal, apply the rules in top-down fashion to build a tree. When all branches are *closed* (i.e. end with an application of a closure rule), the tree is closed. The closed tree is a proof of the goal.

Note that this algorithm is applied in strict depth-first order: we close the current branch before starting work on another branch. Moreover, we work in a non-destructive way: working on one branch will never change the formulas present in any other branch.

We divide these rules into five distinct classes to be used for a more efficient proof search. This extends the usual sets of rules dealing with $\alpha, \beta, \delta, \gamma$-formulas and closure ($\odot$) with the specific rules of Zenon. We list below the five sets of rules and their elements:

α	$\alpha_{\neg\vee}, \alpha_{\wedge}, \alpha_{\neg\Rightarrow}, \alpha_{\neg\neg}, \neg_{\text{refl}}$ unfolding rules
β	$\beta_{\vee}, \beta_{\neg\wedge}, \beta_{\Rightarrow}, \beta_{\Leftrightarrow}, \beta_{\neg\Leftrightarrow}, \neq_{\text{func}}$ trans, pred, sym, transsym, transeq, transeqsym
δ	$\delta_{\exists}, \delta_{\neg\forall}$
γ	$\gamma_{\forall}, \gamma_{\neg\exists}, \gamma_{\forall\text{inst}}, \gamma_{\neg\exists\text{inst}}, \gamma_{\forall\text{un}}, \gamma_{\neg\exists\text{un}}$
$\odot$	$\odot_{\top}, \odot_{\bot}, \odot, \odot_{r}, \odot_{s}$

As hinted by the use of the ϵ symbol in the rules, the δ rules are handled with Hilbert's operator [7] rather than using skolemization.

The following subsections describe specific features of our theorem prover, starting with how metavariables are used in a non-destructive setting.

2.1 Handling of metavariables

What we call here metavariables are often named *free variables* in tableau-related literature. They are not used as variables in Zenon as they are never substituted.

Instead of substitution, we use the following method: when we encounter a universal formula $\forall x\ P(x)$, we apply rule $\gamma_{\forall M}$, which introduces a new metavariable, linked to this universal formula. Then, when we have a potential contradiction such as $\neg R_r(t, X)$, we apply rule $\gamma_{\forall\text{inst}}$ (with the t given by the potential contradiction) *in the current branch* to our original universal formula. If this instantiation closes the subtree rooted at the $\gamma_{\forall\text{inst}}$ node, we know that pruning (see section 2.2) will remove the nodes between the two γ nodes, hence removing the need for substitution of the metavariable.

Closure and cut rules

$$\frac{\bot}{\odot}\odot_\bot \qquad \frac{\neg\top}{\odot}\odot_{\neg\top} \qquad \frac{}{P \mid \neg P}\text{cut}$$

$$\frac{\neg R_r(t,t)}{\odot}\odot_r \qquad \frac{P \qquad \neg P}{\odot}\odot \qquad \frac{R_s(a,b) \qquad \neg R_s(b,a)}{\odot}\odot_s$$

Analytic rules

$$\frac{\neg\neg P}{P}\alpha_{\neg\neg} \qquad \frac{P \Leftrightarrow Q}{\neg P, \neg Q \mid P, Q}\beta_\Leftrightarrow \qquad \frac{\neg(P \Leftrightarrow Q)}{\neg P, Q \mid P, \neg Q}\beta_{\neg\Leftrightarrow}$$

$$\frac{P \land Q}{P, Q}\alpha_\land \qquad \frac{\neg(P \lor Q)}{\neg P, \neg Q}\alpha_{\neg\lor} \qquad \frac{\neg(P \Rightarrow Q)}{P, \neg Q}\beta_{\neg\Rightarrow}$$

$$\frac{P \lor Q}{P \mid Q}\beta_\lor \qquad \frac{\neg(P \land Q)}{\neg P \mid \neg Q}\beta_{\neg\land} \qquad \frac{P \Rightarrow Q}{\neg P \mid Q}\beta_\Rightarrow$$

$$\frac{\exists x\ P(x)}{P(\epsilon(x).P(x))}\delta_\exists \qquad \frac{\neg\forall x\ P(x)}{\neg P(\epsilon(x).P(x))}\delta_{\neg\forall}$$

γ-rules

$$\frac{\forall x\ P(x)}{P(X)}\gamma_{\forall M} \qquad \frac{\neg\exists x\ P(x)}{\neg P(X)}\gamma_{\neg\exists M} \qquad \frac{\forall x\ P(x)}{\forall x_1 ... x_n\ P(s(x_1, ..., x_n))}\gamma_{\forall \text{un}}$$

$$\frac{\forall x\ P(x)}{P(t)}\gamma_{\forall \text{inst}} \qquad \frac{\neg\exists x\ P(x)}{\neg P(t)}\gamma_{\neg\exists \text{inst}} \qquad \frac{\neg\exists x\ P(x)}{\neg\exists x_1 ... x_n\ P(s(x_1, ..., x_n))}\gamma_{\neg\exists \text{un}}$$

Relational rules

$$\frac{P(t_1, ..., t_n) \qquad \neg P(s_1, .., s_n)}{t_1 \neq s_1 \mid ... \mid t_n \neq s_n}\text{pred} \qquad \frac{f(t_1, ..., t_n) \neq f(s_1, ..., s_n)}{t_1 \neq s_1 \mid ... \mid t_n \neq s_n}\text{fun}$$

$$\frac{R_s(s,t) \qquad \neg R_s(u,v)}{t \neq u \mid s \neq v}\text{sym} \qquad \frac{\neg R_r(s,t)}{s \neq t}\neg_{\text{refl}}$$

$$\frac{R_t(s,t) \qquad \neg R_t(u,v)}{u \neq s, \neg R_t(u,s) \mid t \neq v, \neg R_t(t,v)}\text{trans}$$

$$\frac{R_{ts}(s,t) \qquad \neg R_{ts}(u,v)}{v \neq s, \neg R_{ts}(v,s) \mid t \neq u, \neg R_{ts}(t,u)}\text{transsym}$$

$$\frac{s = t \qquad R_t(u,v)}{u \neq s, \neg R_t(u,s) \mid \neg R_t(u,s), \neg R_t(t,v) \mid t \neq v, \neg R_t(t,v)}\text{transeq}$$

$$\frac{s = t \qquad R_{ts}(u,v)}{v \neq s, \neg R_{ts}(v,s) \mid \neg R_{ts}(v,s), \neg R_{ts}(t,u) \mid t \neq u, \neg R_{ts}(t,u)}\text{transeqsym}$$

where R_r, R_s, R_t, and R_{ts} are respectively reflexive, symmetric, transitive, and transitive-symmetric relations.

Fig. 1. MLproof rules (part 1)

Unfolding rules: if $P(x) \hat{=} \mathrm{Def}(x)$ and $f(x) \hat{=} \mathrm{def}(x)$ then

$$\frac{P(x)}{\mathrm{Def}(x)}\ \text{p-unfold} \qquad \frac{\neg P(x)}{\neg \mathrm{Def}(x)}\ \text{p-unfold}_{\neg}$$

$$\frac{f(x) = t}{\mathrm{def}(x) = t}\ \text{f-unfold}_{l=} \qquad \frac{t = f(x)}{t = \mathrm{def}(x)}\ \text{f-unfold}_{r=}$$

$$\frac{f(x) \neq t}{\mathrm{def}(x) \neq t}\ \text{f-unfold}_{l} \qquad \frac{t \neq f(x)}{t \neq \mathrm{def}(x)}\ \text{f-unfold}_{r}$$

Extension rule

$$\frac{C_1, ..., C_p}{H_{11}, ..., H_{1m} \mid ... \mid H_{n1}, ..., H_{nq}}\ \begin{array}{l}\text{ext}(\mathsf{name},\mathsf{args}, \\ [C_i],[H_{1j}, ..., H_{nk}])\end{array}$$

where name is the name of a predefined lemma s.t.
$C_1 \wedge ... \wedge C_p \Rightarrow \bigvee_j (\bigwedge_i H_i j)$

Fig. 2. MLproof rules (part 2)

If the instantiation does not close the subtree, the formulas containing the metavariable are still available in the current branch to trigger other potential contradictions, hence we get as many instantiations as needed from a single application of the $\gamma_{\forall M}$ rule. This means that we do not need to use iterative deepening to ensure completeness.

Let us consider the following example:

$$\dfrac{\forall x, P(x) \vee Q(x) \quad \neg P(a) \quad \neg Q(a)}{\dfrac{P(X) \vee Q(X)}{\dfrac{\dfrac{P(X)}{\dfrac{P(a) \vee Q(a)}{\dfrac{P(a)}{\odot}\odot \quad \dfrac{Q(a)}{\odot}\odot}\beta_\vee}\gamma_{\forall\mathrm{inst}} \quad Q(X)}{}\beta_\vee}}\gamma_{\forall M}$$

In this case, the rule $\gamma_{\forall\mathrm{inst}}$ is triggered by the match between $\neg P(a)$ and $P(X)$, which tells us to instantiate $\forall x, P(x) \vee Q(x)$ with the value a. This tree is not a complete proof because it has an open branch (under $Q(X)$). As we will see in Section 2.2, this open branch does not need to be explored because we can remove it (along with some nodes) to yield a closed proof tree of the original formulas.

2.2 Minimizing the tree size

For efficient proof search, a prover must minimize the size of the search tree. This is done in two ways. The first is by *choosing the order in which the rules are* *applied*: non-branching rules are tried first. It induces the following $\prec$ ordering on the application of the rules $\odot \prec \alpha \prec \delta \prec \beta \prec \gamma$, stating thereby that any applicable $\odot$ rule has priority over any of the other possible rules.

The second is by *pruning*. When a branching node N has a closed subtree as one of its branches B, we can examine this closed subtree to determine which formulas are useful. If the formula introduced by N in B is not in the set of useful formulas, we can remove N and graft the subtree in its place because the subtree is a valid refutation of B without N.

The notion of *useful formula* is defined as follows: a formula is useful in a subtree if it is one of the formulas appearing in the hypotheses (the upper side) of a rule application in that subtree.

Consider the example of section 2.1. There is a subtree rooted at the $\forall$inst node. This subtree does not *use* the formula $P(X)$ that appears just above it, because the premise of the $\forall$inst rule is the formula $\forall x, P(x) \vee Q(x)$ at the root of the proof tree, and none of the other subtree nodes uses $P(X)$. Because of this, we can remove the $\beta_\vee$ node above the subtree, and graft the subtree in its place. We can proceed in the same fashion to remove the $\gamma_{\forall M}$ node, and we get the following tree:

$$\cfrac{\forall x, P(x) \vee Q(x) \quad \neg P(a) \quad \neg Q(a)}{\cfrac{P(a) \vee Q(a)}{\cfrac{P(a)}{\odot}\odot \quad \cfrac{Q(a)}{\odot}\odot}\beta_\vee}\gamma_{\forall\text{inst}}$$

This time, the proof tree is closed and the proof search is over. The importance of this pruning is that we have completely avoided doing the proof search below the $Q(X)$ branch by carefully examining the result of the proof search in the $P(X)$ branch, thereby reducing the branching factor of the search tree. In the process, we have reduced the size of the resulting proof as compared to the proof search tree.

2.3 Extensions

Zenon offers the ability to extend its core of deductive rules to match certain specific requirements. For instance, the extension named `Coqbool` is regularly used in the setting of Focal, where a function $P(x, y)$ returning a boolean result is encapsulated into a `Is_true(P(x,y))` predicate as it is translated into the corresponding Coq file. In the case where P is transitive (for example), this prevents Zenon from using its specific inference rules, thereby reducing the efficiency of the proof search. Our solution is to transform all occurrences of `Is_true(P(x,y))` into a corresponding `Is_true__P(x,y)` predicate which will let Zenon make use of its transitivity property.

Concretely, extensions are arbitrary OCaml files that implement new inference rules; they are loaded through command-line options when Zenon is started, along with Coq files containing the lemmas used to translate the inference rules introduced by the extension.

2.4 Subsumption

Whenever the current branch contains a superset of the formulas used in an already-closed subtree, we can graft this subtree at the current node because it is a valid closure of the current branch. The implementation maintains a data structure with all the subtrees closed so far (indexed by their used formulas) and queries this data each time a formula is added to the current branch.

We can illustrate subsumption with the following example:

$$\dfrac{B \lor C \quad B \Rightarrow D \quad C \Rightarrow D \quad D \Rightarrow E \quad \neg E}{\dfrac{B}{\dfrac{\neg B}{\odot}\,\odot \quad \dfrac{D*}{\dfrac{\neg D}{\odot}\,\odot \quad \dfrac{E}{\odot}\,\odot}\,\beta_{\Rightarrow}}\,\beta_{\Rightarrow} \qquad \dfrac{C}{\dfrac{\neg C}{\odot}\,\odot \quad \mathrm{D}}\,\beta_{\Rightarrow}}\,\beta_{\lor}$$

Consider the $D*$ subtree in the left half of the tree and the open branch under D. The formulas used by the $D*$ subtree are D, $D \Rightarrow E$, and $\neg E$. The same formulas are already available in the open branch, thus we do not need to search for a proof: we can simply reuse the $D*$ subtree. In fact, the implementation does not copy the subtree, but uses sharing (hence turning the proof tree into a dag). Such shared subtrees appear as lemmas in the Coq proof output.

3 LLproof

LLproof is the low-level language of proofs produced by Zenon, which makes the generation of machine checkable proofs possible (see Section 4 for an example in the framework of Coq). Once a proof has been found with the MLproof rules, it is translated to this sequent-like language. We will sketch a proof of soundness and completeness of MLproof proofs w.r.t. LLproof proofs.

LLproof rules (Figures 3 and 4) indeed describe a one-sided sequent calculus with explicit contractions in every inference rule, which roughly resembles an upside-down non-destructive tableau method. This sequent calculus is extended to handle unfolding, lemmas and the extension mechanism of Zenon.

Translating mid-level to low-level proofs gives us a direct proof of soundness for MLproof w.r.t. LLproof. There is a one-to-one correspondence between parts of the two calculi, most notably those which do not introduce quantifiers in MLproof (quantifier-free fragment, axioms).

We can now proceed to prove the following proposition.

Theorem 1 (Soundness and completeness of MLproof w.r.t. LLproof).

1. *Every formula provable in* LLproof *has a proof in* MLproof.
2. *Every formula provable in* MLproof *has a proof in* LLproof.

Proof. Proof of (1) is immediate as every rule of LLproof has a direct equivalent in MLproof, except the lemma rule, but we can only apply the lemma rule when

Closure and quantifier-free rules

$$\frac{}{\bot \vdash \bot}\ \bot \qquad \frac{}{\neg\top \vdash \bot}\ \neg\top \qquad \frac{}{\Gamma, P, \neg P \vdash \bot}\ \text{ax}$$

$$\frac{}{t \neq t \vdash \bot}\ \neq \qquad \frac{\Gamma, P, \neg\neg P \vdash \bot}{\Gamma, P \vdash \bot}\ \neg\neg \qquad \frac{\Gamma, P \vdash \bot \qquad \Gamma, \neg P \vdash \bot}{\Gamma \vdash \bot}\ \text{cut}$$

$$\frac{\Gamma, P \wedge Q, P, Q \vdash \bot}{\Gamma, P \wedge Q \vdash \bot}\ \wedge \qquad \frac{\Gamma, P \vee Q, P \vdash \bot \qquad \Gamma, P \vee Q, Q \vdash \bot}{\Gamma, P \vee Q \vdash \bot}\ \vee$$

$$\frac{\Gamma, P, \neg Q, \neg(P \Rightarrow Q) \vdash \bot}{\Gamma, \neg(P \Rightarrow Q) \vdash \bot}\ \neg\Rightarrow \qquad \frac{\Gamma, \neg P, P \Rightarrow Q \vdash \bot \qquad \Gamma, Q, P \Rightarrow Q \vdash \bot}{\Gamma, P \Rightarrow Q \vdash \bot}\ \Rightarrow$$

$$\frac{\Gamma, \neg P, \neg Q, \neg(P \vee Q) \vdash \bot}{\Gamma, \neg(P \vee Q) \vdash \bot}\ \neg\vee \qquad \frac{\Gamma, \neg P, \neg(P \wedge Q) \vdash \bot \quad \Gamma, \neg Q, \neg(P \wedge Q) \vdash \bot}{\Gamma, \neg(P \wedge Q) \vdash \bot}\ \neg\wedge$$

$$\frac{\Gamma, P \Leftrightarrow Q, \neg P, \neg Q \vdash \bot \qquad \Gamma, P \Leftrightarrow Q, P, Q \vdash \bot}{\Gamma, P \Leftrightarrow Q \vdash \bot}\ \Leftrightarrow$$

$$\frac{\Gamma, \neg P, Q, \neg(P \Leftrightarrow Q) \vdash \bot \qquad \Gamma, P, \neg Q, \neg(P \Leftrightarrow Q) \vdash \bot}{\Gamma, \neg(P \Leftrightarrow Q) \vdash \bot}\ \neg\Leftrightarrow$$

Fig. 3. LLproof rules (part 1)

we have a proof of the lemma's statement, which we can handle in MLproof by grafting a copy of the lemma's proof in the place of the lemma rule.

The proof of (2) is not so immediate as we have to transform some MLproof rules which are the combination of two or more lower-level rules. It proceeds by induction on the size of the MLproof proofs; the details of the proof are not given here.

4 Producing Coq proofs

As we said in the introduction, Zenon is able to produce Coq [13] proofs, and this automatic generation is carried out from the LLproof format described in Section 3. From a theoretical point of view, this feature ensures the soundness of the LLproof formalism (w.r.t. a known theory), whereas from a practical point of view, this provides a (local) guarantee of Zenon's implementation. But especially, in the context of the Focal system [15], this allows us to produce homogeneous Coq code (where the Coq proofs built by Zenon are reinserted in the Coq specifications generated by the Focal compiler), that can be fully verified by Coq.

4.1 Translation

The translation consists in producing, from proofs provided in LLproof format, proofs in the theory of the theorem prover we chose to perform the validation, Quantifier rules

$$\frac{\Gamma, P(c), \exists x\ P(x) \vdash \bot}{\Gamma, \exists x\ P(x) \vdash \bot}\ \exists \qquad \frac{\Gamma, \neg P(c), \neg\forall x\ P(x) \vdash \bot}{\Gamma, \neg\forall x\ P(x) \vdash \bot}\ \neg\forall \text{ where } c \text{ is a fresh constant}$$

$$\frac{\Gamma, P(t), \forall x\ P(x) \vdash \bot}{\Gamma, \forall x\ P(x) \vdash \bot}\ \forall \qquad \frac{\Gamma, \neg P(t), \neg\exists x\ P(x) \vdash \bot}{\Gamma, \neg\exists x\ P(x) \vdash \bot}\ \neg\exists \text{ where } t \text{ is any closed term}$$

Special rules

$$\frac{\Delta, t_1 \neq u_1 \vdash \bot \qquad \ldots \qquad \Delta, t_n \neq u_n \vdash \bot}{\Gamma, P(t_1, \ldots, t_n), \neg P(u_1, \ldots, u_n) \vdash \bot}\ \text{pred}$$

where $\Delta = \Gamma \cup \{P(t_1, \ldots, t_n), \neg P(u_1, \ldots, u_n)\}$

$$\frac{\Delta, t_1 \neq u_1 \vdash \bot \qquad \ldots \qquad \Delta, t_n \neq u_n \vdash \bot}{\Gamma, f(t_1, \ldots, t_n) \neq f(u_1, \ldots, u_n) \vdash \bot}\ \text{fun}$$

where $\Delta = \Gamma \cup \{f(t_1, \ldots, t_n) \neq f(u_1, \ldots, u_n)\}$

$$\frac{\Gamma, C, H \vdash \bot}{\Gamma, C \vdash \bot}\ \text{def}(\mathsf{name}, C, H)$$

if one can go from C to H by unfolding definition name.

$$\frac{\Delta, H_{11}, \ldots, H_{1m} \vdash \bot \qquad \ldots \qquad \Delta, H_{n1}, \ldots, H_{nq} \vdash \bot}{\Gamma, C_1, \ldots, C_p \vdash \bot}\ \begin{array}{l}\text{ext}(\mathsf{name}, \mathsf{args}, \\ [C_i], [H_{1j}, \ldots, H_{nk}])\end{array}$$

where $\Delta = \Gamma \cup \{C_1, \ldots, C_p\}$
name is the name of a predefined lemma s.t.
$C_1 \wedge \ldots \wedge C_p \Rightarrow \bigvee_j (\bigwedge_i H_i j)$

$$\frac{}{C \vdash \bot}\ \text{lemma}(\mathsf{name}, \mathsf{args})$$

if C is the conclusion associated with name in the list of previously-done proofs. Arguments args are the parameters of name.

Fig. 4. LLproof rules (part 2)

which is Coq in our case. This translation is not straightforward for some reasons inherent to the underlying theory of Coq, but also to Coq itself. One of them is that the theory of Coq is based on an intuitionistic logic, i.e. without the excluded middle, whereas LLproof is purely classical. To adapt the theory of Coq to LLproof, we have to add the excluded middle and the resulting theory is still consistent. But Coq does not provide a genuine classical mode (even if the classical library is loaded), i.e. with a classical sequent allowing several propositions on the right hand side, so that proofs must still be completed using an intuitionistic sequent (with only one proposition to the right hand side) and the excluded middle must be added as an axiom. Such a system does not correspond to Gentzen's LK sequent calculus, which is normally used when doing classical proofs, but rather to Gentzen's LJ sequent calculus provided with an explicit excluded middle rule. From a practical point of view, doing proofs in this system is more difficult than in LK (where the right contraction rule is a good shortcut), but in our case this has little effect because all our proofs are produced automatically.

Beyond predicate calculus in general, Zenon, like most of first order automated deduction systems, considers equality as a special predicate and uses specific rules to deal with it. Thus, to translate equality proofs correctly, we have to extend the theory of LJ with equational logic rules. Such a theory will be called $\mathsf{LJ}_{\mathsf{eq}}$ (due to space constraints, we cannot give the corresponding rules, but this theory is quite standard and can be found in literature).

We have the following theorem:

Theorem 2 (Soundness of LLproof w.r.t. $\mathsf{LJ}_{\mathsf{eq}}$). *Every sequent provable in* LLproof *has a proof in* $\mathsf{LJ}_{\mathsf{eq}}$.

Proof. The proof is done by induction over the structure of the proof of the sequent in LLproof. Due to space constraints, we cannot detail the many cases, but as an example, we can consider the translation of the $\neg\wedge$ rule of LLproof, which is the following:

$$\dfrac{\dfrac{\pi_1}{\Gamma, \neg(P \wedge Q), \neg P \vdash \bot} \qquad \dfrac{\pi_2}{\Gamma, \neg(P \wedge Q), \neg Q \vdash \bot}}{\Gamma, \neg(P \wedge Q) \vdash \bot}\ \neg\wedge$$

where π_1 and π_2 are respectively the proofs of $\Gamma, \neg(P \wedge Q), \neg P \vdash \bot$ and $\Gamma, \neg(P \wedge Q), \neg Q \vdash \bot$.

This rule is translated in $\mathsf{LJ}_{\mathsf{eq}}$ as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\widehat{\pi_1}}{\Gamma, \neg(P \wedge Q), \neg P \vdash \bot}}{\Gamma, \neg(P \wedge Q) \vdash \neg\neg P}\ \neg_{\mathsf{right}}}{\Gamma, \neg(P \wedge Q) \vdash P}\ \mathsf{em} \qquad \dfrac{\dfrac{\dfrac{\widehat{\pi_2}}{\Gamma, \neg(P \wedge Q), \neg Q \vdash \bot}}{\Gamma, \neg(P \wedge Q) \vdash \neg\neg Q}\ \neg_{\mathsf{right}}}{\Gamma, \neg(P \wedge Q) \vdash Q}\ \mathsf{em}}{\dfrac{\Gamma, \neg(P \wedge Q) \vdash P \wedge Q}{\Gamma, \neg(P \wedge Q), \neg(P \wedge Q) \vdash \bot}\ \neg_{\mathsf{left}}}\ \wedge_{\mathsf{right}}}{\Gamma, \neg(P \wedge Q) \vdash \bot}\ \mathsf{cont}$$

where $\widehat{\pi_1}$ and $\widehat{\pi_2}$ are the translated proofs of π_1 and π_2, em the excluded middle rule, cont the left contraction rule, $\neg/\wedge_{\mathsf{right}}$ the right rule for $\neg/\wedge$, and $\neg_{\mathsf{left}}$ the left rule for $\neg$.

4.2 Implementation

General scheme The proof of Theorem 2 allows Zenon to produce Coq proofs from proofs in LLproof, since $\mathsf{LJ}_{\mathsf{eq}}$ is included in the underlying theory of Coq, i.e. the Calculus of Inductive Constructions (CIC for short). Actually, we have two kinds of translations: a first one generating proof scripts and a second one directly generating proof terms (thanks to the Curry-Howard isomorphism capability of Coq). In both translations, in order to factorize proofs and especially to minimize the size of the produced proofs, the idea is not to build the proof scripts corresponding to the translated rules, but to prove a lemma for each translated rule once and for all (a macro tactic in $\mathcal{L}_{tac}$ is not appropriate because the body of these macros is rerun each time a translated rule is used in a proof). Thus, the generated Coq proofs are just sequences of applications of these lemmas, and they are not only quite compact, but also quite efficient in the sense that the corresponding Coq checking is fast. For instance, if we consider the $\neg\wedge$ rule of LLproof translated in the proof of Theorem 2, the associated Coq lemma is the following:

```
Lemma zenon_notand : forall P Q : Prop,
  (~P → False) → (~Q → False) → (~(P ∧ Q) → False).
```

As an example of complete Coq proof produced by Zenon and involving the previous lemma, let us consider the proof of $\neg(P \wedge Q) \Rightarrow \neg P \vee \neg Q$, where P and Q are two propositional variables. For this proof, Zenon is able to generate a Coq proof script as follows:

```
Parameters P Q : Prop.
Lemma de_morgan : ~(P ∧ Q) → ~P ∨ ~Q.
Proof.
  apply NNPP. intro G.
  apply (notimply_s _ _ G). zenon_intro H2. zenon_intro H1.
  apply (notor_s _ _ H1). zenon_intro H4. zenon_intro H3.
  apply H3. zenon_intro H5.
  apply H4. zenon_intro H6.
  apply (notand_s _ _ H2);
    [ zenon_intro H8 | zenon_intro H7 ].
  exact (H8 H6).
  exact (H7 H5).
Qed.
```

where NNPP is the excluded middle, *rule*_s (where *rule* is notimply, notor, etc) a definition which allows us to apply partially the corresponding lemma *rule* providing the arguments at any position (not only beginning by the leftmost position), and zenon_intro a macro tactic to introduce (in the context) hypotheses with possibly fresh names if the provided names are already used.

For the same example, Zenon is also able to directly produce the following proof term (without the help of Coq):

```
Parameters P Q : Prop.
Lemma de_morgan : ~(P ∧ Q) → ~P ∨ ~Q.
```

```
Proof.
  exact (NNPP _ (fun G : ~(~(P ∧ Q) → ~P ∨ ~Q) ⇒ (notimply
    (~(P ∧ Q)) (~P ∨ ~Q) (fun (H5 : ~(P ∧ Q))
    (H8 : ~(~P ∨ ~Q)) ⇒ (notor (~P) (~Q) (fun (H6 : ~~P)
    (H7 : ~~Q) ⇒ (H7 (fun H1 : Q ⇒ (H6
    (fun H3 : P ⇒ (notand P Q (fun H4 : ~P ⇒ (H4 H3))
    (fun H2 : ~Q ⇒ (H2 H1)) H5)))))) H8)) G))).
Qed.
```

As said in the introduction, this possibility of generating proof terms is particularly important in the sense that Zenon verifies the De Bruijn criterion [1], i.e. it generates a proof format that can be checked by Coq but also independently, by means of another program or proof system which implements the same type theory. For example, as an alternative and with an appropriate printer, we can imagine using the Matita [16] theorem prover, which has the same underlying theory (CIC) as Coq.

Difficulties In this implementation, we have to be aware of some difficulties. One of them is that we plug first order logic, which is a priori untyped, into a typed calculus (CIC). To deal with this problem, we consider that we have a mono-sorted first order logic, of sort U, and we provide types to variables, constants, predicates and functions explicitly (the type inference offered by Coq does not always allow us to guess these types). Obviously, this must be done only when dealing with purely first order propositions, but can be avoided with propositions coming from Coq or Focal, which are possible inputs for Zenon, since these systems are strongly typed and Zenon keeps the corresponding type information (this is possible since Zenon works in a non-destructive way, see Section 2); in this case, we generally have a multi-sorted first order logic.

Another difficulty, probably deeper, is that mono/multi-sorted first order logic implicitly supposes that each sort is not empty, while in the CIC, types may be not inhabited. This problem is fixed by skolemizing the theory and considering at least one element for each sort, e.g. E for U. Thus, for example, it is possible to prove Smullyan's drinker *paradox* with Zenon as follows:

```
Parameter U : Set.
Parameter E : U.
Parameter d : U → Prop.
Lemma drinker_paradox :
  exists X : U, (d X) → forall Y : U, (d Y).
Proof.
  apply NNPP. intro G.
  apply G. exists E. apply NNPP. zenon_intro H3.
  apply (notimply_s _ _ H3). zenon_intro H5. zenon_intro H4.
  apply H4. zenon_intro T0. apply NNPP. zenon_intro H6.
  apply G. exists T0. apply NNPP. zenon_intro H7.
  apply (notimply_s _ _ H7). zenon_intro H8. zenon_intro H4.
  exact (H6 H8).
Qed.
```

5 Using Zenon in practice

In this section, we consider the effectiveness of Zenon through benchmarks and applications. The interested reader can get the distribution of Zenon, which is available either as part of the Focal environment at `http://focal.inria.fr/`, or directly (as a separate tool) at `http://focal.inria.fr/zenon/`.

5.1 Benchmarks

In order to see how Zenon fares w.r.t. available first-order theorem provers, we benchmarked it against parts of the latest TPTP library [12] release (v3.2.0). The Zenon runs were made on an Apple Power Mac Core 2 Duo 2 GHz, with Zenon's default timeout of 5 min and size limit of 400 Mbytes. The set of TPTP syntactic problems SYN was chosen as representative of Zenon's typical target problems, and indeed we get good results. We also tried Zenon against the problems of the FOF category for the latest CASC competition [11].

Problems	Proof found	No proof		
		time	size	other
SYN theorems (282)	264	10	7	1
CASC-J3 (150)	48	46	56	0

Some of the formulas proved by Zenon in CASC have a rather high rating, such as SWV026+1 (0.79), SWV038+1 (0.71), or MSC010+1 (0.57). This last one consists in proving $\neg\neg P$, assuming P, where P is a large first-order formula. Thanks to the tableau method, Zenon does not need to decompose the formula, and the proof is found immediately. All the proofs found by Zenon were verified by Coq.

5.2 The EDEMOI project

In the framework of the EDEMOI[4] [10] project, Zenon was used to certify the formal models of two regulations related to airport security: the first one is the international standard Annex 17 produced by the International Civil Aviation Organization (ICAO), an agency of the United Nations; the second one is the European Directive Doc 2320 produced by the European Civil Aviation Conference (ECAC) and which is supposed to refine the first one at the European level. The EDEMOI project aims to integrate and apply several requirements engineering and formal methods techniques to analyze standards in the domain of airport security. The novelty of the methodology developed in this project, resides in the application of techniques, usually reserved for safety-critical software, to the domain of regulations (in which no implementation is expected).

[4] The EDEMOI project is supported by the French National "Action Concertée Incitative Sécurité Informatique".

The two formal models of the two considered standards were completed using the Focal [15] environment and can be found in [3], where the reader can also find a brief description of Focal. In this formalization, Zenon was used to prove the several identified theorems ensuring the correctness and the completeness of both regulations (consistency was not studied formally). Concretely, the development represents about 10,000 lines of Focal and 200 proofs (2 years to be completed). Regarding the validation part, Zenon allowed us to discharge most of the proof obligations automatically (about 90% of them). Actually, Zenon also succeeded in completing the remaining 10% automatically but beyond the default timeout (set to 3 min in Focal). This tends to show that Zenon is quite appropriate when dealing with abstract specifications (no concrete types and very few definitions). Zenon also helped us to study the consistency of the regulations from a practical point of view. The idea is to try to derive False from the set of security properties and to let Zenon work on it for a while. If the proof succeeds then we have a contradiction, otherwise we can only have a certain level of confidence. This approach may seem rather naive but appears quite pertinent when used to identify the correlation between the several security measures according to specific attack scenarios. The principle is to falsify an existing hypothesis or to add an inconsistent hypothesis and to study its impact over the entire regulation, i.e. where the potential conflicts are located and which security properties are concerned. For more information regarding this experiment with Zenon, the reader can refer to [4].

6 Conclusion

Zenon is an experiment in progress, but we already have a reasonably powerful prover (see the benchmarks) that can output actual proofs in Coq format (proof scripts or proof terms) for use in a skeptic-style system, such as the Focal environment for example. In addition, the help provided by Zenon in the EDEMOI project framework, where most of the proofs were discharged (and even all the proofs with an extended timeout), tends to show how this tool is appropriate for real-world applications, so that we can be quite optimistic regarding its use, in particular in the context of Focal.

Future work will focus on improving the handling of metavariables in order to get better heuristics for finding the right instantiations, and on implementing some theory-based reasoning by using the extension mechanism of Zenon. Amongst other extensions, we plan to add a theory of arithmetic, but also reasoning by induction (this feature is under development), which is crucial when dealing with specifications close to implementations involving, in particular, concrete datatypes. Finally, it is quite important to apply Zenon to other case-studies, not only to get a relative measure of its automation power, but also to understand the practical needs of automation. For example, proofs provided by Zenon are progressively integrated into the Focal standard library [15] (which mainly consists of a large kernel of Computer Algebra), and a certified development regarding security policies [6] is in progress.

References

1. Henk Barendregt and Erik Barendsen. Autarkic Computations in Formal Proofs. *Journal of Automated Reasoning (JAR)*, 28(3):321–336, 2002.
2. Marc Bezem, Dimitri Hendriks Hendriks, and Hans de Nivelle. Automated Proof Construction in Type Theory Using Resolution. *Journal of Automated Reasoning (JAR)*, 29(3–4):253–275, 2002.
3. David Delahaye, Jean-Frédéric Étienne, and Véronique Viguié Donzeau-Gouge. Certifying Airport Security Regulations using the Focal Environment. In *Formal Methods (FM)*, volume 4085 of *Lecture Notes in Computer Science (LNCS)*, pages 48–63, Hamilton, Ontario (Canada), August 2006. Springer.
4. David Delahaye, Jean-Frédéric Étienne, and Véronique Viguié Donzeau-Gouge. Reasoning about Airport Security Regulations using the Focal Environment. In *International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, Paphos (Cyprus), November 2006.
5. Joe Hurd. Integrating Gandalf and HOL. In Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin, and Laurent Théry, editors, *Proceedings of Theorem Proving in Higher Order Logics (TPHOLs), Nice (France)*, volume 1690 of *Lecture Notes in Computer Science (LNCS)*, pages 311–322. Springer, September 1999.
6. Mathieu Jaume and Charles Morisset. Formalisation and Implementation of Access Control Models. In *Information Assurance and Security (IAS), International Conference on Information Technology (ITCC)*, pages 703–708, Las Vegas (USA), April 2005. IEEE CS Press.
7. Albert C. Leisenring. *Mathematical Logic and Hilbert's ϵ-Symbol*. MacDonald Technical and Scientific, London, 1969. ISBN 0356026795.
8. William McCune and Olga Shumsky. System Description: IVY. In David A. McAllester, editor, *Proceedings of the 17^{th} International Conference on Automated Deduction (CADE-17), Pittsburgh (PA, USA)*, volume 1831, pages 401–405. Lecture Notes in Computer Science (LNCS), June 2000.
9. Laurence C. Paulson and Kong Woei Susanto. Source-Level Proof Reconstruction for Interactive Theorem Proving. In Jens Brandt, editor, *Theorem Proving in Higher Order Logics (TPHOLs)*, Lecture Notes in Computer Science (LNCS). Springer, September 2007.
10. The EDEMOI Project, 2003. `http://www-lsr.imag.fr/EDEMOI/`.
11. Geoff Sutcliffe. CASC-J3 - The 3^{rd} IJCAR ATP System Competition. In Ulrich Ulrich Furbach and Natarajan Shankar Shankar, editors, *International Joint Conference on Automated Reasoning (IJCAR)*, volume 4130 of *Lecture Notes in Computer Science (LNCS)*, pages 572–573. Springer, August 2006.
12. Geoff Sutcliffe and Christian B. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning (JAR)*, 21(2):177–203, 1998.
13. The Coq Development Team. Coq, *version 8.1*. INRIA, November 2006. Available at: `http://coq.inria.fr/`.
14. The Cristal Team. Objective Caml, *version 3.10*. INRIA, May 2007. Available at: `http://caml.inria.fr/`.
15. The Focal Development Team. Focal, *version 0.3.1*. CNAM/INRIA/LIP6, May 2005. Available at: `http://focal.inria.fr/`.
16. The HELM Team. Matita, *version 0.1.0*. Computer Science Department, University of Bologna, July 2006. Available at: `http://matita.cs.unibo.it/`.