HABILITATION À DIRIGER LES RECHERCHES

UNIVERSITÉ PIERRE ET MARIE CURIE
CONSERVATOIRE NATIONAL DES ARTS ET MÉTIERS

ASSISTING USERS OF PROOF ASSISTANTS

DAVID DELAHAYE

DEFENDED ON DECEMBER 9, 2010

– JURY –

MARC POUZET	PRESIDENT
GILLES DOWEK	REVIEWER
HERMAN GEUVERS	REVIEWER
JULIO RUBIO	REVIEWER
CATHERINE DUBOIS	SUPERVISOR
THÉRÈSE HARDIN	EXAMINER
MARIE-LAURE POTET	EXAMINER

ACKNOWLEDGMENTS

First of all, I would like to thank Marc Pouzet for accepting to be the president of my jury. In the same way, I am very grateful to Gilles Dowek, Herman Geuvers, and Julio Rubio for being the reviewers of this document. Their several comments have been of great help when preparing the final version of this document. I also wish to express all my gratitude to Catherine Dubois, Thérèse Hardin, and Marie-Laure Potet for acting as examiners of my jury.

A very special thank goes to Catherine Dubois. More than my supervisor, she is especially the person who succeeded in focusing my energy on consistent objectives, and therefore making me express the best of myself. Without her help, I could not present half of the work described in this document. Another special thank goes to Véronique Viguié Donzeau-Gouge, Thérèse Hardin, and Renaud Rioboo. They have never hesitated to put their trust in me, and provided me with an invaluable help during all these years.

In these acknowledgments, it is difficult to forget to mention the CPR and Focalize teams, as well as the CEDRIC laboratory. If human relations are probably the most difficult link to manage, I have to consider myself quite lucky for finding a so pleasant environment, where an exceptionally good atmosphere prevails at all times.

Last but not least, and even if no word is strong enough to express my gratefulness to them, many thanks to all my family, who has been of constant support.

ABSTRACT

This document proposes to present the results of more than ten years of research by the author, which aim to improve several techniques related to theorem proving. These improvements are guided by the following leitmotiv: how to make theorem proving easier to use? This work starts from the self-evident fact that theorem proving represents actually a very thin part of the spectrum of formal methods, especially compared to model checking. Automation is probably one of the main causes, but some other reasons may also be found elsewhere, such as in the way of building specifications or interacting with theorem provers. Thus, this document is organized around these three topics, that are structuring, automating, and communicating.

The first part of this document focuses on the specification languages involved in theorem provers. In particular, we make use of the Focal language, which allows us to build certified applications. To understand how the design features of Focal can be appropriate in practice, a significant application, realized in the framework of the EDEMOI project, is described, and which consists of the formalization of airport security regulations in the domain of civil aviation. In the idea of clearly identifying specifications and implementations, we present, in the context of the Coq proof assistant, another work which aims to execute inductive relations. Finally, we discuss the notion of reuse with an experiment, which consists in retrieving information, typically theorems, in a proof library using types as keys and up to isomorphisms.

The second part of this document is devoted to automation in theorem proving. First, the problem is handled in terms of power of automation, introducing a proof dedicated meta-language, which is intended to tailor automation in the framework of Coq. Next, we describe several experiments which aim to import computations from computer algebra systems into proof assistants in a pure skeptical way (i.e. verifying the soundness of the computations). Among these experiments, some interfaces have been realized between Coq and Maple, as well as between Focal and Axiom. Lastly, in the same idea of skeptical computations, we propose to adapt this idea to automated deduction with two contributions related to the Zenon automated theorem prover. These contributions respectively consist in verifying the proofs produced by Zenon by generating Coq proofs, and validating supplementary rules involved in applications developed using the B method by means of Zenon proofs.

In the third part of this document, we aim to draw attention to several means of communicating with theorem provers. Regarding the input language and in the framework of Coq, we propose a language designed to describe proofs and intended to be style-independent. Concerning the output language, we present a transformation from Focal specifications to UML models, which is an appropriate means of producing comprehensible documents for end-users, such as the certification authorities in the context of the EDEMOI project. Finally, still considering output languages, we introduce a scheme of compilation for Focal, which is based on the notion of modules and allows us to get traceability between Focal specifications and compiled codes.

CONTENTS

1 INTRODUCTION 1
- 1.1 Formal Methods 1
- 1.2 Theorem Proving 1
- 1.3 Improving Theorem Proving 2
- 1.4 Outline of the Document 3

2 STRUCTURING 5
- 2.1 Certification of Airport Security Regulations 6
  - 2.1.1 The EDEMOI Project 6
  - 2.1.2 Results and Analyses 7
  - 2.1.3 Appropriateness of Focal 9
- 2.2 Code Generation from Specifications 10
  - 2.2.1 Functional Extraction in Coq 10
  - 2.2.2 Mode Consistency Analysis 11
  - 2.2.3 Code Generation 13
  - 2.2.4 Extension to Focalize 14

3 AUTOMATING 17
- 3.1 Deduction and Computer Algebra 18
  - 3.1.1 A Maple Mode for Coq 18
  - 3.1.2 Proofs over Algebraically Closed Fields 22
  - 3.1.3 Tests over Real Closed Fields 24
- 3.2 Certification of Automated Proofs 30
  - 3.2.1 Validation of Zenon Proofs 30
  - 3.2.2 Validation of B Proofs from Zenon 33

4 COMMUNICATING 37
- 4.1 From Focal Specifications to UML Models 38
  - 4.1.1 The Need for Documentation 38
  - 4.1.2 Profile and Transformation Rules 39
  - 4.1.3 Airport Security Regulations 41
- 4.2 A Module-Based Model for Focal 43
  - 4.2.1 High-Level Compilation Schemes 43
  - 4.2.2 Module-Based Compilation 44

5 CONCLUSION 49
- 5.1 Achievements 49
- 5.2 Perspectives 50

A THE FOCAL ENVIRONMENT 55
- A.1 What is Focal? 55
- A.2 Specification: Species 56

A.3 Implementation: Collection 57
A.4 Certification: Proving with Zenon 57
A.5 Further Information 58

B FORMER CONTRIBUTIONS 59
B.1 Information Retrieval in Proof Libraries 59
B.1.1 Use of Type Isomorphisms 60
B.1.2 Application to Proof Libraries 61
B.2 A Proof Dedicated Meta-Language 63
B.2.1 Evolution of Meta-Languages 63
B.2.2 The $\mathcal{L}_{tac}$ Meta-Language 64
B.2.3 Future of Meta-Languages 65
B.3 Free-Style Theorem Proving 67
B.3.1 The Several Proof Styles 67
B.3.2 The $\mathcal{L}_{pdt}$ Proof Language 68
B.3.3 The Next Proof Languages 69

C STUDENT SUPERVISION 71
C.1 PhD Students 71
C.1.1 Jean-Frédéric Étienne (2004-2008) 71
C.1.2 Pierre-Nicolas Tollitte (2009-now) 71
C.1.3 Mélanie Jacquel (2010-now) 71
C.2 Master and Engineering Students 72
C.2.1 Yuan Gang (2003) 72
C.2.2 Nicolas Bertaux (2008) 72
C.2.3 Pierre-Nicolas Tollitte (2009) 72
C.2.4 Sanaa Toumi (2009) 73
C.2.5 Benjamin Lalière (2009) 73

D PUBLICATION ADDENDUM 75
D.1 Paper 1: Airport Security Regulations in Focal 75
D.2 Paper 2: Executing Inductive Relations 92
D.3 Paper 3: A Maple Mode for Coq 109
D.4 Paper 4: The Zenon Automated Theorem Prover 137
D.5 Paper 5: From Focal to UML 153

Bibliography 163

"How thoroughly it is ingrained in mathematical science that every real advance goes hand in hand with the invention of sharper tools and simpler methods which, at the same time, assist in understanding earlier theories and in casting aside some more complicated developments."

David Hilbert (1862 - 1943).

1

INTRODUCTION

1.1 FORMAL METHODS

In computer science, formal methods are mathematically-based techniques used for the specification, development and verification of software and hardware systems. The use of formal methods is mainly motivated by the notions of safety or security which must be ensured in the design of some critical systems, typically embedded systems for example. Generally, formal methods are considered to require a high cost of development, so that they are rather used in systems where "critical" means that human life is at stake. Thus, formal methods are quite popular in avionics and aerospace systems, and also in the domain of health, with applications such as heart monitors.

Regarding verification, formal methods offer a wide panel of possibilities which correspond to different techniques. These possibilities can be roughly seen as being part of two approaches. The first one is model checking, which consists of a systematically exhaustive exploration of the mathematical model (finite or infinite). Usually, this consists in exploring all states and transitions in the model, by using smart and domain-specific abstraction techniques to consider whole groups of states in a single operation and reduce computing time. Model checking is, by definition, automatic and is therefore of great benefit to engineering industry where it is desirable that human assistance may be avoided. However, model checking is not always effective as the combinatorial explosion may be significant and the abstraction techniques not enough efficient to sufficiently decrease the size of the corresponding model. Another approach to formal verification is theorem proving, which we propose to focus on in this document.

1.2 THEOREM PROVING

Theorem proving aims to ensure properties using logical deduction. There exist many systems using theorem proving and which can be distinguished according to different characteristics. Some of them use first order logic, like the B method [1] or ACL2 [123], some others use higher order logic, like Coq [129] or HOL [135]. Some systems are based on classical logic, like PVS [146] or HOL, some others are based on intuitionistic logic (consequently benefiting from the Curry-Howard capability), like ALF [124] or NuPRL [141]. These systems can also be distinguished according their use of set theory, like the B method or Mizar [140], or their use of type theory, like Coq or PVS.

Some systems are interactive, like LEGO [137] or HOL, some others are automated like Vampire [154] or Gandalf [121]. Finally, some systems cannot be characterized according to the previous features, since they are logical frameworks (which can be instantiated with different logics), like Isabelle [136] or LF [151].

Compared to model checking, theorem proving has the advantage to produce not only a statement of validity but also an evidence of this validity. Admittedly, model checkers behave more like oracles, producing results which must be verified by other means. Nevertheless, theorem proving suffers a lack of automation, which is generally only partial and requires the user to get a sufficient understanding of the system to validate manually the uncompleted proofs. Depending on the system, this last point may expect a high level of expertise. Thus, automation is probably one of the most crucial problems of theorem proving and we can easily understand why it is a very active research topic, in particular in the domains of first order logic, induction and rewriting. But theorem proving can also be improved at some other levels (higher or lower in the development process), such as the structure of specifications (which may influence the way of building proofs) or the way of communicating with the prover (either to specify or prove properties, or to produce documentation for completed specifications).

1.3 IMPROVING THEOREM PROVING

This document proposes to present the results of more than ten years of research by the author (including his Master/PhD theses started in 1997), which aim to improve several techniques related to theorem proving. These improvements are guided by the following leitmotiv: how to make theorem proving easier to use? This work starts from the self-evident fact that theorem proving represents actually a very thin part of the spectrum of formal methods, especially compared to model checking. As said previously, automation is probably one of the main causes, but some other reasons may also be found elsewhere, such as in the way of building specifications or interacting with theorem provers. Thus, this document is organized around these three topics, that are structuring (Chapter 2), automating (Chapter 3), and communicating (Chapter 4).

The idea behind these three topics into which we propose to go further is to draw what we believe an ideal theorem prover could look like. Basically, such a theorem prover should provide a language allowing us to write highly structured specifications, as well as appropriate means to combine them. In addition, such high-level specifications are nothing without a suitable automation, which should offer not only a significant set of automated deduction procedures, but also dedicated languages, tools or interfaces to enhance this automation. Finally, the theorem prover and end-users should be able to smoothly communicate in a transparent way and in both directions, i.e. when providing specifications and proofs to the theorem prover, and when trying to understand and interpret already formalized and compiled specifications. This document tries to humbly bring some elements of answer in this task consisting in asymptotically building this ideal theorem prover.

1.4 OUTLINE OF THE DOCUMENT

The first part of this document focuses on the specification languages involved in theorem provers (see Chapter 2). In particular, we aim to show how the way of specifying is essential to facilitate the way of proving. To do so, we make use of the Focal language [13, 132], which allows us to build certified applications. In this language, there is a neat separation between specifications (which may be quite abstract) and implementations (which are concrete). Between specifications and implementations, there exists a notion of refinement, which is actually inheritance coming from the object-oriented programming paradigm. To understand how the design features of Focal can be appropriate in practice, a significant application, realized in the framework of the EDEMOI project [131], is described, and which consists of the formalization of airport security regulations in the domain of civil aviation. In the same idea of clearly identifying specifications and implementations, we present, in the context of the Coq proof assistant [129], a work which aims to execute inductive relations. In fact, inductive relations are more than specifications and also contain implementations, which can be extracted to be executed. Finally, along the same line of providing reusability as offered by the inheritance feature of Focal, we discuss the notion of reuse at another level, which is to retrieve information, typically theorems, in a proof library using types as keys and up to isomorphisms. This work deals with specific features of some type theories, such as polymorphism, dependent types or strong sum types, and has been implemented in an earlier version of Coq.

The second part of this document is devoted to automation in theorem proving (see Chapter 3). First, the problem is handled in terms of power of automation, introducing a proof dedicated meta-language, called $\mathcal{L}_{tac}$, which is intended to tailor automation in the framework of Coq. This language appears quite appropriate for small and local automation, but also for significant tactics, such as the fully reflexive tactic "field", which aims to prove equalities over fields. Next and as a result of this work, we discuss the notion of computation, which may be quite inefficient when performed in an autarkic way (i.e. within the theorem prover). Therefore, we propose a method to externalize some computations using more suitable tools, such as computer algebra systems, in a pure skeptical way (i.e. verifying the soundness of the computations). An experiment has been conducted between Coq and Maple [138] for computations over fields, while the corresponding computations were verified by the tactic "field". This experiment, called the Maple mode for Coq, has been extended to deal with gcd computations over polynomials and has allowed us to implement a quantifier elimination procedure for algebraically closed fields. In the continuity of this procedure and still in the idea of benefiting from external computations as oracles, another procedure has been designed in the context of the Focal environment to test the validity of properties over real closed fields using the computation of cylindrical algebraic decomposition performed by a routine of Axiom [125]. Lastly, in the same idea of skeptical computations, we propose to adapt this idea to automated deduction with two contributions related to the Zenon automated theorem prover [24], which is used by Focal in particular. The first contribution deals with the proofs produced by Zenon, and which are translated into Coq proofs for checking. The second contribution

consists in validating supplementary rules involved in applications developed using the B method [1] by means of Zenon proofs, which are translated back to B proofs.

In the third part of this document, we aim to draw attention to several means of communicating with theorem provers (see Chapter 4). Regarding the input language and in the framework of Coq, we propose a language, designed to describe proofs and called $\mathcal{L}_{pdt}$. This language has the advantage to be style-independent in the sense that it gathers the three well-identified proof styles, i.e. the procedural, declarative and proof-term styles. Concerning the output language, we present a transformation from Focal specifications to UML models [144], which has been formally described and proved sound. This transformation appears quite appropriate as a means of automatic documentation and especially as a means of producing comprehensible documents for end-users. In particular, in the context of the EDEMOI project, we can hopefully expect that documents in UML are a good basis to converse with certification authorities. Finally, still considering output languages, we introduce another scheme of compilation for Focal, which is based on the notion of modules. The Focal compiler is able to produce not only OCaml code [128] for execution, but also Coq code for certification. As both OCaml and Coq offer the notion of modules, this scheme of compilation is quite possible and arises as a higher level alternative to the flat current scheme using records. In particular, this compilation allows us to get traceability between Focal specifications and compiled codes.

The last part of this document is the conclusion, which will provide not only a summary of the different contributions described previously, but also several sets of perspectives. Some perspectives in the short and medium terms will have been already presented in the parts introduced above, and the conclusion will propose long term and more ambitious perspectives.

It should be noted that Appendix D provides 5 publications in their entirety, and which intend to support the three previous chapters described above. More precisely, papers of Section D.1 and D.2 are related to Chapter 2, papers of Section D.3 and D.4 to Chapter 3, and finally paper of Section D.5 to Chapter 4.

2

STRUCTURING

> "There are two ways of constructing a software design; one way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."
>
> *Charles Antony Richard Hoare*
> *(aka Tony Hoare).*

This chapter aims to focus on the importance of structuring specifications through three distinct and well identified contributions by the author. The first one is related to the Focal environment [13, 132], which allows us to build certified applications by means of a language clearly distinguishing specifications from implementations, and in an incremental way through the use of inheritance coming from the object-oriented programming paradigm and which provides an appropriate notion of refinement. The reader can refer to Appendix A for an overview of Focal. As exposed in this overview of Focal, Focal was initially designed in the idea of developing effective mathematics with a structuring framework appropriate for this kind of applications and in particular for the several involved notions (i.e. sets, algebraic structures, etc). To show how the design features offered by Focal can be also suitable in other contexts, we present the development of a significant application, which consists of the formalization of airport security regulations in the domain of civil aviation and which has been realized in the framework of the EDEMOI project [131]. The second contribution is actually set in a similar context where specifications are intended to be separated from implementations, and aims to generate purely functional code from inductive relations. This work initially experimented in the Coq proof assistant [129] is currently being adapted to the Focalize environment [133] (successor of Focal). Finally, the last contribution resides in providing another notion of reuse than the one induced by the inheritance mechanism of Focal, and which consists in retrieving information, basically theorems, in proof libraries using types as keys and up to isomorphisms. In particular, a search procedure has been developed in a calculus including polymorphism, dependent types and strong sum types, and has been implemented in an earlier version of Coq. Due to lack of space and as this last contribution is actually more former than the two first ones, it is described in Section B.1 of Appendix B with some perspectives relying on some current trends.

2.1 CERTIFICATION OF AIRPORT SECURITY REGULATIONS

Contribution in collaboration with V. Viguié Donzeau-Gouge and J.-F. Étienne (PhD student; see Subsec.C.1.1 of Appx. C). CPR team, CEDRIC (CNAM), Paris (France), 2004-2008. Published in [54, 55, 56, 60, 58] (see Sec. D.1 of Appx. D).

This first contribution is focused on the Focal environment and aims to provide some elements which tend to show that Focal is an appropriate tool for real-world applications. In particular, we present the results of a 4 year study, which consisted in certifying airport security regulations using Focal and which was conducted in the framework of the EDEMOI project.

2.1.1 *The EDEMOI Project*

The security of civil aviation is governed by a series of international standards and recommended practices that detail the responsibilities of the various stake-holders (states, operators, agents, etc). These documents are intended to give the specifications of procedures and artifacts which implement security in airports, aircraft and air traffic control. A key element to enforce security is the conformance of these procedures and artifacts to the specifications. However, it is also essential to ensure the correctness, completeness and consistency of the specifications. Standards and recommended practices are usually written in natural language in order to be easily understood and adopted by a large number of stake-holders. Nevertheless, the normative documents are generally of voluminous size, ambiguous and often open to interpretation. Moreover, it is very difficult to automatically process natural language documents in search for inconsistencies. All these problems highlight the lack of a formal drafting process and this is where modeling techniques can help. Recent work [90, 2] has shown that there is an increased interest in providing automated and systematic support to reason about regulations due to the growing complexity of safety and security requirements.

In this section, we report on an experience which consists in building and analyzing the formal models of two standards 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 Annex 17 at the European level. This formalization was realized using the Focal environment [13, 132] (see Appendix A), and within the framework of the EDEMOI project[1] [131]. The EDEMOI project aims to integrate and apply several requirements engineering and formal methods techniques to analyze regulations in the domain of airport security. The methodology of this project may be considered as original in the sense that it tries to apply techniques, usually reserved to critical software, to the domain of regulations (in which no implementation is expected). The project used a two-step approach. In the first step, standards described in natural language were analyzed in order to extract security properties and to elaborate a

1 The EDEMOI project was supported by the French National "Action Concertée Incitative Sécurité Informatique".

conceptual model of the underlying system [93]. The second step, which this work is part of, consisted in building formal models and to analyze/verify these models by different kinds of formal tools.

In this project, we had several motivations. First, from the formalization of the two standards (previously mentioned), it was expected to improve the quality of the normative documents and hence to increase the efficiency of the conformity assessment procedure. Second, thanks to this significant formalization, another motivation was to validate the design features as well as the reasoning support offered by Focal, and to extend this environment if required by the needs of the considered modeling.

2.1.2 *Results and Analyses*

Our approach consists of three steps: first, a preliminary analysis is performed on the airport security regulations considered in this project; second, the formal models of the two standards seen previously are designed using the Focal environment; third, the obtained formal models are validated using the reasoning support of Focal. Actually, there is also a fourth step, which aims to propose an extension of Focal to produce UML models for documentation; this step is described in Section 4.1 of Chapter 4.

The preliminary analysis [54, 55, 66, 60, 58] is realized by applying a variant of the KAOS goal-oriented requirements engineering methodology [40]. In fact, unlike the original elaboration method, our aim is not to derive the requirements of an envisioned system, starting from the identification of the high-level goals to be achieved to the operations, objects and constraints to be implemented. In our case, the requirements already exist in the form of standards and recommendations that each airport facility has to comply to. Instead, the WHY and HOW elaboration tactics of the KAOS goal-oriented approach are used to put under scrutiny the process by which the airport security properties have been derived. Basically, this consists in identifying the fundamental security properties and to determine how they are decomposed into sub-properties with respect to the terminologies and concepts used in the normative documents. Similarly, a bottom-up approach (by asking WHY questions) is also considered to clearly identify the intention of each specific security property. In so doing, this allows us to unveil any implicit hypotheses that led to the formulation of the preventive security properties. The hierarchy of properties thus obtained can be used to provide an appropriate structuring framework to facilitate the traceability with the normative documents; it should be noted that the well-organized content of the standards has largely facilitated the elaboration of the hierarchy of goals, whereby the decomposition obtained almost reflects the structuring of the documents. In addition, through this framework, it would be possible to identify the impact of a particular security property over the entire regulatory system, to analyze the effects of changes on the regulations, and to verify the effectiveness of the regulations against specific attack scenarios. As mentioned previously, Doc 2320 is supposed to clarify and refine the security measures outlined in Annex 17 at the European level. This refinement is concretized by ensuring that each security property extracted from Doc 2320 is not less restrictive than or does not invalidate those obtained from Annex 17.

Once the preliminary analysis performed, the Annex 17 and Doc 2320 standards were formalized [54, 55, 66, 60, 58] using the Focal environment. This formalization consists of a general model structure obtained for Annex 17, starting from the elaboration of the domain environment to the formalization of the security properties. The Doc 2320 specification is then obtained by extending the Annex 17 model for each correspondence between the two standards. The task of modeling the Annex 17 and Doc 2320 security properties has required a significant effort. In fact, it mainly corresponds to a knowledge engineering task, whereby the domain knowledge described in the normative documents is transformed into an appropriate Focal representation. In particular, we shown that for the sake of ensuring a certain traceability with the original documents, a proper classification has to be determined for the subjects being regulated. This classification also has for purpose to properly formalize the security properties common to certain categories of subjects. Furthermore, we pointed out that the provision of an appropriate framework to reason about the regulations also required the specification of additional domain relevant constraints. These are used to clearly delimit the application context of each security property, thus eliminating any unrealistic scenarios. More importantly, various ambiguities and imprecisions were also clarified during the formalization of the identified security properties. Such inferred information tend to improve the quality of the standards by eliminating as far as possible any imprecision residing in the informal definitions of the security properties considered.

Different analyses were carried out on the formal models produced in order to validate the regulations considered [56, 66, 60, 58]. Basically, this consists in establishing the correctness and completeness of these regulations, and one way of doing so is to prove the derivability of each decomposition of security property established during the analysis phase. By correctness, we mean that the specific security properties may imply the fundamental ones. By completeness, we mean that all the specific security properties are necessary to satisfy the fundamental ones, i.e. a fundamental security property is no longer satisfied when one of its sub-properties is omitted. The correctness theorems proved during the validation step have served to clarify any remaining imprecision in the formal models. In particular, by systematically exploring the decomposition obtained for each of the different categories of prevention considered, we managed to identify a set of hidden assumptions (or omissions) that shed light on the intention of each specific security property within the entire regulation. These hidden assumptions must not be considered as failures of the regulation, but more as implicit security requirements. Thus, the validation addresses a certain form of completeness for the regulation, where every security requirement has been made explicit. It should also be noted that these hidden assumptions were identified in a pragmatic approach, i.e. by exploiting the information gathered from failed proof attempts while referring to the general context of the regulation. Regarding the validation of Doc 2320, refinement theorems allowed us to formally establish that Doc 2320 indeed refines Annex 17 at the European level. In particular, we shown that each new security property identified at the Doc 2320 level does not invalidate any of the Annex 17 properties and either induces hidden assumptions that are more restrictive than those identified at the Annex 17 level or sustains a specific security objective through a scope extension (when new categories of subjects/preventions are

identified). Moreover, under the assumption that any omitted Annex 17 property still prevails, we also provided evidence that each existing security property is either made more precise and restrictive or is preserved as is within the new application context.

The formal models produced in Focal correspond to a complete formalization of Annex 17 and Doc 2320. They consist of about 10,000 lines of code with in particular 150 species and 200 proofs. It took about 2 years to be finalized, with at least 11 attempting versions in between. This formalization is described in particular in [54], which can be found in Section D.1 of Appendix D.

2.1.3 *Appropriateness of Focal*

Regarding the Focal specification language, this experiment revealed that it appears to be well-suited for such type of application, even if it was initially designed for the development of certified computer algebra libraries. In essence, the inheritance (refinement) and parameterization (modularity) features of the language allowed us to provide an appropriate structure for the formal models that facilitates traceability with the normative documents, while providing a neat separation between the domain relevant knowledge and the formalized security properties. More precisely, with inheritance we were able to naturally represent the classification determined for the subjects being regulated. Moreover, by using inheritance as an incremental refinement mechanism, we could clearly establish the correlation between Annex 17 and Doc 2320. Finally, by coupling inheritance with the parameterization feature, we were not only able to factorize our development, but also were allowed to resolve the vocabulary differences between Annex 17 and Doc 2320.

Concerning the reasoning support of Focal, the declarative-like proof language allowed us to describe the correctness and refinement proofs in a natural way. As for the automated theorem prover Zenon, it provided us with an effective help by discharging most of the proof obligations automatically. In terms of specifications and proofs, Focal can be seen as a front-end for the Coq system, and at the beginning of the EDEMOI project, the correctness and refinement theorems were proved directly in Coq (at that time, Zenon was still highly experimental). However, any slightest change in the Focal specification resulted in redoing most of the proof obligations, which became intractable as the complexity of the formal models grown. The use of Zenon allowed us to palliate this problem. In essence, it could be used to automatically generate the corresponding Coq proof scripts in each step of the development: in the prototyping phase, providing a set of properties and definitions (to be used by Zenon) to be convinced that a given lemma is correctly formulated; in the finalizing phase, providing more detailed proofs to obtain more readable specifications together with a reasonable compilation time.

By experiencing the Focal environment within this new application area, we also identified some suitable evolutions that might enhance the expressive power and reasoning support of the underlying formalism. In particular, the notion of subtyping (an enhancement to the current notion of sub-species) seems a desirable feature to bring more flexibility to factorization and component reuse. Moreover, the integration of temporal mechanisms to the language might also allow the expression of complex

behavioral properties. Regarding Zenon, we needed to handle proofs by induction and second order properties, which are for the time being not yet supported and should be quite important improvements to be considered for this prover in the future; at the time of writing, induction is currently being integrated to Zenon.

2.2 CODE GENERATION FROM SPECIFICATIONS

In the same vein of providing appropriate means for structuring specifications, and in particular, in the idea of clearly separating (abstract) specifications from (concrete) implementations, we present a second contribution, which aims to generate implementations from specifications, more precisely purely functional code from inductive relations. This work was initially experimented in the Coq proof assistant [129], and is currently being adapted to the Focalize environment [133] (successor of Focal; see Appendix A). The very first root of this work can be found in the generation of ML programs from Typol specifications [63].

2.2.1 *Functional Extraction in Coq*

[*Contribution in collaboration with C. Dubois, J.-F. Étienne,*
and P.-N. Tollitte (Engineering/PhD student; see Subsec.C.1.1/C.2.3 of Appx. C).
CPR team, CEDRIC (CNAM), Paris (France), 2006-2010.
Published in [49, 50] (see Sec. D.2 of Appx. D).]

The idea of writing specifications and automatically extracting programs from these specifications is quite ancient. The Lex and Yacc tools are perfect examples of application and allow us to quickly and safely produce parsers from grammars. Always in the domain of programming languages, extensive work has been carried out to generate interprets, compilers and typecheckers from specifications of semantics [25, 108, 9, 63, 153]. In a more general setting, some other studies deal with the same problem. For example, some proof assistants like Coq [106, 98] or Isabelle [17] propose general extraction mechanisms to produce functional code from specifications and/or proofs. The same idea is used in [7], where OCaml programs [128] are extracted from CASL specifications [8], and also in [19], where ML programs are extracted from Isabelle specifications including inductive definitions of relations (this work has been recently extended [18], in order to produce an intermediary language under the form of an equational specification and independent from ML). Such a code generation from specifications is motivated not only by prototyping for purposes of specification animating or testing, but also by the possibility of offering an operational means of code generation, which is correct w.r.t. specifications.

With an intention close to the one found in the work previously mentioned, the aim of this contribution is to provide an extraction mechanism to translate logical inductive specifications into functional code. Like [19], this work goes beyond semantic applications [25, 108, 9, 63, 153], even though it is a typical domain of applications for such a work. This mechanism is presented in the framework of the Coq proof assistant [129], but is currently being adapted to the Focalize environment [133] (see Appendix A). Coq appears as an appropriate candidate for such a work as inductive types and inductive

relations in particular are the idiomatic way of building specifications. However, even if inductive relations contain (several) computable contents, they are not considered in the regular extraction mechanism of Coq [106, 98]. Therefore, this work is a natural extension of Coq's extraction. In addition, the extraction we propose is intended to only deal with inductive relations that can be turned into purely functional programs. This is mainly motivated by the fact that proof assistants providing an extraction mechanism generally produce code in a functional framework; this is the case of Coq in particular. Thus, we do not consider inductive definitions of relations that would require backtracking, that is more a Prolog-like paradigm. In that sense, this work separates from [25, 108, 19].

The extraction method we propose consists of two steps. The first step aims to analyze the inductive relation in order to know if it can be turned into a function that can compute some results. To do so, we need additional information. In particular, we need to know which arguments are inputs and which arguments are outputs. This information is provided by the user using the notion of modes. These modes are used to determine if a functional computation is possible, in which case we say that the mode is consistent. Otherwise, the mode is rejected and the functional extraction is not possible. The notion of mode, going back actually to attribute grammars [9], is fairly standard, especially in the logical programming community. For example, the logical and functional language Mercury [105] requires mode declarations to produce efficient code. Similar mode systems have already been described in [63, 19, 118]. The second step of the extraction we propose consists in producing the functional code. This code generation uses the mode supplied by the user and relies on a syntactical translation of the inductive clauses defining the initial relation. It should be noted that these two steps are based on decidable algorithms and the global procedure is therefore fully automatic.

The extraction mechanism described above has been formalized in the calculus of inductive construction (the underlying theory of Coq in particular), and its soundness has also been proved. This formalization is presented in [49], which can be found in Section D.2 of Appendix D. An implementation is available (on demand) under the form of a prototype developed in the Coq framework. In the following, we will illustrate the extraction method considered by means of examples, and we will also see an extension of this method to the Focalize environment.

2.2.2 *Mode Consistency Analysis*

The purpose of the mode consistency analysis is to check whether a functional execution is possible from the considered inductive relation. Basically, it is a very simple data-flow analysis performed on each inductive clause of this relation. To do so, a mode is required (and supplied by the user) for the inductive relation, and recursively for each inductive relation occurring in the inductive clauses defining this relation. Given an inductive relation, a mode is defined as a set of indices denoting argument positions of the inductive relation and which represent the inputs of this relation; the remaining positions are the outputs of the inductive relation. Although a mode is defined as a set, the order of the inputs in the logical inductive type is

relevant; they will appear in the functional translation in the same order. In practice, it is often the case to use the extraction with, either a mode corresponding to all the arguments except one, or a mode indicating that all the arguments are inputs. With no loss of generality, it is possible to only deal with these two kinds of modes (if more than one output is necessary, we can consider that the outputs are gathered in a tuple).

The mode consistency analysis consists in building a set of computed variables. Initially, this set contains the variables occurring in the inputs of the conclusion of the selected inductive clause. Next, each premise is inspected in its order of appearance in the inductive clause. For a given premise, we must verify that the variables occurring in its inputs are in the set of computable variables and if so, the variables occurring in its output (if there is an output) are added to the set of computed variables. Finally, once all the premises processed, we must verify that the variables occurring in the output (if there is an output) of the conclusion are in the set of computed variables and if so, the analysis is successful. If an analysis fails on a given order of the premises, a permutation of the premises may be operated to make the analysis successful (the new obtained order will also have to be used during the code generation step). The overall analysis is successful if there exists, for each inductive clause of the relation, a permutation of the premises for which the previous analysis is successful, otherwise it fails and the extraction is refused for this set of modes. The algorithm corresponding to this mode consistency analysis is formally described in [49] (see Section D.2 of Appendix D).

To illustrate this notion of mode consistency analysis, let us consider an example with the relation *add* that specifies the addition of two natural numbers, i.e. given three natural numbers n, m and p, (*add n m p*) defines that p is the result of the addition of n and m. Using the syntax of Coq, this relation may be defined as follows:

```
Inductive add : nat → nat → nat → Prop :=
  | add_O : forall n : nat, add n O n
  | add_S : forall n m p : nat, add n m p → add n (S m) (S p).
```

where *nat* is the type of natural numbers, defined as an inductive type with the two constructors *O* (zero) and *S* (successor).

For instance, let us check that the mode $\{1, 2\}$ (which corresponds to the computational content of addition) is consistent for the relation *add*. As said previously, this consistency is verified for each constructor:

- *add_O*: the initial set of computed variables consists of the input variables of the conclusion (*add n O n*), which is $S_0 = \{n\}$; as there is no premise, we just have to verify that the output variables of (*add n O n*) are in the set of computed variables, i.e. $\{n\} \subseteq S_0$, which is the case.

- *add_S*: the set of input variables of *add n* (*S m*) (*S p*) is $S_0 = \{n, m\}$; for the premise, we must verify that the input variables of (*add n m p*) are in the set of computed variables, i.e. $\{n, m\} \subseteq S_0$, which is the case; a new set of computed variables is obtained with the output variables of (*add n m p*), that is to say $S_1 = S_0 \cup \{p\} = \{n, m, p\}$; finally, the output variables of *add n* (*S m*) (*S p*) must be in the set of computed variables, i.e. $\{p\} \subseteq S_1$, which is verified.

In the same way, we can verify that the modes $\{2,3\}$ (which corresponds to the subtraction function) and $\{1,2,3\}$ (which represents the function verifying that the relation is satisfied) are also consistent for the same relation.

2.2.3 *Code Generation*

As the mode consistency analysis, the code generation is purely syntactic and is based on an analysis of the inductive definition of the considered relation. This analysis uses the set of modes supplied by the user (a mode for the relation and recursively for each inductive relation occurring in its definition), which must have been verified to be consistent for the relation, and generates code for each inductive clause defining the relation with the order of premises which has been used to establish the consistency of the provided set of modes. The code generation scheme distinguishes two cases according to whether the provided mode for the inductive definition considered selects no output or just one output. This code generation scheme has been formally described in [49] (see Section D.2 of Appendix D).

To understand this code generation scheme, let us see an example of extraction using the relation *add* introduced previously and with the mode $\{1,2\}$, for which we have already verified it is a consistent mode. If we use a syntax in the style of OCaml [128] for the generated code, the header of the extraction function has the following form:

```
let rec add12 (p1, p2) =
  match (p1, p2) with
  (* code generation for inductive clauses *)
```

where the function has two arguments as indicated by the provided mode.

From each inductive clause, a pattern-matching clause is produced. The corresponding pattern is built from the terms occurring at the input positions of the conclusion of the inductive clause (here, at positions 1 and 2). If the terms are not linear, a renaming is performed and a conditional expression representing the constraints between the several variables is introduced as a guard of the pattern-matching clause. In our case, the patterns are all linear, but this is not the case with the mode $\{1,2,3\}$ (with no output) for example. Each inductive clause is analyzed in the same order as in the definition of the inductive relation, and the extracted function becomes:

```
let rec add12 (p1, p2) =
  match (p1, p2) with
  | (n, O) → (* code generation for set of premises 1 *)
  | (n, S m) → (* code generation for set of premises 2 *)
```

The right-hand side parts of the pattern-matching clauses are made of the translations of the premises and the expected result (term occurring at the output position). The scheme of these translations is performed as follows:

```
match f1 (t11, ..., t1n1) with
| out1 →
  (match f2 (t21, ..., t2n2) with
   | out2 →
     (... → (* result of the inductive clause *)))
```

where f_k is the generated function for the predicate of the premise k, $t_{k1} \ldots t_{kn_k}$ the input terms of the premise k, and out_k the output term of the premise k.

The missing code fragment for the first inductive clause is immediate, as this clause has no premise. In this case, the result is just the output term of the conclusion of the inductive clause, here n. For the second inductive clause, we follow the scheme given above with one recursive premise. Finally, we obtain the following code:

```
let rec add12 (p1, p2) =
  match (p1, p2) with
  | (n, O) → n
  | (n, S m) →
    (match add12 (n, m) with
     | p → S p);
```

With a mode with no output (here with the mode $\{1,2,3\}$ for instance), the code generation is a little different, and the inductive relation is actually extracted as a boolean function. The interested reader is invited to refer to [49] (see Section D.2 of Appendix D), where the extraction with such modes is fully described with some examples in particular.

2.2.4 *Extension to Focalize*

The previous work has been formalized and developed in the context of the Coq proof assistant. A recent extension of this work [50] has consisted in integrating a similar mechanism into the Focalize environment [133] (successor of Focal; see Appendix A). Focalize is a quite appropriate environment for this experiment, as it relies on a language which provides a neat separation between specifications (species) and implementations (collections). This extension is actually more than a simple adaptation of the previous ideas developed in the framework of Coq, and brings some major evolutions in this domain of functional extraction from inductive relations. Here are some of these evolutions:

- The extraction mechanism produces functional code within the framework of the Focalize environment. Thus, contrary to the development realized in Coq, the specification language and the target language for extraction are the same, so that we obtain genuine Focalize code after extraction. As a consequence, this raises some new problems due to the constraints of consistency of the specification language. In particular, we have to deal with the termination of the extracted functions, and currently, we only deal with structural recursion.

- Correctness theorems are also generated during the extraction process. As expected, these theorems ensure that given some inputs, the result returned by the extracted function verifies (at least) one inductive clause of the considered relation. For a given extracted function, these theorems are split according to the different inductive clauses (there is one theorem per inductive clause), and are produced automatically (the fact of only considering structural recursion simplifies quite a lot the generation of the corresponding proofs).

- There is no notion of inductive relation in Focalize, i.e. a relation seen as the smallest set verifying given properties. As a consequence, the extraction is made w.r.t. a set of properties which is selected by the user and which is supposed closed to build the corresponding inductive relation to be extracted. Therefore, contrary to Coq where the relation is represented by a fixed and closed inductive type, the extraction in Focalize does not require a closed type and only considers the type as closed when an extraction is required. The situation is then much more flexible, since even after a given extraction, the user still has the opportunity to enrich the type considered for the extraction by adding some inductive clauses. In the same way, the user always has the possibility to ask for another extraction by omitting some inductive clauses.

To illustrate these evolutions, let us consider the example of the addition relation considered previously in Coq. This relation is introduced by the signature *add*, and uses the type *nat* defined by the two constructors *Zero* and *Succ*. The relation *add* is then specified by the two properties *addZ* and *addS* as follows (the code below uses the syntax of Focalize, the new version of the Focal compiler, which is slightly different from the syntax of the previous versions of the Focal compiler described in Appendix A):

```
type nat = | Zero | Succ (nat);;

species AddSpecif =
  signature add : nat → nat → nat → bool;
  property addZ : all n : nat, add (n, Zero, n);
  property addS : all n m p : nat, add (n, m, p) →
    add (n, Succ (m), Succ (p));
end;;
```

The extraction is required by the user using the new command "extract", which has been added to the Focalize language. In order to clearly separate specification from implementation (one of the main leitmotivs of Focalize), the user must define a new species which inherits from the appropriate specification and which will contain the extraction command(s). For instance, species *AddImpl* below requires the extraction of the addition of two natural numbers from the relation *add* defined by the properties *addZ* and *addS* in species *AddSpecif*:

```
species AddImpl =
  inherit AddSpecif;
  extract add12 = add (1, 2) from (addZ addS) (struct 2);
end;;
```

where *add12* is the name of the extracted function, (1, 2) the extraction mode, *(addZ addS)* the list of properties considered for the extraction, and "struct" is followed by the position of the argument which structurally decreases along the recursive calls.

The command "extract" invokes the extraction and can be seen as the definition of a new Focalize function (here function *add12*) if the extraction is successful. The generated code itself is transparent for the user (who can only see the extraction command and not the generated code), and is very similar to what we obtain in the

context of Coq. The only difference between the two code generations actually resides in the absence of guards in the pattern-matching of Focalize, which imposes to insert the non-linearity conditions as conditional expressions into the right-hand side parts of the pattern-matching clauses. This workaround is correct insofar as the several conclusions of the inductive clauses do not overlap, which is actually imposed by our extraction mechanism (see [50] for more details). Thus, in our example, the generated code for the function *add12* is the following:

```
let rec add12 (p1, p2) (struct p2) =
  match (p1, p2) with
  | (n, Zero) → n
  | (n, Succ (m)) →
    (match add12 (n, m) with
     | p → (Succ (p)));
```

As said previously, the extraction command also provides automatically the correctness theorems (the corresponding properties together with the proofs). These correctness proofs allow us to extend the use of the generated code to applications which require a high level of safety. Even if this code generation is essentially used with the intention of animating specifications and using the extracted code as an oracle to validate some tests, these correctness proofs allow us to ensure a certain level of confidence in the oracle. In the same way as for the code generation, the generation of the correctness theorems is transparent for the user (who can use them but cannot see them) and actually depends of the provided mode (with no or one output). In the case of one output, as in our example of the function *add12* extracted from the relation *add*, new properties are generated from the properties considered for the extraction (one property is generated per extraction property). For instance, the correctness theorems for the extraction function *add12* seen previously are generated as follows:

```
theorem add12_addZ : all n : nat, add (n, Zero, add12 (n, Zero))
proof = coq proof {* intro n; simpl; apply addZ. *};

theorem add12_addS : all n m p : nat, add (n, m, add12 (n, m)) →
  add (n, Succ (m), add12 (n, Succ (m)))
proof = coq proof {* intros n m p H; simpl; apply addS; auto. *};
```

where "coq proof" indicates that the provided proof is a Coq proof, which must be directly inserted (without any processing) into the Coq file produced by the Focalize compiler. The reason for using raw Coq proofs instead of proofs managed by Zenon, the automated theorem prover and reasoning support of Focalize, resides in the fact that Zenon still does not fully handle induction reasoning (some work in this domain is in progress by D. Doligez, and we should be able to provide genuine Focalize proofs of these theorems supported by Zenon in the short term).

These proofs follow a same scheme of proof, which consists in computing with the function *add12*, and then applying the corresponding property (among the properties selected in the extraction command). In the case of a mode with no output, the code and proof generation schemes are quite different; the interested reader can refer to [50] to get some examples with this kind of mode.

3

AUTOMATING

> "The general case of the Entscheidungsproblem of the engere Funktionenkalkül is unsolvable."
>
> *Alonzo Church.*
> *A Note on the Entscheidungsproblem.*
> *The Journal of Symbolic Logic.*
> *Vol. 1, No. 1, pp. 40-41, Mar. 1936.*

The purpose of this chapter is to tackle the problem of automation in theorem proving, and to propose several specific directions and in particular three groups of contributions by the author. The first contribution consists in providing appropriate means to increase automation in (interactive) proof assistants; it does not intend to increase automation itself but to improve the different tools which allow us to build suitable automatic proof procedures. This contribution resides in the design of a tactic language, called $\mathcal{L}_{tac}$, developed in the framework of the Coq proof assistant [129]. This new meta-language allows the user to write not only small and local automation routines, but also significant and complex proof procedures. This language has received a very favorable welcome from the Coq users since its introduction in the Coq distribution. Due to lack of space and as this contribution is more former than the two following ones, it is described in Section B.2 of Appendix B, in which we draw some perspectives regarding the future of meta-languages, such as $\mathcal{L}_{tac}$. The second contribution comes within the scope of the different possible interactions between deduction and computer algebra. The author is a very active member of the community working in this domain, and is currently a trustee of the Calculemus interest group [126] (the author was co-chair of Calculemus 2010 [11] in particular). This contribution can be actually divided into several sub-contributions. Among these sub-contributions, there is the development of a Maple mode for Coq, which allows us to import into Coq computations from Maple [138] over fields. Afterwards, this mode has been extended to deal with gcds over polynomials, in order to implement a quantifier elimination procedure over algebraically closed fields in the context of Coq. Still in the idea of benefiting from external computations as oracles, a procedure has been designed for the Focal environment [13, 132] to test the validity of properties over real closed fields using the computation of cylindrical algebraic decomposition performed by a routine of Axiom [125]. Finally, the third and last contribution is along the same lines of skeptical computations, and tries to apply this idea to automated deduction with two sub-contributions related to the Zenon automated theorem prover [24], which is

the reasoning support of Focal in particular. The first sub-contribution deals with the proofs produced by Zenon, and aims to translate them into Coq proofs for checking. The second sub-contribution intends to validate supplementary rules involved in applications developed using the B method [1] by means of Zenon proofs, which are translated back to B proofs.

3.1 DEDUCTION AND COMPUTER ALGEBRA

Automation in proof assistants necessarily requires a high power of computation, since proof assistants also deal with computation and not only with deduction. Some domains of computer science are completely focused on these aspects of computation. For example, this is the case of computer algebra, which aims to develop algorithms, programs and systems, that facilitate the use of symbolic mathematics on computers. In the following, we present three contributions which realize concrete interactions between proof assistants (PAs) and computer algebra systems (CASs). In particular, we describe two experiments involving Coq [129] and Maple [138], and a third one between Focal [13, 132] (see Appendix A) and Axiom [125].

3.1.1 *A Maple Mode for Coq*

⌈ *Contribution in collaboration with M. Mayero*
and with the assistance of T. Coquand.
Programming Logic Group, Chalmers University of Technology.
Göteborg (Sweden), 2001-2002. Published in [52] (see Sec. D.3 of Appx. D). ⌉

As has been rumored, computation generally appears as a weak point of PAs. This rumor, like every rumor, has a grain of truth in it, but is not entirely true either. Actually, if it is difficult to write efficient functions in a PA, it is mainly due to the constraints imposed by the environment of the PA, which must respect the underlying theory upon which it relies on. For example, termination, which is required for consistency purposes, is one of these constraints. Another source of potential inefficiency for functions in PAs resides in the choice of data structures. It is well know in the theory of complexity that data structures play a fundamental role in the complexity of an algorithm. However, we are not really free to choose any data structures in PAs, as we tend to favor data structures which will simplify the corresponding proofs of correctness. In addition, a data structure appropriate for efficiency is not necessarily suitable with regard to the feasibility of correctness proofs, and vice versa. For instance, for functions over natural numbers, Peano arithmetic is the right way to go when doing proofs over these functions, while we prefer to use binary integers when executing these functions. As a consequence, it is not rare, even these days, to get validated code, which appears unusable in practice for lack of reasonable efficiency; conversely, some functions are so efficient and tricky that tackling their validation is almost impossible.

One way to reconcile validation with efficiency is to sacrifice the global correctness of a function for a notion of local correctness. More precisely, this means that we do not prove the correctness of a function, but for each application of this function,

we prove that the result is correct. Thanks to local correctness, the function is not constrained anymore. In particular, it is possible to use any data structure, which is considered as appropriate for efficiency reasons. Furthermore, it is also possible to deal with non-terminating functions, and therefore to completely externalize the function. This is possible because the proof of local correctness only considers the result coming from the application of the function, and does not inspect the function itself, which is seen as a black box. However, as the correctness is only local, such functions cannot be used as usual in safety and security framework. Thus, such functions are intended to be used when we need efficient computations together with correctness guarantee of these computations. CASs are typical examples of systems which require such guarantee, as they provide powerful computations but without verifying side conditions, and they may then produce incorrect results. The present contribution proposes to make PAs and CASs interact in a fair way, i.e. in such a way that PAs can get efficient computations from CASs, and that CASs can benefit from guarantee over their computations in return.

Interactions between PAs and CASs

H. Barendregt and A. M. Cohen define in [14] several approaches to import CAS functionalities into PAs and conversely to bring more confidence to CAS computations. These different approaches actually depend on the degree of confidence between the PA and the CAS. For example, given a term t, if t' is a term resulting from a computation over t by a CAS, a first approach, called the "believing approach", consists in considering this computation as correct without any additional verification; concretely, this means we assume the equality $t = t'$ is an axiom. This solution is not satisfactory in terms of consistency, as we can imagine how easy it is to introduce false assumptions coming from incorrect computations. Another approach, called the "sceptical approach", resides in not trusting the previous equality, which must therefore be proved in the PA. To do so, there are actually two methods: either the result is verified independently of how the CAS obtained it, or the PA takes a trace of the rules that the CAS applied, and then uses that as a suggestion for what theorems should be used to construct a proof of the result. The latter method exactly follows the computation step by step and certifies each step; for instance, given a polynomial P, if P' is a factorized form of P, we will verify that each step of the factorization process respects the properties of the ring of which P is a member. As a consequence, this method may turn out to be as costly as the corresponding computation is. On the contrary, the former method is independent from the computation, and relies on whether the verification of the equality $t = t'$ is possible without reproducing the computation from t to t' or not; in the case of the example above, it is possible to verify that $P = P'$ without refactorizing P, simply by normalizing P and P', and comparing the two normal forms. The previous example is exactly the kind of problems we aim to deal with in the current contribution, which is described more precisely below. A last approach to combine PAs and CASs is the "autarkic approach", which consists in realizing the CAS computations within the PA. If this approach is appropriate in terms of confidence, it still keeps the same drawbacks mentioned previously and which are essentially due to the constraints imposed by the environment of the PA.

If the verification proposed in the skeptical approach allows us to increase the degree of confidence, it does not provide local correctness (introduced above) though. For instance, if we consider the example of the factorization of P, verifying that $P = P'$ does not ensure that the CAS computation actually realizes a factorization of P. This verification just guarantees that the CAS computation uses "legal" operations over polynomials. Thus, it is possible to go further with this skeptical approach by imposing that the result of the CAS computation verifies a given specification. This method can be seen as a hybrid method compared to the two methods already mentioned, as it does depend on the computation itself but may also require some additional information from the CAS, which can be seen as certificates. For example, given two polynomials P and Q, if we ask a CAS to compute the gcd G of P and Q, we may also ask the CAS to return the corresponding quotients P_1 and Q_1, and cofactors A and B, such that we can simply verify that G divides P and Q, i.e. $P = GP_1$ and $Q = GQ_1$, and that the Bézout relation holds, i.e. $AP + BQ = G$; see Subsection 3.1.2 for an example using this relation to verify computations involving polynomials over algebraically closed fields. It should also be noted that such a method relies on the feasibility of verifying a result against a given specification. In some cases, this verification may turn out to be very complex, and even as complex as the proof of global correctness. For instance, given a real closed field E and A a set of polynomials in r variables with coefficients over $\mathbb{Z}$, it seems quite difficult to verify that the result of a decomposition D of E^r is an A-invariant Cylindrical Algebraic Decomposition (CAD) [38] of E^r without performing the algorithm of CAD itself and proving its correctness; see Subsection 3.1.3 for an example involving CAD, and which, in the absence of local or global correctness proofs, proposes to use CAD to test the validity of propositions over real closed fields.

Computations from Maple to Coq

The present contribution [52] (see Section D.3 of Appendix D) concerns an experiment which has consisted in building a bridge between Coq [129] and Maple [138]. In this experiment, computations over fields are realized in Maple and then imported into Coq, which is asked to validate these computations. According to the different approaches seen previously, this interface between Coq and Maple is a skeptical approach in the weak sense of the term, i.e. we aim to validate computations but not against specifications. Therefore, it is not our intention to provide neither local nor global correctness of the exported Maple routines. Apart from the fact that Maple is both popular and easy to use, the choice of Maple for performing CAS computations is not motivated by specific features of Maple, as the exported functions (described below) are actually basic from the computer algebra point of view; in a way, some other CASs could also have been appropriate as much as Maple. However, the choice of Coq for validating the CAS computations actually relies on the fact that the corresponding validation can be done automatically. This validation is managed by the tactic "field" [51, 52] (see Section D.3 of Appendix D), which aims to solve equalities between algebraic expressions over fields. This tactic is entirely written in $\mathcal{L}_{tac}$ [44, 46, 45] (see Section B.2 of Appendix B), and in a total reflexive way [27, 78], which ensures its correctness in particular. As the problem is not

decidable in general, the tactic also generates some side conditions (typically that some expressions occurring in inverses must be non-zero), that the user must prove manually. Even though it was not the initial goal (which was actually to automate proofs over real numbers in Coq), this tactic has actually opened an opportunity of building bridges between Coq and CASs. It should also be noted that this tactic also offers a means to verify the result of a CAS computation independently of how the CAS obtained it, since the principle of this tactic consists in normalizing both sides of the equality and comparing the two normal forms (see [51, 52] and Section D.3 of Appendix D for more details), and not in reproducing the CAS computation.

The Maple functions dealing with algebraic expressions over fields which have been exported are the following: "simplify" which applies simplification rules to an expression, "factor" which factorizes a multivariate polynomial, "expand" which expands an expression, and "normal" which normalizes a rational expression. This set of functions is limited, but adding other functions (also with higher arities) is quite direct and very easy for the user. Let us illustrate the use of one of these functions in order to show how the bridge between Coq and Maple actually works. Given x and y two non-zero elements of an ordered field, we would like to prove the following inequation:

$$(\frac{x}{y} + \frac{y}{x})\ x.y - (x.x + y.y) + 1 > 0$$

To do so, we call the function "simplify" of Maple with the left-hand side member of the inequation as argument, and the application of this function returns 1 as result. As we have adopted a skeptical approach, an equation between the initial term and the result must be generated (and proved afterwards) as follows:

$$(\frac{x}{y} + \frac{y}{x})\ x.y - (x.x + y.y) + 1 = 1$$

To prove this equation, we call the tactic "field" of Coq, which succeeds and generates the side condition $x.y \neq 0$. This side condition is trivially true by hypotheses. Once this equation proved, we can replace the initial left-hand side member by 1 in the inequation, and we then obtain the new inequation $1 > 0$, which is trivially true. It should be noted that in this computation, everything is automatic except the proofs of the side conditions generated by the application of the tactic "field". This high automation nature tends to assert the computational aspect of this mechanism, as the user should not be surprised by too many deduction obligations coming from a given computation. For other (more complex) examples of computations, the reader can refer to [52] (see Section D.3 of Appendix D).

The implementation of this interface between Coq and Maple is available as a Coq contribution (see [129]). The implementation is quite light and the corresponding code is very short with about 300 lines of ML (we use a basic system of pipes between Coq and Maple). The contribution also contains some examples of use for each imported Maple function.

3.1.2 *Proofs over Algebraically Closed Fields*

Contribution in collaboration with M. Mayero
and with the assistance of T. Coquand.
Programming Logic Group, Chalmers University of Technology.
CPR team, CEDRIC (CNAM).
Göteborg (Sweden), 2001-2002; Paris (France), 2002-2005. Published in [53].

This contribution [53] can be seen as a quite direct sequel of the previous work, and consists in building an automated proof procedure which makes use of CAS computations. The idea is that importing computation into a PA is somewhat artificial if it is not used for increasing the automation of this PA. Thus, we propose the implementation of a proof procedure, which aims to solve systems of polynomial equations and inequations over algebraically closed fields. An Algebraically Closed Field (ACF) K is a field which has no proper algebraic extension, i.e. every algebraic extension is K itself. This means that every non-constant polynomial of $K[X]$ has a root in K. With respect to the usual properties of field, this adds the following condition:

$$\forall P \in K[X].\deg(P) > 0 \Rightarrow \exists x \in K.P(x) = 0 \tag{3.1}$$

where $\deg(P)$ denotes the degree of P.

The field of complex numbers, which is the algebraic closure (i.e. the algebraic extension which is algebraically closed) of the field of real numbers, is an example of ACF. The field of algebraic numbers, which is the algebraic closure of the field of rationals, is another example of ACF. It can be shown that equation (3.1) is equivalent to what is known as the Fundamental Theorem of Algebra (FTA), also called D'Alembert's theorem[1] when proved over the field of complex numbers, which states that, given an ACF K, every polynomial of $K[X]$ of degree $n > 0$ has exactly n roots (which may not be distinct). From a mathematical point of view, this theorem has the nice and direct consequence that polynomials over $K[X]$ can be factorized and it is possible to solve polynomial equation and inequation systems of the following form:

$$\begin{cases} P_1(X) = 0, \ldots, P_n(X) = 0 \\ Q_1(X) \neq 0, \ldots, Q_m(X) \neq 0 \end{cases} \tag{3.2}$$

To solve this kind of system, we only have to factorize all the polynomials and to choose a common root of all P_i which is not a root of any Q_j. However, from a computational point of view, this process of factorization is not fully automatic in general. It can only be done in some particular cases such as over the field of complex numbers $\mathbb{C}$ using, for instance, Kneser's constructive proof of equation (3.1) (see the FTA project [134]). Moreover, in practice, these methods turn out to be unsatisfactory: the former raises a problem of complexity (in general, the size of the minimal polynomial defining a splitting field of a polynomial is exponential in the degree), whereas the latter can produce arbitrary algebraic numbers as roots (typically, in [134] for $\mathbb{C}$, roots are pairs of limits of Cauchy sequences) and these are generally difficult to deal with.

1 Due to the first serious attempt at a proof of the FTA by D'Alembert in 1746, even if the first proof is usually credited to Gauss in his doctoral thesis of 1799.

In this contribution [53], we aim to solve systems such as (3.2), and consider an alternative method, called quantifier elimination (due to the implicit existential quantifier over the main variable X of the polynomials, we are trying to eliminate), which is mainly based on the idea of getting rid of the polynomial parts which do not contain the solution. The proposed method heavily relies on computations of polynomial gcds, which make it possible to simplify the system to be solved. Naively, we might think that polynomial gcd is a simple operation, which can be carried out in an autarkic way, but this would tend to underestimate the prolific nature of research in this domain. In fact, computer algebra provides many different algorithms of polynomial gcd with a full range of complexities, and some of them are implemented in CASs. Therefore, the principle of the considered implementation is to integrate this method into a PA, but also in externalizing the computations of gcds which must be performed by a CAS. De facto, the Maple mode for Coq [52] (see Subsection 3.1.1) appeared as a privileged candidate, and this method was then developed as a proof procedure for Coq [129], with the support of Maple [138] for gcd computations.

Quantifier Elimination over ACFs

In the following, given an ACF K, we consider polynomials over $K'[X]$ where $K' \subseteq K$ s.t. the equality to zero can be decided. In addition, there is no constraint over the characteristic of K, which may be non-zero. The algorithm we propose to solve systems such as (3.2) is precisely described in [53]. Thus, given P the gcd of P_i and Q the product of Q_i (instead we could also consider the lcm of Q_i), it mainly relies on whether P and Q are relatively prime or not: if they are, the system is reduced to $P = 0$, otherwise it looks for a solution for the system $P_1 = 0$ and $G \neq 0$, where G is the gcd of P and Q, and where P_1 is s.t. $P = GP_1$. There are also some particular cases depending on whether n or m are zero or not (i.e. some degenerate cases where there is no P_i or no Q_i); these cases are also handled and presented in [53]. Let us give an example of application of this method with the following system:

$$\begin{cases} 3X^3 + 10X^2 + 5X + 6 = 0 \\ 2X^2 + 5X - 3 \neq 0 \end{cases}$$

Given $P = 3X^3 + 10X^2 + 5X + 6$ and $Q = 2X^2 + 5X - 3$ the two polynomials involved in the system above, we first have to compute the gcd of P and Q, which is $G = X + 3$. Therefore, we are in the case where the two polynomials are not relatively prime, and given $P_1 = 3X^2 + X + 2$ s.t. $P = GP_1$, the previous system can be turned as follows:

$$\begin{cases} 3X^2 + X + 2 = 0 \\ X + 3 \neq 0 \end{cases}$$

Again, we have to compute the gcd of P_1 and G, which results in 1. As the two polynomials are relatively prime, the system is equivalent to $P_1 = 0$, which has a solution by the definition of ACF.

The reader can refer to [53] for additional examples of application of this procedure, as well as for the proof of decidability of this problem (a constructive proof which results in the extraction of the method described above).

Implementation in Coq using Maple

The previous method was developed in Coq [129] as an extension of the Maple mode [52] (see Subsection 3.1.1) and using polynomials with coefficients over $\mathbb{Q}$ (so that the previous constraints are respected, i.e. the equality to zero can be decided and the gcd remains a polynomial with coefficients over $\mathbb{Q}$). The externalized computation performed by Maple [138] is the gcd operation. As said previously, this may seem a little excessive to externalize this operation, since if K is a field, then $K[X]$ is a Euclidean domain, so that $\gcd(P, Q)$ for $P, Q \in K[X]$ can be easily computed by the (generalized) Euclidean algorithm and therefore in an autarkic way. However, it is well known that this algorithm suffers severely from coefficient growth in the intermediate steps, and it is clearly unacceptable in practice. More complex algorithms, such as Brown-Collins' subresultant algorithm [37, 28], are therefore far more appropriate. Thus, it appears wiser to let specialized tools such as CASs handle this kind of problems, and to simply call them when such computations are needed.

The externalization of the gcd computation respects the skeptical approach of the Maple mode for Coq, and is even stronger in the sense that the result provided by Maple is required to be actually the gcd and not only a divisor. This constraint is imposed by the method presented above, which relies on some theorems involving gcd. To ensure this constraint, the idea is to use the Bézout relation, i.e. given P, Q and G three non-zero polynomials, if G divides P and Q, and if there exist two polynomials A and B s.t. $AP + BQ = G$ then G is the gcd of P and Q. This relation has the advantage of allowing us to avoid to recompute the gcd within Coq when verifying the Maple computation. Nevertheless, in addition to the gcd, it requires further information, such as the two quotients P_1 and Q_1 s.t. $P = GP_1$ and $Q = GQ_1$, and the two cofactors A and B. The former information is used to verify that G is a divisor of P and Q, while the latter allows us to verify the Bézout relation. This additional information provided by the CAS can be seen as certificates which are intended to help the PA prove the correctness of the computation. However, some of them are little more than simple certificates, for example, the quotient P_1 is also used in the procedure itself (see the example provided previously). Concretely, the validation consists in verifying three polynomials equations, which is automatically done using polynomial operations and comparing polynomial coefficients by means of the tactic "field" [51, 52] (see Section D.3 of Appendix D). In this way, the call to Maple is quite transparent for the Coq user.

The reader can refer to [53] for some examples of use of this implementation in the context of the ACF $\mathbb{C}$ of complex numbers, and handling polynomials with coefficients over $\mathbb{Q}$.

3.1.3 *Tests over Real Closed Fields*

[*Contribution in collaboration with R. Rioboo*
and S. Toumi (Master student; see Subsec.C.2.4 of Appx. C).
CPR team, CEDRIC (CNAM), Paris (France), 2009.]

The aim of the present contribution was to extend the work over algebraically closed fields [53] (see Subsection 3.1.2) to real closed fields, while keeping a skeptical

approach. However, the several algorithms applied in the context of real closed fields are quite different from those used for algebraically closed fields, and contrary to what one might think, nothing of the previous experiments between Coq [129] and Maple [138] can be actually reused. Moreover, some additional difficulties regarding the results returned by the considered algorithms tend to thwart the opportunity of verifying the correctness of these results in an independent way, i.e. without proving the global correctness of these algorithms, which is a quite significant task. Nevertheless, these results can be exploited in another way, as an oracle for a test procedure, which can be used very early in the development life cycle during the analysis phase to determine the relevance of the set of system requirements, and also during the steps of prototyping when building a working model. This test procedure, which is described in detail in the following, was developed in the framework of the Focal environment [13, 132] (see Appendix A), and by means of an interface with the Axiom computer algebra system [125].

A Real Closed Field (RCF) E is a field for which there is a total order on E making it an ordered field such that, in this ordering, every positive element of E is a square in E and any polynomial of odd degree with coefficients in E has at least one root in E. Another equivalent definition of a RCF E is that E is elementarily equivalent to the field $\mathbb{R}$ of real numbers, i.e. it has the same first-order properties as the reals, which means that any sentence in the first-order language of fields is true in E if and only if it is true in $\mathbb{R}$. Some examples of RCFs are the field $\mathbb{R}$ of real numbers, the field of real algebraic numbers, and the field ${}^*\mathbb{R}$ of hyperreal numbers. It is possible to build RCFs from ordered fields by means of the Artin-Schreier theorem, which given an ordered field F states that F has an algebraic extension, called the real closure E of F, such that E is a RCF whose ordering is an extension of the given ordering on F, and is unique up to order isomorphism. For instance, the real closure of the field $\mathbb{Q}$ of rational numbers are the real algebraic numbers.

If the theory of real closed fields clearly lies within the domain of algebra, it was actually taken up with enthusiasm by logicians. By adding to the ordered field axioms, the axiom asserting that every positive number has a square root, and the axiom scheme asserting that all polynomials of odd order have at least one real root, we obtain a first-order theory. In 1948, A. Tarski [122] proved that this theory including the binary predicate symbols "=", "$\neq$", and "<", and the operations of addition and multiplication, admits elimination of quantifiers, which implies that it is a complete and decidable theory. This means that there exists an algorithm able to decide whether a logical formula, which is a combination of quantified polynomial equalities or inequalities with rational coefficients, is true or false. Thus, this algorithm can be applied to every problem which can be decomposed into a problem of first-order real arithmetic. In particular, this is the case of real algebraic geometry, which deals with algebraic varieties and which is therefore a decidable theory since it is also a model of the real field axioms. However, the initial algorithm proposed by Tarski has non-elementary complexity, which makes it unusable in practice except for very simple problems. In 1975, G. E. Collins proposes a drastic improvement with the algorithm of Cylindrical Algebraic Decomposition [38] (CAD), whose the complexity is "only" doubly exponential in the number of variables and polynomial in the degree of polynomials. Later work, see [15] for example, will achieve to improve

this complexity even better, but no implementation of these methods has been realized so far. As for CAD, there exist some implementations, but still very few due to the intrinsic difficulty of this algorithm for which it is quite complicated to provide a correct implementation. Thus far, there are Mathematica [139], where CAD has been developed by A. W. Strzeboński [119], QEPCAD [147] by H. Hong and others, which performs partial CAD and uses it for quantifier elimination, and the implementation of CAD in Axiom [125] by R. Rioboo. We decided to use the second implementation, as we actually just need CAD for our experiment (and not quantifier elimination), and as R. Rioboo, one of the participants of the present contribution, is also the author of this implementation, which was of great help when developing the interface between Focal and Axiom.

Cylindrical Algebraic Decomposition

Given a RCF E and A a finite set of polynomials in r variables and with integer (or rational) coefficients, the aim of the algorithm of CAD is to produce a partition of E^r, which is adapted to A, i.e. the sign of each polynomial of A is constant over each cell of this partition. As can be guessed, there may exist many of such partitions, and the partitions produced by the algorithm of CAD additionally verify some specific properties. More precisely, let us define what a CAD is. In the following, we use the definition of CAD described in [5, 6], which is a somewhat different (but equivalent) definition than that in [38].

We call a region, a nonempty connected subset of E^r. For a region R, the cylinder over R, written $Z(R)$, is $R \times E$. A section of $Z(R)$ is a set s of points $(x, f(x))$, where x ranges over R and f is a continuous real-valued function on R (in other words, s is the graph of f). We say such an s is the f-section of $Z(R)$. A sector of $Z(R)$ is a set $\hat{s}$ of all points (x, y), where x ranges over R and $f_1(x) < y < f_2(x)$ for continuous real-valued functions $f_1 < f_2$. The constant functions $f_1 = -\infty$ and $f_2 = +\infty$ are allowed. Such an $\hat{s}$ is the (f_1, f_2)-sector of $Z(R)$. Clearly, sections and sectors of cylinders are regions. Note that if $r = 0$, then $R = E^0$ is reduced to a point and $Z(R) = E^1$, such that any point of E^1 is a section of $Z(R)$ and any open interval in E^1 is a sector of $Z(R)$.

For any subset X of E^r, a decomposition of X is a collection of disjoint regions whose union is X. Continuous real-valued functions $f_1 < f_2 < \ldots < f_k$, $k \geq 0$, defined on R, naturally determine a decomposition of $Z(R)$ consisting of the following regions: (1) the (f_i, f_{i+1})-sectors of $Z(R)$ for $0 \leq i \leq k$, where $f_0 = -\infty$ and $f_{k+1} = +\infty$, and (2) the f_i-sections of $Z(R)$ for $1 \leq i \leq k$. We call such a decomposition a stack over R (determined by $f_1, \ldots, f_k$).

A decomposition D of E^r is cylindrical if either (1) $r = 1$ and D is a stack over E^0, or (2) $r > 1$, and there is a cylindrical decomposition D' of E^{r-1} such that for each region R of D', some subset of D is a stack over R. It is clear that D' is unique for D, and thus associated with any cylindrical decomposition D of E^r are unique induced cylindrical decompositions of E^i for $i = r-1, r-2, \ldots, 1$. Conversely, given a cylindrical decomposition $\hat{D}$ of E^i with $i < r$, a cylindrical decomposition D of E^r is an extension of $\hat{D}$ if D induces $\hat{D}$.

A subset of E^r is semi-algebraic if it can be constructed by finitely many applications of of the union, intersection, and complementation operations, starting from sets of

the form $\{x \in E^r \mid F(x) \geq 0\}$, where F is an element of $\mathbb{Z}[x_1, \ldots, x_r]$, the ring of integral polynomials in r variables. We write I_r to denote $\mathbb{Z}[x_1, \ldots, x_r]$. There exists an equivalent definition using the notions of definable set and defining formula (by formula, we mean a well-formed formula of the first order theory of RCFs). A definable set in E^k is a set S such that for some formula $\Psi(x_1, \ldots, x_k)$, S is the set of points in E^k satisfying Ψ. Ψ is a defining function for S. A definable set is semi-algebraic if it has a defining formula which is quantifier-free. It is well-know that there exists a quantifier elimination method for RCFs [122]. Hence a subset of E^r is semi-algebraic if and only if it is definable.

A decomposition is algebraic if each of its regions is a semi-algebraic set. A Cylindrical Algebraic Decomposition (CAD) of E^r is a decomposition which is both cylindrical and algebraic.

Let X be a subset of E^r, and let F be an element of I_r. F is invariant on X (and X is an F-invariant), if F has a constant sign on X (positive, zero, or negative). Let $A = \{A_1, \ldots, A_n\}$ a subset of I_r. X is A-invariant if each A_i is invariant on X.

Given a set $A \subset I_r$, the algorithm of CAD by Collins [38] provides an A-invariant CAD of E^r, i.e. a CAD of E^r for which each region is A-invariant. Note that a set $A \subset I_r$ does not uniquely determine an A-invariant CAD D for E^r. Since any subset of an A-invariant region is also A-invariant , we can subdivide one or more regions of D to obtain another "finer" A-invariant CAD. We should also note that all the previous remains valid for $I_r = \mathbb{Q}[x_1, \ldots, x_r]$.

Let us illustrate the notion of A-invariant CAD with an example. Given $A \subset I_2$ the set $\{A_1(x, y)\} = \{xy\}$, the following collection of regions forms an A-invariant CAD of E^2:

$$\begin{array}{ll} \{(x,y) \mid x < 0 \text{ and } y > 0\} & \{(x,y) \mid x > 0 \text{ and } y > 0\} \\ \{(x,y) \mid x < 0 \text{ and } y = 0\} & \{(x,y) \mid x > 0 \text{ and } y = 0\} \\ \{(x,y) \mid x < 0 \text{ and } y < 0\} & \{(x,y) \mid x > 0 \text{ and } y < 0\} \\ \{(x,y) \mid x = 0\} & \end{array} \tag{3.3}$$

We will not describe the CAD algorithm of Collins here, as we are just interested in using the result of the algorithm in this contribution (we will see how below). But, basically, the general algorithm consists of three phases: projection (computing successive sets of polynomials in $I_{r-1}, I_{r-2}, \ldots, I_1$; the zeros of each set contain a "silhouette" of the "significant points" of the zeros in the next higher dimensional space), base (constructing a decomposition of E^1), and extension (successive extension of the decomposition of E^1 to a decomposition of E^2, E^2 to E^3,..., E^{r-1} to E^r). The reader can refer to [38, 5, 6] for more details.

A computation of CAD can be used for several purposes, such as quantifier elimination (see the QEPCAD tool [147], for instance), i.e. in order to transform a first-order formula into an equivalent quantifier-free formula, or verification of the validity of first-order formulas. In this contribution, we focus on the latter application of CAD, which can be easily performed using an arborescent representation of the CAD. We will detail this example of use in the following. It is important to note that the correctness of this example of application relies on the fact that the computed decomposition is actually an A-invariant CAD, where A is the set of input polynomials built from the set of first-order formulas to be verified. The question is then the following: is it

possible to verify that a given decomposition D is an A-invariant CAD? This problem is undecidable in general (verifying that a collection of sets is a decomposition of E^r is still undecidable), and this tends to ruin any hope of externalizing the CAD computation using a skeptical approach (as can be done in the contributions described previously in Subsections 3.1.1 and 3.1.2). However, even if we cannot ensure that a given decomposition is a CAD, we can use it for testing purposes, as a CAD appears to be a very precise oracle when testing first-order formulas. In particular, this allows us to increase the degree of confidence in the relevance of a set of system requirements built during an analysis phase for example. Thus, in this contribution, we propose a test procedure for first-order formulas over RCFs and which is based on the CAD algorithm of Collins [38].

A Test Procedure using CAD

The test procedure introduced above for verifying first-order formulas over RCFs was implemented as an interface between the Focal environment [13, 132] (see Appendix A), and the Axiom computer algebra system [125]. The choice of Focal is essentially motivated by the significant library of computer algebra developed by R. Rioboo, and which provides a large part of the material (such as recursive and distributed polynomials, a preliminary version of real closure, etc) necessary to the elaboration of this test procedure. As for Axiom, the choice is driven by the existing implementation of Collins' CAD algorithm [38] by R. Rioboo, who is also one of the participants of this contribution. As said previously, there exist very few implementations of this algorithm (there is also QEPCAD [147] for example), and this version has the advantage of only focusing on producing a CAD (and does not perform quantifier elimination for instance).

The CAD implementation in Axiom can return several kinds of results, some of which are of interest for our test procedure. One of them is the list of regions composing the CAD and which is also supposed to be A-invariant, where A is the set of initial polynomials given as inputs. Contrary to the CAD described in the example (3.3), a region is not described as a subset of E^r, but the CAD routine only returns a sample point of this region. If we suppose that the returned result is correct, this is not problematic when verifying formulas as the polynomials of A are supposed to have the same sign all over the region and it is therefore sufficient to verify the formulas for one sample point of this region. It should be noted that the components of these sample points are algebraic numbers, and they may be either rational or not. In the latter case, we cannot get a value of them (only an approximation), and specific operations are needed to handle them. Again, this is actually not problematic as we ask Axiom to perform these operations when required (typically when we need to compute the sign of a polynomial for a given sample point). The presence of algebraic numbers in the sample points is not surprising and occurs in the base phase of the CAD algorithm in particular. In this phase, the decomposition of E^1 consists of the roots of the input polynomials, one sample point between two successive roots, as well as a lower bound and an upper bound of this set of points. If it seems possible to choose rational points for the sample points and the lower and upper bounds, it is not always the case for the roots of input polynomials (for instance, the golden ratio ϕ is

the root of $x^2 - x - 1$ and is not rational). Another result returned by the CAD routine of Axiom is the list of signs of input polynomials for each region, i.e. computed for each sample point (thanks to this result, it is not necessary to ask Axiom to compute the input polynomials applied to the several sample points).

The two lists returned by the CAD routine allow us to build a treelike structure which is very appropriate for verifying first-order formulas. The structure of tree consists in decomposing each component of the sample points, which are embedded in the nodes, and in storing the corresponding signs in the leafs. Thus, a tree is either a leaf containing a list of signs for the input polynomials, or a node containing a component i and a list s of trees, such that for each component j of each tree of s, there exists a sample point with components i and j. The tree is built in such a way that a node with a given component is unique. The order of the component decomposition is the same as the order of the quantifiers occurring in the formula to be verified. Thus, a tree is always associated with a formula, whose it is intended to help verify the validity. From such a tree, verifying a quantified formula is actually straightforward: a universal quantifier over a component implies to browse all the branches leading to the level of this component and to verify the formula for each branch (once the quantifiers have been eliminated), whereas an existential quantifier implies to find one branch verifying the formula.

Let us consider the previous example of the A-invariant CAD of E^2, where $A \subset I_2$ is the set $\{A_1(x,y)\} = \{xy\}$. In this example, the CAD routine of Axiom returns the two following lists (respectively the list of sample points and the list of signs):

$$\{(-1,-1),(0,-1),(1,-1),(0,0),(-1,1),(0,1),(1,1)\}$$
$$\{1,0,-1,0,-1,0,1\}$$

where 1, 0 and -1 means respectively positive, zero and negative in the list of signs.

Let us now verify the formula $\exists x.\forall y.A_1(x,y) = 0$, for instance. From this formula (from the order of the quantified variables actually), it is possible to build a tree according to the rules introduced above as shown in Figure 1. From this tree, we can remark that the second branch leading to the node where $x = 0$ seems to work as every branch starting from this node leads to a leaf where $A_1(x,y) = 0$ whatever the value of y. This formula is therefore valid.

The implementation of this test procedure consists of about 300 lines of ML code. As perspectives, we consider to use a more standard format to exchange data between Focal and Axiom. For example, OpenMath [145] could be a suitable solution, as some OpenMath functionalities are already available both in Focal and Axiom (even if not actually available in standard versions). In addition, another perspective should consist in increasing the development of the theory of RCFs in Focal. In particular, the preliminary implementation of real closure should be completed in order to handle effective RCFs.

3.2 CERTIFICATION OF AUTOMATED PROOFS

Automated Theorem Provers (ATPs) and more generally automated tools aim to provide proofs or results in a fully automatic way when possible. However, the question is: what degree of confidence are we ready to give to these proofs or results?

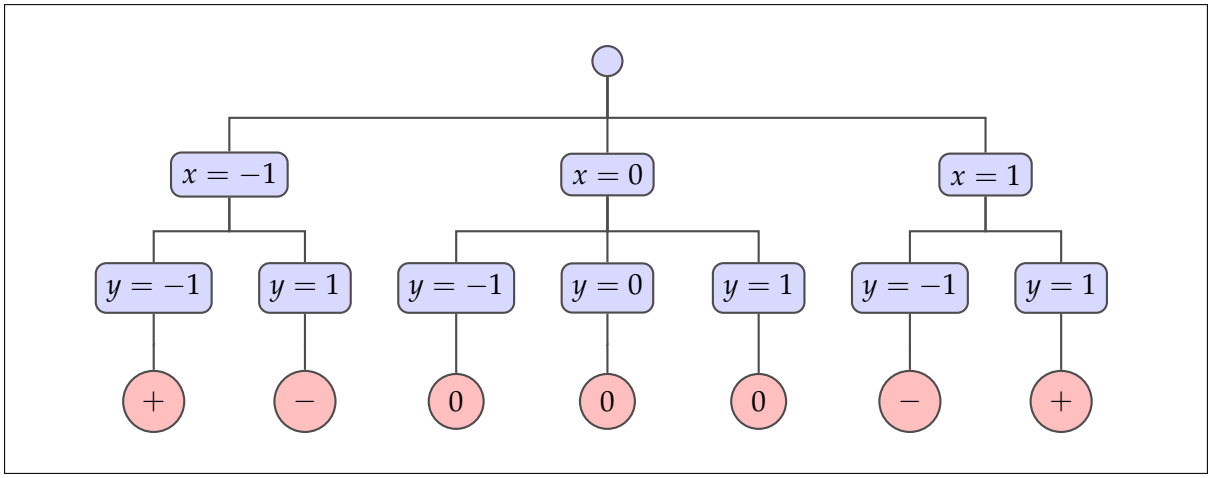

Figure 1.: A-Invariant CAD of E^2, where $A = \{xy\}$ and with list of signs

This question is actually even more critical when we know that most of the time these systems or tools only produce results and at best some traces intended to justify their deductions or computations. In a safety or security context, the use of such tools is clearly unsatisfactory, as the results are provided without any guarantee. The two following contributions intend to mitigate this lack of confidence by proposing, in addition to the results provided by a given tool, to build objects justifying these results and that can be verified independently from the tool. The first contribution deals with the proofs produced by the Zenon automated theorem prover, and which are translated into Coq proofs [129] for checking. The second contribution consists in validating supplementary rules involved in applications developed using the B method [1] by means of Zenon proofs, which are translated back to B proofs.

3.2.1 *Validation of Zenon Proofs*

[*Contribution with the assistance of D. Doligez.*
CPR team, CEDRIC (CNAM), Paris (France), 2003-2004.
Published in [24] (see Sec. D.4 of Appx. D).]

As could be expected, ATPs should be suitable companions for formal systems, such as interactive theorem provers for example, whose aim is to achieve checkable proofs of theorems. However, most of these systems are more concerned by proving more theorems in a finite and especially reasonable time than building the formal proofs of theorems for which they guess a solution. Fortunately, most of them also produce traces, which can be potentially exploited to build the corresponding formal proofs. This is more or less feasible, as these traces are in general not much documented. Over the past ten years, some experiments were conducted to increase the integration of ATPs into systems specialized in formal and mechanized proofs, such as between Gandalf and HOL [82], between Otter and ACL2 [103], between Bliksem and Coq [22], or between E, SPASS, Vampire and Isabelle [107]. In this contribution, we propose a similar approach with the Zenon first-order ATP [24] (see Section D.4 of Appendix D), whose proofs can be translated into Coq proofs [129]. Compared to the experiments mentioned above, this approach is actually not as similar as could be expected, since

the generated Coq proofs are not intended to be used in Coq but are only produced to be checked by Coq, which is seen as a proof checker. This can be explained by the fact that Zenon is the reasoning support mechanism of the Focal environment [13, 132] (see Appendix A), which is able to produce Coq specifications for certification in particular. In the compilation scheme of Focal, Zenon is involved between the specification level and the generated Coq implementation, where it helps the user find his/her proofs and produces Coq proofs which are reinserted in the Coq specifications generated by the compiler and fully verified by Coq. This final verification by Coq is a guarantee of soundness both for Zenon and the Focal compiler.

The Zenon ATP

The Zenon ATP, developed by D. Doligez, deals with first order classical logic (with equality), and is based on the tableau method. Even though, these days, the tableau 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, called LLproof (see [24] and Section D.4 of Appendix D), which is very close to a sequent calculus. From LLproof proofs, Zenon can directly produce proofs for Coq (it could be easily done for other proof assistants). The search method of Zenon, called MLproof, relies on inference rules described in [24] (see Section D.4 of Appendix D) and which are applied with the usual 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. ending with the application of a closure rule), the tree is closed and we have a proof of the goal.

The search method of Zenon has some specific features. One of them is to handle the δ rules with Hilbert's ϵ-operator [81, 97] rather than using Skolemization. While most of the ATPs use Skolemization rather than ϵ-terms (typically because Skolem terms are handled like usual terms of the object language, while ϵ-terms are new terms which extend the object language), this approach provides some benefits though. For example, ϵ-terms lead to exponentially speed-up (and probably non-elementary speed-up) w.r.t. some sophisticated Skolemization rules [71], such as δ^{++} [16] for instance. This is explained by the fact that ϵ-terms are strictly related to the formulas which introduce them, while Skolem terms are generally unrelated to the formulas that generate them. In this way, ϵ-terms allow us to automatically discover shorter proofs which might rely on relationships between Skolem terms. Shorter proofs are a decisive argument for adopting such an approach when eliminating existential quantifiers, since Zenon proofs are intended to be verified by Coq and this verification must remain feasible in terms of size of proofs.

Another specificity of Zenon is its non-destructive framework. Typically, metavariables (often named free variables in tableau-related literature), which are introduced when dealing with universally quantified formulas (or negations of existentially quantified formulas), are never substituted. They are used to trigger potential contradictions, and when a contradiction has been identified, the corresponding instantiation rules are used but the metavariables remain available (providing as many instantiations as needed). Thus, a closed MLproof tree may still contain metavariables, but they can be removed by pruning (see below).

A last specific feature of Zenon is the opportunity of minimizing the size of the search tree by pruning (we can actually also minimize this size by providing a suitable order over the rules to be applied, but this is quite usual in tableau-like methods). The pruning works as follows. 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. A formula is said to be 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. As said previously, pruning can be performed to remove metavariables in particular (when all the instantiations have been found in a given subtree). It should be noted that pruning is applied during the proof search (and not only when the proof tree is closed), and may reduce the branching factor of the search tree, which results in shorter proofs and a significant speed-up.

Generation of Coq Proofs

The generation of Coq proofs from Zenon proofs is carried out from the LLproof format mentioned previously. From a theoretical point of view, this translation ensures the soundness of the LLproof formalism (w.r.t. a known theory), while from a practical point of view, this provides a (local) guarantee of correctness regarding Zenon's implementation. But especially, in the context of the Focal system [13, 132] (see Appendix A), 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.

This translation into Coq proofs 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 also considers equality as a special predicate and uses specific rules to deal with it. Thus, to translate proofs with equality correctly, we have to extend the theory of LJ with equational logic rules. We called this theory $\mathsf{LJ}_{\mathsf{eq}}$ (see the rules in [24] and Section D.4 of Appendix D). Thus, we can prove that every sequent provable in LLproof has a proof in $\mathsf{LJ}_{\mathsf{eq}}$.

As for the implementation, in order to factorize proofs and especially to minimize the size of the produced proofs, the idea is to prove a lemma for each translated rule.

Thus, a generated Coq proof is simply a sequence of applications of these lemmas. The proofs are not only quite compact, but also quite efficient to be checked. For instance, for the $\neg\wedge$ rule of LLproof (see the rules in [24] and Section D.4 of Appendix D), 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 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 partially apply the corresponding lemma *zenon_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.

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. 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 it 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 is able to keep this type information (this is possible since Zenon works in a non-destructive way, see above); in this case, we generally have a multi-sorted first order logic. Another difficulty is that mono/multi-sorted first order logic implicitly supposes that each sort is nonempty, while types may be not inhabited in Coq. This problem is fixed by systematically skolemizing the theory and considering at least one element for each sort, e.g. E for U.

3.2.2 *Validation of B Proofs from Zenon*

[*Contribution in collaboration with C. Dubois and M. Jacquel (PhD student; see Subsec.C.1.3 of Appx. C). CPR team, CEDRIC (CNAM), Paris (France), 2010.*]

This contribution is quite recent and work in progress. This is the fruit of a collaboration with Siemens Transportation Systems (STS), formerly Matra Transport International (MTI), which is the world leader in the domain of automated urban transport. Due to the critical nature of the systems developed by STS, their production line heavily relies on formal methods. In particular, they have been making an intensive use of the B method [1] for more than 15 years, and have been participating (together with RATP, Alstom, SNCF, and INRETS) to the development of the Atelier B (by Steria, and later by ClearSy), which is the corresponding industrial tool. The use of such method is spectacular when in November 1997, the Meteor metro line (now line 14), deployed by MTI, is launched in Paris, as this metro line is the first automated line, which also accepts "manual" trains, but especially because 80% of the safety software was developed with the B method. Since then, one of the main goals of the successive research and development teams at MTI and later at STS was to increase the degree of automation in the use of the Atelier B in particular.

The B method [1] is a collection of mathematically based techniques for the specification, design and implementation of software components. Systems are modeled as a collection of interdependent abstract machines. An abstract machine comprises a state together with operations on that state. In a specification and a design of an abstract machine, the state is modeled using an ad-hoc first-order set-theoretical language (sets, relations, functions, sequences, etc). The operations are modelled using pre and post-conditions (expressed using generalized substitutions). In an implementation of an abstract machine, the state is again modeled using a set-theoretical model, but this time we already have an implementation for the model. The operations are described using a pseudo-programming notation that is a subset of the set-theoretical language introduced above. The B method also prescribes how to check the specification for consistency (preservation of invariant properties), and how to check designs and implementations for correctness (correctness of data refinement and correctness of algorithmic refinement). These checks produce proof obligations, and the Atelier B provides in particular a tool, called PP, which is an ATP intended to help the user discharge a maximum of his/her proof obligations.

The PP ATP cannot deal with all the proof obligations, and the idea developed by STS is to add these unproven proof obligations as derived rules (which can be therefore used afterwards) and to validate them by other means. This validation is performed thanks to an external prover, namely Coq [129]. More precisely, a deep embedding of the B theory, called BCoq [20], has been realized, and each rule to be validated is translated in this reified version of B. The proof of a derived rule is then carried out using Coq and especially the reified type corresponding to B proofs. Once the proof completed, the derived rule is considered as validated. The advantage of a deep embedding is that the logic of Coq does not "contaminate" the B proofs, which are genuine proofs of the B theory. However, a drawback is that we cannot benefit from the automation of Coq, and each automation must be written from scratch. The present contribution aims to mitigate this drawback by plugging some external ATPs in order to deal with these proofs. Thus, some experiments have been conducted with Zenon [24] (see Section D.4 of Appendix D), E [130], and SPASS [150]. Nevertheless, directly tackling reified proofs with these ATPs quickly appeared unsatisfactory, as the encoding of B theory clearly adds an additional complexity in the proof search.

Thus, an alternative approach is to unreify the rules to be validated and to launch the ATPs on them. Once a proof is found for a given rule, the proof is then rereified, so that we obtain a genuine reified B proof. This experiment is currently in progress using Zenon, which appeared as the most appropriate ATP for this work. In fact, Zenon is able to prove almost all the rules we want to deal with, and what is more, the Coq output of Zenon (see Subsection 3.2.1) is quite suitable when rereifying the proofs of the derived rules.

4

COMMUNICATING

> "Programs must be written for people to read, and only incidentally for machines to execute."
>
> *Harold Abelson and Gerald J. Sussman.*
> *Structure and Interpretation of Computer Programs.*
> *The MIT Press, July 1996.*

This chapter aims to deal with the notion of communication between proof assistants and their users. The word "users" must be considered in a broad sense here, as it concerns not only developers writing specifications in proof assistants, but also people who analyze and evaluate developments, such as people in charge to assert that a given development is in accordance with a given regulation for example. This notion of communication takes on several aspects related to the way of writing specifications, the way of producing documentation, and also the way of compiling specifications. In particular, we have to make sure that some features can be guaranteed, such as readability, maintainability, documentation, and traceability for instance, which all of them are important components of dependability in system and software engineering. In the following, these several aspects and features of communication between proof assistants and users are highlighted through three contributions by the author. The first one is upstream of the proof assistant, as it consists of a new proof language, developed in the context of the Coq proof assistant [129]. This proof language is intended to be independent of a given proof style, and therefore allows the user to develop proofs in procedural, declarative and proof-term based styles. Due to lack of space and as this contribution is actually more former than the two following ones, it is described in Section B.3 of Appendix B, in which we present some perspectives regarding the next proof languages, such as $\mathcal{L}_{pdt}$. The second contribution is completely downstream of the proof assistant, and proposes an automatic transformation of Focal specifications [13, 132] (see Appendix A) to UML class diagrams [144] for documentation purposes. This work lies within the framework of the EDEMOI project [131] introduced in Section 2.1 of Chapter 2. Finally, the third contribution is somewhere between the two other contributions, as it aims to elaborate a compilation scheme for Focal based on modules, which is supposed to be an alternative to the current scheme using records. This new compilation model has the advantage of providing a higher level view of compiled specifications supplying in particular traceability.

4.1 FROM FOCAL SPECIFICATIONS TO UML MODELS

Contribution in collaboration with V. Viguié Donzeau-Gouge and J.-F. Étienne (PhD student; see Subsec.C.1.1 of Appx. C). CPR team, CEDRIC (CNAM), Paris (France), 2004-2008. Published in [57, 59] (see Sec. D.5 of Appx. D).

This contribution lies within the framework of the EDEMOI project [131], introduced in Section 2.1 of Chapter 2 and which aims to integrate and apply several requirements engineering and formal methods techniques to analyze regulations in the domain of airport security. In the context of this project, several formalizations of different airport security regulations were developed in particular using the Focal environment [13, 132] (see Section 2.1 of Chapter 2). The objective of the present contribution is to also provide documentation for these formalizations by means of an automatic tool implemented as an extension of Focal and able to produce UML models [144] from Focal specifications.

4.1.1 *The Need for Documentation*

Even though formal methods offer a systematic approach for verification, the validation process still relies on a high degree of interaction between the various stake-holders (developers, customers, end-users, certification authorities, etc) involved in a critical project. In addition, the use of formal methods requires a certain level of expertise in mathematics, which usually hinders communication. In fact, the mathematical notations used are often too obscure for inexperienced users to properly understand the exact meaning. As a result, the validation of requirements is difficultly achievable. This may even jeopardize the entire project as misinterpretations or specification errors may lead to the validation of a totally wrong implementation.

A widely adopted solution to these problems is the integration of formal and graphical specification. In general, the use of graphical notations is quite useful when interacting with end-users. In fact, these tend to be more intuitive and are easier to grasp than their formal (or textual) counterparts. During the last few years, UML [144] has emerged as a standard in industry for modeling software systems. It provides a set of graphical constructs, which enables the modeling of systems in an object-oriented style. Currently, it is supported by a wide variety of tools, ranging from analysis, testing, simulation to code generation and transformation. Interoperability between these tools is generally achieved by exporting the UML models using the XMI interchange format [143].

There have been several researches devoted to establishing the link between UML and formal methods. One of the approaches that has been largely studied in this domain is the translation of UML diagrams into formal specifications [89, 91, 92, 64], which attempts to benefit from the formal methods tools and techniques while still having control over the UML-based industrial practice. The converse approach [84, 83] is here considered to generate UML diagrams as graphical documentations for Focal specifications [13, 132] (see Appendix A). There are also some work [68] that stress on the seamless integration of formal notation and UML, whereby the purpose is a

development environment that facilitates the translation in both directions. We do not consider such issue for the time being, but believe that it plays an important role in the validation process: graphical models produced from the analysis phase and validated by certification authorities; formal models generated from the graphical ones and completed for verification; anomalies detected in formal models propagated back to graphical models for discussion with certification authorities.

As said previously, the main motivation for this work lies within the framework of the EDEMOI project [131], and in the context of this project, we used Focal to realize the formal models of two regulations, namely the international standard Annex 17 and the European directive Doc 2320 [54, 56] (see Section 2.1 of Chapter 2). Within this project, the purpose of the UML diagrams is two-fold. First, to provide a graphical documentation of the formal models produced for developers, thus establishing a common understanding of what is being formalized and analyzed. Second, to generate higher-level views of the formal models that would be more appealing to certification authorities.

For our concern, the choice of UML as a graphical notation mainly resides in the fact that most of the Focal design features can seamlessly be represented in UML. One could argue that the creation of a domain specific language exclusively for Focal would be a better approach, as it avoids us from having to deal with the intricacies of the UML semantics. The use of text-to-model tools [72], such as xText or TCS, generally facilitates such process, whereby the grammar of the target language is taken as input and the corresponding metamodel, parser and editor is generated as output. However, regardless of these facilities, we still have to develop a graphical concrete syntax for each concept. Moreover, the corresponding semantics might be intuitive to developers but not necessarily to end-users or certification authorities (which is our long-term objective). For example, from the Focal compiler we do have the possibility to generate inheritance and dependency graphs. Nevertheless, these graphs are generally intended for developers and mainly convey information extracted from the type inference and dependencies analysis performed by the compiler. Finally, the choice for UML also allows us to have access to a wide variety of tools ranging from analysis to code generation and transformation. For instance, the UML models produced can serve to map Focal specifications to other object-oriented languages, i.e. Java or C#.

4.1.2 *Profile and Transformation Rules*

The transformation of Focal specifications to UML models consists of three parts: a formal description for a subset of the UML 2.1 static structure constructs; a dedicated profile extending the UML metamodel in order to cater for the semantic specificities of the Focal language; a set of transformation rules based on the UML profile obtained. It should be noted that this transformation is fully formalized, so that it is possible to prove and ensure its correctness.

The UML syntax we consider is a subset of the UML 2.1 static structure constructs [144] used as a means to provide a graphical documentation for Focal specifications. We mainly focus on the basic constructs necessary to represent the notion of species and collection. Normally, the syntax and semantics of each UML modeling

construct are described in the form of a metamodel. The syntax is specified using class diagrams, while the semantics are well-formedness rules expressed in a combination of OCL [142] and English. In order to devise a formal framework for our transformation, we proposed in [59] an abstract syntax for a subset of the UML 2.1 static structure constructs. The syntax was mainly derived from the UML 2.1/XMI schema [143] to reflect as much as possible our implementation. In [57] (see Section D.5 of Appendix D), we present a new syntax that hides some of the complexities inherent to the UML metamodel and thus less dependent on the XMI format. This not only allows us to increase the readability of our transformation rules but also to provide an appropriate means to facilitate reasoning.

An alternative approach can consist in making use of a text-to-model tool [72], e.g. xText or TCS, to obtain a metamodel of the Focal specification language instead of defining an abstract syntax for UML. The automatic transformation from Focal to UML may then be realized at a metamodel level through the use of a model-to-model transformation language [85], such as ATL or QVT. Nevertheless, even though such an approach can be considered during the implementation phase, it does not allow us to prove the soundness of our transformation.

In order to properly specify, visualize and document Focal models using UML notations, there is also a need to extend the default UML metamodel as defined in [144] to cater for the semantic specificities of the Focal specification language. The necessary extensions are realized via the creation of a profile, where appropriate stereotypes are defined to encode the semantics of each Focal construct in the form of appropriate OCL constraints [142]. These stereotypes are namely «Species», «Collection», «FocalType», «Method», «In», «Is», «ParameterizedInheritance», «Inheritance» and «Implements». To validate our transformation, we also provided a complete formalization of the semantics relative to the template binding construct via the introduction of intermediate stereotypes declared as required (i.e. mandatory when the corresponding profile is applied). For this purpose, we based ourselves on the OCL formalization realized by [30], which we extended to handle nested bound classes and inherited members. Thus, the syntax introduced previously is slightly extended to reflect these new stereotypes (see [57] and Section D.5 of Appendix D).

From the dedicated UML profile introduced above, we can already have an insight of how the UML class constructs can be used to model a Focal specification. However, despite their similarities, Focal species and UML classes are based on two different concepts. In Focal, the functions defined in a species are intended to manipulate entities of a given representation, which are static items having a unique value. Hence, we model a species as an abstract factory class (stereotyped with «Species»), which defines an interface for manipulating immutable value objects of a given type. More precisely, a part of the transformation rules is described in [57] (see Section D.5 of Appendix D), while all the rules are presented in [66]. In the latter presentation, in the first place, it is shown how a Focal species is translated into a UML class. Based on the formal syntax of UML and extended as explained above, the transformation rules are given according to the order in which each component of a UML class is specified. In the second place, it is shown how these rules are modified to consider the specificities relative to the generation of a UML class from a Focal collection. It should be noted that our transformation captures every aspect of the Focal specification language.

This transformation has been proved sound in the following way: firstly, by showing that the structure of each well-typed species (or collection) is preserved when transformed into UML, w.r.t. the Focal dedicated profile; secondly, by showing that the transformation of a well-typed Focal specification results into a well-formed UML model. This notion of soundness actually states that typing is preserved from Focal to UML (even if the well-formedness rules are said to characterize the semantics of UML). Another form of soundness, not considered here, would be to establish that the semantics of Focal is also preserved by the transformation, which is equivalent to show that there exists a model of the UML metamodel (together with the profile), for which the well-formedness rules are correct and which is compatible with a model of Focal.

An implementation of the previous transformation has been developed and consists of two parts. In the first part, we define a UML profile for the Focal specification language through the use of the UML2 Eclipse plug-in. This plug-in provides an implementation of the UML 2.1 metamodel and its integrated OCL checker allows us to validate the constraints defined in our profile. The ability to specify statically defined profiles also facilitates the definition of the operations and derived attributes characterizing each stereotype constituting our profile. This step is essential as it provides the necessary tool to validate the UML models to which our profile is applied. In fact, each OCL constraint specified in our profile is parsed and evaluated at runtime. This mechanism offers a convenient way to validate the soundness of our transformation. The second part concerns the development of an XSLT stylesheet that specifies the rules to transform a Focal specification generated in FocDoc format [99] (an XML schema used by the compiler for documentation) into a UML model expressed in the XMI interchange format [143].

4.1.3 *Airport Security Regulations*

To illustrate our transformation process, we consider a relatively concise example extracted from the Focal formalization realized within the EDEMOI project [131], and also described in [57] (see Section D.5 of Appendix D). This concerns the specification established for cabin persons. In the Focal formalization, the corresponding species is defined as follows:

```
species cabinPerson (cb is cabinBaggage) =
  rep;
  sig equal in self → self → bool;
  sig identityVerified in self → bool;
  sig cabinBaggage in self → cb;
  property equal_reflexive: all x in self, !equal (x, x); ...
end
```

It can be observed that *cabinPerson* is a parameterized species and its representation is left undefined. We also assume that the representation of species *cabinBaggage* is still abstract. To give an example of inheritance and show how the abstraction of a concrete representation is handled during the transformation process, we also introduce collection *cabinPerson_col*, which provides an implementation for species *cabinPerson*:

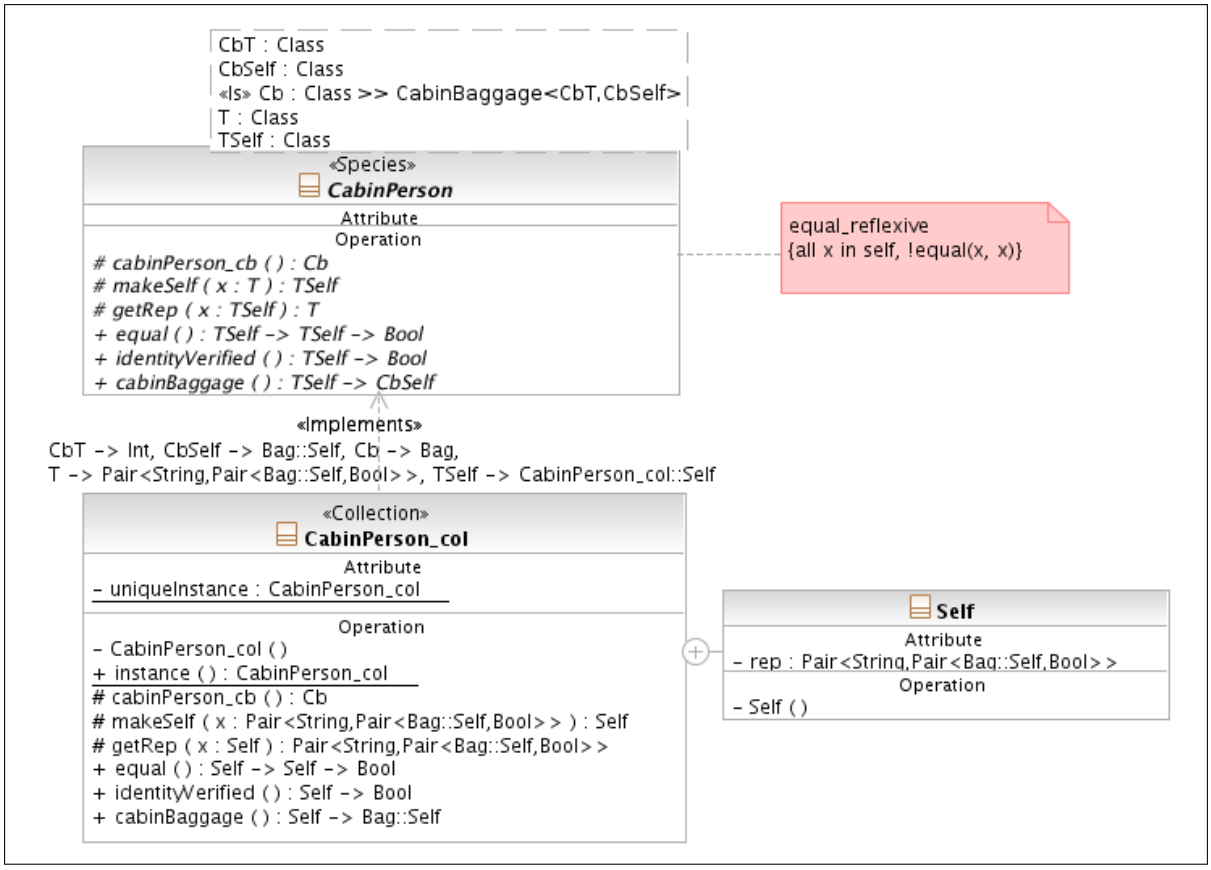

Figure 2.: CabinPerson Classes

```
collection cabinPerson_col implements cabinPerson (bag) =
  rep = string * bag * bool;
  let name (s in self) in string = #first (s);
  let cabinBaggage (s in self) in bag = #first (#scnd (s));
  let identityVerified (s in self) in bool = #scnd (#scnd (s)); ...
end
```

where *#first* and *#scnd* are respectively the first and second projections of a pair.

In this collection, the representation is specified as a triple, with the functions *name*, *cabinBaggage* and *identityVerified* defined accordingly. In the "implements" clause, species *cabinPerson* is instantiated with *bag*, which is a collection derived from *cabinBaggage*.

By applying the rules of our transformation described in [57] (see Section D.5 of Appendix D) and more exhaustively in [66], the UML classes shown in Figure 2 (using the corresponding graphical visualization) are obtained.

It should be noted that the generated UML classes are quite close to the initial Focal specification. As a consequence, if such a translation appears quite appropriate for developers when documenting their specifications, it is not suitable for evaluation purposes because the produced UML models are too close to the implementation and contain too many details in particular. Thus, in the future, we expect to use the present transformation rules as a basis to generate higher-level views that would be closer to conceptual models (similar to those produced at the preliminary stage

of the EDEMOI project [93]), and hence more pertinent for certification authorities or more generally end-users (not only for developers). Another perspective is to apply our transformation process to more concrete specifications (the formal models realized within the framework of the EDEMOI project are quite abstract), such as the standard library of Focal, which consists of a large formalization of computer algebra. In this way, it would be possible to see whether the generated UML models are fairly comprehensible and can be used for managing libraries. Finally, we aim to generate more dynamic views of the formal models (sequence and state-transition diagrams) through static analysis performed on the definitions involved in Focal specifications. This evolution should be directly related to the integration of temporal mechanisms to the Focal language, which might allow the expression of complex behavioral properties and which have been told to be a desirable coming feature in Section 2.1 of Chapter 2.

4.2 A MODULE-BASED MODEL FOR FOCAL

Contribution in collaboration with
N. Bertaux (Master student; see Subsec.C.2.2 of Appx. C).
CPR team, CEDRIC (CNAM), Paris (France), 2008.
Published in [21].

This contribution also concerns the Focal environment [13, 132], and its compiler more precisely. As said in Appendix A, Focal is equipped with a compiler producing OCaml code [128] for execution and Coq code [129] for certification, and the objective of this contribution is to propose a compilation scheme based on modules, which is supposed to be an alternative to the current scheme using records and aims to provide a higher level view of compiled specifications supplying in particular traceability.

4.2.1 *High-Level Compilation Schemes*

The rational behind the Focal language [13, 132] (see Appendix A) relies on the intention of providing a language in which it is possible to write highly structured specifications, and which is based on several paradigms, going from abstract data types to object-oriented programing. To understand the motivations and especially the foundations of Focal, we have actually to go back to the middle of 90's with the informal discussions which took place within the BiP working group animated in particular by T. Hardin, V. Viguié Donzeau-Gouge and J. R. Abrial. From this group composed of experts both in Coq [129] and B [1] emanated the idea of a language with more structured specifications than the rather "flat" formalizations made in Coq and with a notion of incremental development in the idea of B's refinement. Shortly afterwards, the Focal project (initially Foc project) was started in 1997 by T. Hardin and R. Rioboo with, in particular, an initial case study, which was consisting in implementing a library of computer algebra. For this purpose, a new language was designed, in which it is possible to build applications step by step, going from abstract specifications, called species, to concrete implementations, called collections. These different structures are combined using inheritance and parameterization, inspired by

object-oriented programming. Moreover, each of these structures is equipped with a carrier set, providing a typical algebraic specification flavor.

Basically, the underlying model of Focal, as presented in [77], consists of a class built over a data type and which provides functionalities over this data type in an approach similar to that of abstract data types, but with all the power of multiple inheritance and redefinition. This is exactly how it is encoded in the experiment described above and which consists in producing UML models from Focal specifications (see Subsection 4.1). The elaboration of such a model has allowed S. Boulmé to develop a formal specification of it in Coq [26], which has shown in particular how the logical consistency could be ensured in this model. V. Prevosto thereafter developed a compiler for this language [111], able to produce OCaml code [128] for execution and Coq code [129] for certification. In the first versions of the compiler, the implementation was using the object-oriented features of OCaml to produce the computational part of Focal specifications, while the Coq compilation was relying on an encoding using records (since Coq is not object-oriented). However, if the OCaml compilation was having the advantage of sticking to the initial model of Focal, the presence of two different compilation models for OCaml and Coq was quite unsatisfactory, as the code certified in Coq was actually not the code to be executed in OCaml. To palliate this problem of distance between execution and certification, a common compilation model based exclusively on records [112] was thereafter implemented and is currently the model used by the latest version of the compiler [133].

As can be noticed, the compilation scheme of Focal is of high level as it actually relies on high level target languages. Nevertheless, the data structures used for the compilation, namely records, are of low level and tend to break the structure of Focal specifications. Thus, to keep the highly structured nature of Focal specifications, we propose, in this contribution, a model of Focal relying on modules, which is supposed to be an alternative to the actual compiler using records, and which can be applied both to OCaml and Coq since these two languages offer a module system. As modules are higher level structures than records, such a compilation allows us to preserve, at a certain extent, the structure of Focal specifications in the compiled code and then provides traceability w.r.t. these specifications, which is not possible with the model based on records where the notion of inheritance disappears and where specifications must be flattened.

4.2.2 *Module-Based Compilation*

In the following, we present an overview of the module-based compilation scheme from Focal to OCaml and Coq through a small example. In particular, we focus on the compilation of the representation of a species and a simple case of inheritance. For more complex examples and for an exhaustive description of this compilation scheme, the reader is invited to refer to [21]. The considered example concerns the compilation of the predefined species *setoid* (and by extension of the species *basic_object*, which is a root species for Focal specifications), which represents a non-empty set supplied with a decidable equality.

The basic idea of the module-based compilation for OCaml and Coq is that a species corresponds to a functor parameterized by some attributes still abstract and a collection corresponds to a module resulting from the application of a functor representing the implemented species to modules representing the actual parameters provided to the species. We suppose that the reader is familiar with OCaml and Coq, and with their respective module systems in particular; otherwise, the reader can refer to [128, 129] for more information regarding both systems.

In Focal, every specification usually starts with the following predefined root species *basic_object*, which provides an abstract representation in particular:

```
species basic_object =
  rep;
  let print (x in self) = "<abst>";
  let parse (x in string) in self = #foc_error ("not parsable");
end
```

where *#foc_error* is the operator to signal exceptions.

To compile this species in OCaml using modules, we should first remark that in OCaml, modules cannot be partially defined, contrary to Focal species where not only representations can be abstract, but also functions or properties. To keep this abstraction in OCaml, the idea is to create a functor parameterized by the attributes still abstract (typically, representations and functions). Thus, the considered species is compiled into the following functor *Basic_object*:

```
module type BASIC_OBJECT =
sig
  type self
  val print : self → string
  val parse : string → self
end

module Basic_object (Abs : sig type self end) :
  BASIC_OBJECT with type self = Abs.self =
struct
  type self = Abs.self
  let print (x : self) = "<abst>"
  let parse (x : string) : self = failwith "not parsable"
end
```

In Coq, the module system offers a quite mixin-oriented approach, in the sense that a module and even a module type may contain abstract and defined attributes (typically, declarations and definitions, but also axioms and theorems). This approach is probably one of the most appropriate to model the semantics of Focal and this allows us to get rid of this notion of module including the abstract attributes (module *Abs* in OCaml), and which appears as a parameter of the functor representing the compiled species. The representation, if abstract, must still be a parameter, but does not need to be included in the module signature representing the interface of the species as required by OCaml (see module type *BASIC_OBJECT*), since we can use a

parameterized module signature, which is a feature recently provided by Coq. The Coq compilation is the following:

```
Module Type REP.
  Parameter t : Set.
End REP.

Module Type BASIC_OBJECT (Self : REP).
  Parameter print : Self.t → string.
  Parameter parse : string → Self.t.
End BASIC_OBJECT.

Module Basic_object (Self : REP) <: BASIC_OBJECT (Self).
  Definition print (x : Self.t) : string := "<abst>".
  Definition parse (x : string) : Self.t :=
    foc_error Self.t "not parsable".
End Basic_object.
```

where *foc_error* is a function encoding the corresponding exception operator.

In the following, we focus on the functor corresponding to the compiled species (typically, *Basic_object* in the previous example), and we do not provide the module signature representing the interface of this species (i.e. *BASIC_OBJECT* in the previous example).

In Focal, the notion of non-empty set provided with a decidable equality is introduced by species *setoid*, which inherits from species *basic_object*:

```
species setoid inherits basic_object =
  sig equal in self → self → bool;
  sig element in self;
  let different (x, y) = #not_b (!equal (x, y));
  property equal_reflexive : all x in self, !equal(x, x);
  theorem same_is_not_different : all x y in self,
    !different (x, y) ↔ not (!equal (x, y))
  proof: def !different; ...
end
```

where *#not_b* is the negation over type *bool*.

The OCaml compilation of this inheritance is made by means of the inclusion of a module which results from the instantiation of the functor corresponding to the inherited species. The actual parameter of this functor is a module containing the attributes which are abstract in the inherited species and which may be either still abstract or concrete in the sub-species. In our case, this module only includes the representation, which is still abstract. The compilation is as follows:

```
module Setoid
  (Abs : sig
     type self
     val equal : self → self → bool
     val element : unit → self
   end) : SETOID with type self = Abs.self =
```

```
struct
  include Basic_object (struct type self = Abs.self end)
  let equal = Abs.equal
  let element = Abs.element
  let different x y = not (equal x y)
end
```

The Coq compilation of this inheritance is rather similar and is also realized through the inclusion of the module which corresponds to the instantiation of the functor representing the inherited species. As seen previously, this instantiation only concerns the representation. The compilation is the following:

```
Module Setoid (Self : REP) <: SETOID (Self).
  Include Basic_object (Self).
  Parameter equal : Self.t → Self.t → bool.
  Parameter element : Self.t.
  Definition different (x y : Self.t) : bool := negb (equal x y).
  Axiom equal_reflexive : forall x : Self.t, Is_true (equal x x).
  Theorem same_is_not_different : forall x y : Self.t,
    Is_true (different x y) ↔ Is_true (negb (equal x y)).
  Proof. ...
End Setoid.
```

The previous case of inheritance is actually quite simple, as there are few dependencies. For example, in OCaml, the only dependency occurs in the inheritance module (the module which is included), which depends on the actual module of abstractions (the module containing the instantiations of the attributes of the inherited species previously abstract). However, some other dependencies may appear when we concretize a function previously abstract using a function which is added in the considered species, or when a function which is added in the considered species, depends on a function coming from the inheritance. This new dependencies then imply mutual dependencies between the module of inheritance, the actual module of abstractions, and the module gathering the functions of the compiled species. In OCaml, this is handled by means of a block of recursive modules. In addition, the inclusion mechanism of OCaml ("include" expression) is replaced by a selective inclusion, as the module of inheritance and the module of the functions of the compiled species may overlap. This new inclusion mechanism also allows us to deal with multiple inheritance. This manual inclusion is actually not problematic since it is intended to be automatically generated; moreover, this inclusion is even preferable as each attribute of a species appears explicitly in the module representing this species.

In Coq, the absence of the module of abstractions allows us to avoid the use of a block of recursive modules. The compilation is made by means of a selective inclusion of the module corresponding to the instantiation of the inheritance functor, and which consists in only including inherited attributes which are not defined or redefined in the compiled species. The attributes added in the compiled species are then also included. As in OCaml, the generated code does not use the primitive inclusion of Coq.

Another difficulty appears when compiling late binding. In OCaml and in modules in particular, function calls are statically linked. As a consequence, a redefinition

implies that every function referring to this redefined function cannot be inherited as it refers to the former definition of this function and not to the latter. To solve this problem without having to repeat the code of every function referring to the redefined function, we introduce the notion of function generator (this system is also used in the model based on records [112]). A function generator is a function based on the previous defined function where every reference to another function of the species has been abstracted. The corresponding defined function is then obtained applying its function generator to the actual functions of the species that have been abstracted in the function generator. For each function requiring the use of a function generator, the corresponding function generator is added to the module representing the compiled species and can then be reused later by inheritance.

In Coq, the redefinition of a function poses the same problem with wider influences. In the same way, we have to use function generators for defined functions using a redefined function. However, the dependencies w.r.t. a redefined function also concerns properties, whose statements as well as the proofs may depend on this function. Therefore, we have to introduce the notion of property generator, which actually consists of two generators: a statement generator and a proof generator (if the property is a theorem). Similarly to function generators, these two generators are functions which make an abstraction of the functions, but also of the properties, respectively involved in the statement and the proof of a property. For proof generators, the abstraction of a function is made only if the proof does not depend on the definition of this function, as the proof is invalidated if this function is redefined. As in OCaml, all the generators are included in the module representing the compiled species.

As can be seen, the thorny points of the above compilation essentially resides in the compilation of the object-oriented features of Focal. This is not surprising as we know that module and object paradigms are rather orthogonal. To deal with all the potentially problematic cases, the compilation has been completely formalized and can be found in [21]. This formalization tends to show that thanks to modules, which are high level structures, this compilation scheme can preserve the structure of Focal specifications and ensures a certain traceability. The next step is to develop an implementation of this compilation scheme, which should allow us to assess the feasibility of such an approach in practice. In addition, this work has allowed us to compare the module systems of OCaml and Coq, and to show their respective evolutions. In particular, since the first versions of modules in Coq [36], remarkable improvements were carried out [116], such as the introduction of parameterized module signatures or the possibility of using abstract attributes in modules and module signatures. Some additional changes are being developed along with the possibility of concretezing abstract attributes for example. All these improvements and changes tend to significantly facilitate the compilation of Focal in Coq using modules, even if we are still far from the Focal model described in [77].

5

CONCLUSION

5.1 ACHIEVEMENTS

In this document, we present different contributions which tend to propose several improvements in the use of theorem proving along three well-identified lines of work. The first line of work is structuring (see Chapter 2), which consists in focusing on the way of building appropriately structured specifications, as it has obviously some direct consequences on the way of certifying them for example. In this line of work, we describe three contributions. In the framework of the EDEMOI project, the first contribution consists of the formalization of airport security regulations using the Focal environment. This experiment has allowed us to assess the appropriateness of the design features of Focal applied to a real-world example, as well as the effectiveness of Zenon, the automated reasoning support of Focal. Along the same lines of clearly distinguishing specification from implementation as introduced by the Focal language, we present a second contribution, which resides in an extraction procedure of functional code from inductive relations and which has been implemented in the context of the Coq proof assistant and the Focalize environment (the successor of Focal). The third contribution highlights the consequences of highly structured specifications over tools intended to support a theorem prover, with the elaboration of an information retrieval procedure in proof libraries using type isomorphisms and implemented in the framework of Coq. In particular, we show how the introduction of dependent types involves major modifications of the corresponding theory of type isomorphisms compared to the theory for CCCs or that for ML.

The second line of work developed in this document is automating (see Chapter 3), which relies on the credo that a theorem prover must offer a suitable level of automation and/or appropriate means to enhance this automation. In this line of work, we detail three contributions. The first contribution focuses on the way of increasing the power of automation of a theorem prover with the introduction of an alternative meta-language, which allows us to write not only small but also complex automation routines in the context of Coq. The second contribution deals with some possible interactions between deduction and computer algebra in a pure skeptical way (i.e. verifying the soundness of the computations). In this contribution, three experiments have been conducted. The first one consists of the development of an interface between Coq and the Maple computer algebra system, which allows us to import into Coq computations from Maple over fields. In a second experiment, this interface has been extended to deal with computations of gcds over polynomials, in order to implement a quantifier

elimination procedure over algebraically closed fields in the framework of Coq. In the continuity of the previous procedure, a third experiment consists in designing a procedure for Focal to test the validity of first-order properties over real closed fields using the computation of cylindrical algebraic decomposition performed by a routine of the Axiom computer algebra system. Lastly, the third contribution proposes to adapt the idea of skeptical computations to automated deduction with two studies related to the Zenon automated theorem prover. The first study deals with the proofs generated by Zenon, and which are translated into Coq proofs for checking, while the second study consists in validating supplementary rules involved in applications developed using the B method [1] by means of Zenon proofs, which are translated back to B proofs.

Finally, the third line of work detailed in this document is communicating (see Chapter 4), which aims to draw attention to several means of communicating between theorem provers and end-users. This line of work consists of three contributions. The first contribution deals with the input language of a theorem prover and presents a language for Coq to describe proofs, which has the advantage to be style-independent in the sense that it gathers the three well-identified proof styles, i.e. the procedural, declarative and proof-term styles. Conversely, the second contribution focuses on the output language of a theorem prover and proposes a transformation from Focal specifications to UML models, which appears quite appropriate as a means of automatic documentation and especially as a means of producing comprehensible documents for end-users. In particular, in the context of the EDEMOI project, we can hopefully expect that documents in UML are a good basis to converse with certification authorities. The third and last contribution introduces another scheme of compilation for Focal, which is based on the notion of modules and which is supposed to be an alternative to the current scheme using records. This new compilation model has the advantage of providing a higher level view of compiled specifications supplying in particular traceability w.r.t. the initial Focal specifications.

5.2 PERSPECTIVES

In the previous chapters describing the three lines of work detailed in this document (Chapters 2, 3 and 4), some perspectives in the short and medium terms have been already introduced. Here, we aim to present long term and more ambitious perspectives by wondering what the next-generation theorem provers could look like and what kind of features could be worth working on. This is obviously a challenge itself to anticipate the future research directions of the community in the domain of theorem proving, but the following perspectives can also be seen as trends in which the author strongly believes.

The first set of perspectives is related to the development of the Focal environment, since in the last decade, Focal has allowed us to focus on the importance of structuring specifications (see Chapter 2). The formalization of airport security regulations in the framework of the EDEMOI project [131] and described in Section 2.1 of Chapter 2 has shown the appropriateness of the design features of Focal (as well as the reasoning support ensured by Zenon), but it has also highlighted some limitations, which we have

to deal with if we want to make Focal evolve. Among these limitations, there is the integration of temporal mechanisms to the language in order to allow the expression of behavioral properties. These behavioral properties may be either properties involving synchronization mechanisms, or properties where physical time occurs. To handle time properties, we need to appeal to temporal logics, which can be seen as extensions of classical logic (the basic logic of Focal), in order to formalize the required behavioral properties. As a temporal logic for Focal, a possibility is to consider TLA [94] (Temporal Logic of Actions), since recent experiments [32, 33] have been conducted in the framework of TLA+ [96], in which Zenon is able to manage TLA proofs of safety properties and produce Isabelle proofs [136] for checking.

To deal with time properties, a temporal logic is not enough. We must also be able to produce code satisfying these properties. The very functional Focal code allows us to do so, but appears not to be readable as there is no notion of internal state. To remain in the Focal approach (i.e. as functional as possible to make the activity of proving easier), we aim to experiment some reactive features, based on the synchronous model. The synchronous approach [76] relies on an ideal model where computations and communications are supposed to be instantaneous. In this model, time is logically defined as a sequence of reactions to input signals. Thus, the reaction of the system to its environment is supposed to be instantaneous. The main advantage of this approach is that it is possible to specify deterministic behaviors even in presence of parallelism and communications. This determinism can be preserved from specification to implementation and allows us to perform formal verification, which remains one of our main objectives in the Focal project. For the choice of a reactive programming language for Focal, some languages like Lucid Synchrone [110] or Reactive-ML [102] are quite good candidates to be investigated. The advantage of considering these languages is that they rely on two different programming paradigms, i.e. the data flow and control styles. This allows us to keep a possibility of choice. But especially, these two languages can be compiled to OCaml, which allows us to focus on the problem of compiling them to Coq. At a certain extent, this simplifies the design of the compilation schemes for Focal.

Still in the context of the EDEMOI project, the basic design pattern of Focal appeared a little inadequate. Even though parameterization provides a form of parametric and bounded polymorphism, the absence of free subtyping hinders the factorization of some security properties. Thus, a security property related to a species appearing as a parameter must be reinstantiated for each sub-species of this species. If the factorization is effective in the Focal specification, it is not the case in the compiled code where the code of the instantiated species is duplicated for each instantiation. To palliate this drawback, J.-F. Étienne proposed in his PhD thesis [66] the use of explicit conversion functions between sub-species and super-species. However, he did not thoroughly investigate the implications that this approach may have on concrete implementations. Though this solution appears to be appropriate, it may nonetheless leave space to specification errors, as we cannot always guarantee that a species is indeed a proper subtype of a given super-species. Hence, the ultimate solution to this problem might be to consider an enhancement of the Focal specification language with improved subtyping capabilities. Another point related to this problem is the specific use of the object-oriented features in Focal, where it is actually impossible to

use species and inheritance to build data types, as it is usually done in object-oriented programming languages. This is not possible because Focal relies on the principle that every species shares the same representation along the inheritance path. As a consequence, very few design patterns coming from the community of object-oriented programming can be applied. To deal with this problem, a solution would be to relax a little the constraint over representations and to allow extensible representations. Thus, a sub-species could not change the representation of its super-species, but just add some type components to this representation. In this way, object-oriented data types could be introduced, while a new algorithm of computation of dependencies should be considered as all the theorems of the super-species depending on the representation should be reproved (or at least completed).

Some other limitations have been underlined during the development of the certified library of computer algebra by R. Rioboo (it was the initial case study of Focal). Among others, there is the possibility of writing recursive species, which is currently not allowed in Focal. For instance, the problem may occur when formalizing real closed fields (required in the experiment described in Subsection 3.1.3 of Chapter 3). A possible definition is the following: a real closed field is a field E s.t. there is a total order on E making it an ordered field s.t. in this ordering, every positive element of E is a square in E and any polynomial of odd degree with coefficients in E has at least one root in E. This definition is recursive since to define E, we need to consider polynomials with coefficients in E. This point is currently not solved, even if a compilation to OCaml has been proposed. Another limitation emphasized by the development of this library is the absence of invariants over representations. However, in mathematics, this is frequently used when defining quotients over sets for example. R. Rioboo proposes a solution in [113], but this solution is not primitive and it could be worth supplying Focal with a built-in system of representation invariants. To do so, a possibility could consist in using the concrete types with invariants studied in the framework of the Quotient project [148], and implemented in a tool, called Moca [23], able to generate construction functions for OCaml data types with invariants.

All the extensions proposed above have to be formally and semantically founded. This could be done by integrating these extensions to the abstract model of Focal proposed by S. Boulmé [26], and which is strongly inspired by contextual categories introduced by J. Cartmell [31] and considered to axiomatize dependent records. Nevertheless, this model is quite abstract and far from a concrete implementation of Focal. To mitigate this problem, a more operational model of Focal has been elaborated by S. Fechter [67], but this model essentially relies on formal models of object-oriented programming languages and therefore only focuses on the computational behavior of Focal specifications leaving properties and theorems uninterpreted. Thus, if we want to formalize any extension of Focal, a preliminary work should consist in elaborating an operational model able to also deal with properties and theorems. Once the semantics of these extensions precised, an appropriate model of compilation will have to be found and integrated to the compiler initially implemented by V. Prevosto [111] (and re-implemented later under the name of Focalize by F. Pessaux). By appropriate, we mean a model which is at least consistent and at best correct w.r.t. the semantics of these extensions expressed in the operational model. To do so, an encoding of Focal in a well-identified calculus, with polymorphism and dependent types in particular,

could be an option, as Focal would not only benefit from the properties (such as consistency) of this calculus, but it would also allow us to precise the place of Focal in the land of these calculi.

A second set of perspectives concerns the interactions of deduction and computer algebra introduced in Section 3.1 of Chapter 3. Over the last 10 past years, the author has been very implied in this domain with a progressive line of work. First, there was the development of the Coq tactic "field" for dealing with equalities over fields. Next, the author implemented an interface between Coq and Maple, which was able to import Maple computations into Coq and to certify them by means of the "field" tactic. This interface was extended afterwards to deal with computation of gcds over polynomials in order to design a proof procedure for Coq over algebraically closed fields. Finally, a test procedure for real closed fields was elaborated for Focal using a CAD implementation in Axiom. In this succession of experiments, the author aimed to keep a skeptical approach and deal with ever more complicated fields, going from basic fields to real closed fields passing by algebraically closed fields. A natural perspective for this work is to transform the test procedure for real closed fields implemented as an interface between Focal and Axiom into a proof procedure for Focal, which would certify the computation of CAD provided by Axiom. As said in Subsection 3.1.3 of Chapter 3, the problem is difficult and is actually equivalent to asking the following question: given a real closed field E and A a subset of I^r, is it possible to verify that a given decomposition D of E^r is an A-invariant CAD of E^r? This problem is undecidable in general (verifying that a collection of sets is a decomposition of E^r is still undecidable), but some results consisting in dealing with specific dimensions (such as dimension 2, for instance), or related to the connexity of regions could be exploited to find an appropriate notion of certificate. This experiment should be a significant work and might require some new results about CAD. It should also be noted that our approach remains skeptical (with an external computation), and is therefore an alternative to similar but autarkic developments of CAD, such as [101].

A

THE FOCAL ENVIRONMENT

A.1 WHAT IS FOCAL?

To understand the motivations and especially the foundations of the Focal language [13, 132], we have to go back to the middle of 90's with the informal discussions which took place within the BiP working group animated in particular by T. Hardin, V. Viguié Donzeau-Gouge and J. R. Abrial. From this group composed of experts both in Coq [129] and B [1] emanated the idea of a language with more structured specifications than the rather "flat" formalizations made in Coq and with a notion of incremental development in the idea of B's refinement. Moreover, it was important for this new language to be strongly typed, probably in a slightly less powerful way than in Coq, but in a more elaborated way than the set theory present in B.

The Focal project (initially Foc project) was started in 1997 by T. Hardin and R. Rioboo with, in particular, the PhD thesis of S. Boulmé [26]. In this thesis, a new language was designed, in which it is possible to build applications step by step, going from abstract specifications, called species, to concrete implementations, called collections. These different structures are combined using inheritance and parameterization, inspired by object-oriented programming. Moreover, each of these structures is equipped with a carrier set, providing a typical algebraic specification flavor. V. Prevosto thereafter developed a compiler for this language [111], able to produce OCaml code [128] for execution, Coq code for certification, but also code for documentation [99]. D. Doligez also provided a first-order automated theorem prover, called Zenon [24], which helps the user to complete his/her proofs in Focal through a declarative-like proof language. This automated theorem prover can produce pure Coq proofs, which are reinserted in the Coq specifications generated by the Focal compiler and fully verified by Coq.

As an initial case study, a certified Computer Algebra library (distributed with the compiler as the standard library of Focal) was developed by R. Rioboo (with significant efficiency results compared to existing Computer Algebra systems). Later, other formalizations were carried out regarding airport security regulations [54, 56] (see Chapter 2, Section 2.1) by J.-F. Étienne, V. Viguié Donzeau-Gouge and the author, security policies [86] (Bell-LaPadula and more advanced models) by M. Jaume and C. Morisset, and more recently, models of component-based systems in the framework of the REVE project [149]. All these formalizations tend to show that Focal can be considered as a general-purpose specification language, appropriate not only for mathematics but also for real-world applications.

A.2 SPECIFICATION: SPECIES

The first major notion of the Focal language [13, 132] is the structure of *species*, which corresponds to the highest level of abstraction in a specification. A species can roughly be seen as a list of attributes of three kinds:

- the carrier type, called *representation*, which is the type of the entities that are manipulated by the functions of the species; the representation can be either abstract or concrete;
- the functions, which denote the operations allowed on the entities of the representation; the functions can be either *definitions* (when a body is provided) or *declarations* (when only a type is given);
- the properties, that must be verified by any further implementation of the species; the properties can be either simply *properties* (when only the proposition is given) or *theorems* (when a proof is also provided).

The syntax of a species is the following:

```
species <name> =
  rep [= <type>];                (* representation *)
  sig <name> in <type>;          (* declaration *)
  let <name> = <body>;           (* definition *)
  property <name> : <prop>;      (* property *)
  theorem <name> : <prop>        (* theorem *)
  proof : <proof>;
end
```

where <*name*> is simply a given name, <*type*> a type expression (mainly typing of core-ML without polymorphism but with concrete data types), <*body*> a function body (mainly core-ML with conditional, pattern-matching and recursion), <*prop*> a (first-order) proposition and <*proof*> a proof (expressed by means of a declarative proof language). In the type language, the specific expression "self" refers to the type of the representation and may be used everywhere except when defining a concrete representation.

As said previously, species can be combined using (multiple) inheritance, which works as expected. It is possible to define functions that were previously only declared or to prove properties which had no provided proof. It is also possible to redefine functions previously defined or to reprove properties already proved. However, the representation cannot be redefined and functions as well as properties must keep their respective types and propositions all along the inheritance path. Another way of combining species is to use parameterization. Species can be parameterized either by other species or by entities from species. If the parameter is a species, the parameterized species only has access to the interface of this species, i.e. only its abstract representation, its declarations and its properties. These two features can be used simultaneously within the same species and complete the previous syntax given above as follows:

```
species <name> (<name> is <name>[(<pars>)], <name> in <name>, ...)
        inherits <name>, <name> (<pars>), ... = ...
end
```

where <*pars*> is a list of <*name*>, which denotes the names used as effective parameters. When the parameter is a species parameter declaration, the "is" keyword is used. When it is an entity parameter declaration, the "in" keyword is used.

A.3 IMPLEMENTATION: COLLECTION

The other main notion of the Focal language is the structure of *collection*, which corresponds to the implementation of a specification. A collection implements a species in such a way that every attribute becomes concrete: the representation must be concrete, functions must be defined and properties must be proved. If the implemented species is parameterized, the collection must also provide implementations for these parameters: either a collection if the parameter is a species or a given entity if the parameter denotes an entity of a species. Moreover, a collection is seen (by the other species and collections) through its corresponding interface; in particular, the representation is an abstract data type and only the definitions of the collection are able to manipulate the entities of this type. Finally, a collection is a terminal item and cannot be extended or refined by inheritance. The syntax of a collection is the following:

```
collection <name> implements <name> (<pars>) = ... end
```

A.4 CERTIFICATION: PROVING WITH ZENON

The certification of a Focal specification is ensured by the possibility of proving properties. To do so, a first-order automated theorem prover (based on the tableau method), called Zenon [24], helps us to complete the proofs. Basically, there are two ways of building proofs with Zenon: the first one is to provide all the properties (proved or not) and definitions needed by Zenon to build a proof automatically; the second one is to introduce additional auxiliary lemmas (by means of a purely declarative proof language) to help Zenon find a proof. In the first way, proofs are described as follows:

```
theorem <name> : <prop>
proof : by <props> def <defs>;
```

where <*props*> is a list of properties and <*defs*> a list of definitions.

The proof language of the second option is inspired by a proposition by L. Lamport [95], which is based on a practical and hierarchical structuring of proofs using numeric labels for proof depth. The syntax is the following:

```
theorem <name> : <prop>
proof :
<<level>><label> assume <hyps> prove <prop>
<<level>><label> qed [by <props> def <defs>];
```

where <*level*> is a natural number, <*label*> a name and <*hyps*> a list of hypotheses (of the form "<*name*> : <*type*> or <*prop*>"). The "assume ... prove" expression is used to introduce a new goal to be proved ("assume" provides skolemization). The proof of the new goal is detailed in sub-levels, whereby the numeric label is increased accordingly. The "qed" expression closes a proof level, possibly with the help of some properties/definitions provided by the user through the "by ... def" expression.

A.5 FURTHER INFORMATION

For additional information regarding Focal and its applications, the reader can refer to [13, 132]. It should also be noted that a new version of the Focal compiler, called Focalize, has been recently released and is available at [133].

B

FORMER CONTRIBUTIONS

This appendix presents three contributions by the author, which respectively lie within the three lines of work considered in this document, i.e. structuring (Chapter 2), automating (Chapter 3), and communicating (Chapter 4). These contributions are a little more former than those described in the previous chapters, since they are mainly based on the Master/PhD theses of the author. However, they are analyzed with the benefit of hindsight, and some perspectives are proposed relying on the current trends and the work realized since then. More precisely, the first contribution consists in retrieving information, basically theorems, in proof libraries using types as keys and up to isomorphisms. In particular, a search procedure has been developed in a calculus including polymorphism, dependent types and strong sum types, and has been implemented in an earlier version of the Coq proof assistant [129]. The second contribution resides in the design of a tactic language, called $\mathcal{L}_{tac}$, developed in the framework of Coq. This new meta-language allows the user to write not only small and local automation routines, but also significant and complex proof procedures. Finally, the third contribution consists of a new proof language, developed in the context of Coq. This proof language is intended to be independent of a given proof style, and therefore allows the user to develop proofs in procedural, declarative and proof-term based styles.

B.1 INFORMATION RETRIEVAL IN PROOF LIBRARIES

Contribution in collaboration with B. Werner and R. Di Cosmo.
Coq project (INRIA), Rocquencourt (France), 1997.
Published in [43, 48, 42].

As seen in Chapter 2, it is quite important to develop well structured specifications, as it provides some appropriate properties, such as maintainability or reusability for example. However, these more and more structured specifications impose deep changes in the methods used to retrieve information in these specifications. In particular, these deep changes require a good knowledge of the structures used in the specifications in order to make these methods at least effective and at best efficient. In the following, we describe a contribution which consists of a method of information retrieval in proof libraries using type isomorphisms, and which was experimented in the context of the Coq proof assistant [129].

B.1.1 *Use of Type Isomorphisms*

When dealing with information retrieval, it seems clear that using identifiers as keys is quite ineffective. Finding a piece of code, a given theorem or anything else just by providing its name is more a matter for pure coincidence than for a thoughtful approach. Thus, a more effective approach consists in considering types (if available) as search patterns and performing comparisons modulo a given equivalence. Again, this equivalence between types must be elaborated enough, otherwise the corresponding search is quite ineffective. For instance, the syntactic equality is inappropriate as it does not deal with the problem of argument permutations in functions for example. The most favorable concept for search in libraries has been highlighted by M. Rittri [114], and is that of type isomorphism. One of the main interests of this concept is that it has been a very intensive research domain, and it has been studied for many years by R. Di Cosmo, G. Longo, K. Bruce and S. Soloviev [115, 29, 62] for example. However, when using type isomorphisms for retrieval information, the main difficulty resides in building a theory with appropriate properties. In particular, the theory must be obviously at least correct, but it is also desirable to have a complete and decidable theory. As expected, the more sophisticated the typing system is, the more difficult it is to find a suitable theory. Typically, the presence of high-level structures, such as modules or objects, tends to make the theory much more complicated. In addition, to make information retrieval more effective, it is necessary to add unification in the type isomorphism theory. This problem is even more complicated, as the unification modulo the whole theory is generally undecidable even when considering the simplest theory (see below).

The basic theory from which every theory is actually built and used for information retrieval is that one established for Closed Cartesian Categories (CCCs) by S. Soloviev [115], and which consists of the seven following type equalities:

1. $A \times B = B \times A$
2. $A \times (B \times C) = (A \times B) \times C$
3. $(A \times B) \to C = A \to B \to C$
4. $A \to (B \times C) = (A \to B) \times (A \to C)$
5. $A \times \top = A$
6. $A \to \top = \top$
7. $\top \to A = A$

where A, B and C are arbitrary types, and $\top$ is a constant for the unit type.

This theory has been shown correct and complete for CCCs. Furthermore, it is also a decidable theory, which can be implemented by simple term-rewriting algorithm and which has been used by M. Rittri [114] for information retrieval in function libraries of the functional language Lazy ML [10]. Afterwards, R. Di Cosmo enhanced this theory with polymorphism and unification [61], and a tool based on this theory was implemented for information retrieval in libraries of Caml Light [127]. One of the

main goals of the contribution described below was actually to deal with information retrieval in proof libraries (i.e. searching at the same time functions and theorems), which required to extend the above theory in order to consider more complicated type systems able to totally specify the behavior of functions in particular.

B.1.2 *Application to Proof Libraries*

Contrary to libraries of functions, libraries of proofs contain both functions and theorems (properties with proofs). If the types of functions may be simple (if they are just partial specifications of functions), the types of theorems may be quite complicated depending on the expressiveness of the corresponding type system. As a consequence, it is necessary to explicitly define the domain of types for which we want to write a theory of type isomorphisms, since these types may be very rich. In the present contribution [43], the idea was to develop an information retrieval tool for the Coq proof assistant [129], which relies on the theory of Calculus of Inductive Constructions (CIC), a quite expressive type theory. The objective of this tool was obviously not to consider the whole theory of CIC, and the aimed extension was to deal with dependent types (dependent products and Σ-types) in order to manage types of theorems effectively. This extension may probably seem minor, but it has some major consequences over the theory of type isomorphisms, as well as over the corresponding search method. Here are some of difficulties we faced and the different solutions we provided:

- When looking for a given theorem and when finding another theorem with a different type (but isomorphic to the search type), it is necessary to be able to apply this theorem using an appropriate function of conversion. To do so, the corresponding theory must only deal with definable type isomorphisms, i.e. type isomorphisms with conversion functions which allow us to pass from one type to the other one and vice versa, and which can be expressed in the considered language. This condition together with the presence of dependent types imposes to explicitly keep track of the conversion functions in the several equalities representing the theory of type isomorphisms. For example, given two types A and B and the corresponding conversion functions $\sigma : A \to B$ and $\tau : B \to A$, it makes no sense to consider an isomorphism between $\Pi x : A.C$ and $\Pi x : B.C$; there is actually no reason for having both these two types well formed at the same time. However, we can exhibit conversion functions between $\Pi x : A.C$ and $\Pi x : B.C[x \leftarrow (\tau\ x)]$. The situation is quite similar with Σ-types. This necessity of keeping the conversion function attached to the equalities of type isomorphisms is not really surprising, and was also observed when dealing with type isomorphisms in presence of modules [3, 4].

- A property of conversion functions is that they commute, i.e. considering the two previous types A and B with their conversion functions σ and τ, we have $\sigma \circ \tau = Id_B$ and $\tau \circ \sigma = Id_A$, where Id_B and Id_A are respectively the identity functions over types B and A. To ensure this property, we have to modify the conversion rule by extending the reduction rules with more extensional simplifications like η-conversion or surjective pairing. Up to now, the main reason

for considering these additional reductions was to make the theory complete, that is to have the syntactical and semantical notions coincide. In the case of calculi with dependent types, the situation changes. Namely, these generalized η-reductions become necessary not only for a matter of completeness, but also, more drastically, to be able to build up a theory compatible with the typing. For instance, given the two types A and B above with their conversion functions, the two types $\Pi x : A.C$ and $\Pi x : B.C[x \leftarrow (\tau\ x)]$ are isomorphic and their conversion functions are the following:

$\lambda f : (\Pi x : A.C).\lambda x : B.f\ (\tau\ x)$
$\lambda f : (\Pi x : B.C[x \leftarrow (\tau\ x)]).\lambda x : A.f\ (\sigma\ x)$

The last function is of type $\Pi x : B.C[x \leftarrow (\tau\ x)] \rightarrow \Pi x : A.C[x \leftarrow (\tau\ (\sigma\ x))]$, which is only of the expected type if the term $(\tau\ (\sigma\ x))$ can be reduced to x possibly using the extensional rules. This example also shows that the contextual closure of the basic equalities of type isomorphisms is not straightforward, as a replacement of a type by another type may induce some changes over the context.

- The previous requirement of extending the reduction rules is actually problematic when the goal is to develop an information retrieval procedure for the Coq proof assistant. As it is not possible to extend the reduction rules of Coq in a standard way, the basic idea consists in only considering the reduction rules of Coq, such as β-reduction. However, restricting reduction rules makes the theory incorrect; typically, the above example does not work. Therefore, the idea is to allow a given type replacement which implies the presence of a conversion function in the type context only if the conversion functions of this type replacement commute using only β-reduction. Thus, the above example is allowed only if $(\tau\ (\sigma\ x))$ can be reduced to x by β-reduction. With this restriction, the theory is obviously incomplete, but remains correct. This incompleteness has some consequences over the term-rewriting algorithm used to implement a search tool for Coq. In particular, the rewriting system is not confluent. For example, the type $\Pi x : \top.\Sigma y : A.B$ can be rewritten into two different types: either $(\Sigma y : A.B)[x \leftarrow *]$ or $\Sigma y : (\Pi x : A.\top).\Pi x : \top.B[y \leftarrow (y\ x)]$, where $*$ is the unique element of the $\top$ type; this critical pair cannot be reduced since this requires a replacement in the type of y in the second term which involves additional reduction rules concerning the $\top$ type. This problem of confluence implies to impose a partial order over the application of rewriting rules and we obtain normal forms (probably more complicated than could be expected) that can be compared up to permutation of the Σ-components of the type (in the same way as the procedures of M. Rittri and R. Di Cosmo respectively do for Lazy ML and Caml Light).

An implementation of the theory seen above (it is actually an implementation of a sub-theory of this theory corresponding to the restriction over the reduction rules)

was developed for the Coq proof assistant (for some sub-versions of version 6), and was able to browse the whole standard library of Coq.

This work should be able to be reused in the framework of the Focal environment [13, 132], introduced previously. Even if properties are first-order propositions in Focal, the notion of species introduces both polymorphism and dependent types (see Appendix A) by means of parameterization. The species parameters (using the "is" keyword) are actually type parameters and are similar to parametric polymorphism, whereas the entity parameters (using the "in" keyword) allow a type to get dependencies w.r.t. given terms and therefore provide a sort of dependent types. However, the previous work regarding the dependent types of Coq cannot be directly applied in the context of Focal, since the notions of polymorphism and dependent types are actually embedded in the concept of species, and thus some preliminary work is necessary. In particular, it is important to understand how the theory of Focal can be interpreted w.r.t. the existing Pure Type Systems (PTSs). For instance, species clearly introduce dependent types, but probably under a weaker form than in $\lambda\Pi$, and this point remains to be precised. Once this relationship with PTSs clarified (by means of a specific encoding of Focal in a given PTS for example), it should be possible to apply to Focal the work developed for Coq and probably in a more straightforward way.

B.2 A PROOF DEDICATED META-LANGUAGE

[*Contribution in collaboration with B. Werner.*
Coq project (INRIA), Rocquencourt (France), 1997-2001.
Published in [44, 46, 51, 45].]

The second contribution consists of the description of a tactic language, called $\mathcal{L}_{tac}$ and designed in the framework of the Coq proof assistant [129]. This language is intended to provide a higher power of automation for the proof language while remaining quite abstract w.r.t. the implementation language of the proof system.

B.2.1 *Evolution of Meta-Languages*

In the early 1970's, M. J. C. Gordon, R. Milner and C. P. Wadsworth designs a formal reasoning assistant, called LCF [75] (the short for "Logic for Computable Functions") and initially implemented at the universities of Edinburgh and Stanford. As part of the LCF paradigm, there is a notion of Meta-Language [74] (ML for short), as opposed to the object language (i.e. the logic language). This language is functional and allows us to implement an abstract data type of theorems, as well as some techniques to build proofs automatically. Initially, the first versions of LCF were implemented in Lisp, but with the evolution of ML, which is now a quite general-purpose programming language (with numerous variants), most of the direct descendants of LCF use ML also an implementation language. For example, this is the case of Coq [129] or HOL [135]. This evolution of ML has some interesting consequences over the LCF-like proof assistants. In particular, this fusion between the meta-language and the implementation language allows us to write any tactic, which can have stronger and deeper interactions with the system. This is a significant evolution, which makes the

design of more complex tactics possible. However, the choice of such a meta-language has also several consequences and constraints that must be considered. Here are some of these consequences and constraints:

- The proof assistant must provide the means to prevent possible inconsistencies arising from user tactics. This can be done in various ways. For example, in LCF and HOL, this is done by means of an abstract data type of theorems and tactics must produce objects of this type. In Coq, this is achieved by the type-checker, which verifies the term built by a tactic is of the expected type.
- The user must learn another language which is, in general, quite different from the proof language. Therefore, it is important to consider how much time the user is ready to spend on this task which may be quite difficult or at least, tedious.
- The language must come with a dedicated debugger, because finding errors in tactic code is much harder than in proof scripts developed in the proof language, where the system is supposed to assist the user in locating errors.
- The proof system must have a clear and a well documented code, especially for the proof engine part. The user must be able to easily and quickly identify the necessary primitives or he/she could easily get lost in the development code.
- The tactics are not portable (contrary to the first versions of LCF, where the system evolutions did not affect the meta-language layer) and must be maintained.
- The language is quite general-purpose and does not provide proof dedicated procedures in a primitive way.

Thus, writing tactics in a full programmable language involves many constraints not only for developers, but also for users. The idea of a new meta-language, which would evolve between the proof and implementation languages, started from this observation. This language did not have to be fully programmable, but instead had to provide an appropriate set of proof combination primitives, which would allow the user to easily write powerful tactics. From this idea, a new tactic language, called $\mathcal{L}_{tac}$, was then introduced in the Coq proof assistant.

B.2.2 *The $\mathcal{L}_{tac}$ Meta-Language*

The introduction of $\mathcal{L}_{tac}$ [44, 46, 45] in Coq (from version 7) can be seen as an extension of the set of proof combination primitives, called tacticals in Coq. This set of tacticals was actually very basic (branching, loops, etc), and the idea of $\mathcal{L}_{tac}$ was to provide a more expressive language with new operators dedicated to proof engineering in particular. Therefore, $\mathcal{L}_{tac}$ consists of a small functional core with recursion to have some higher order structures and with pattern-matching operators both for terms as well as for proof contexts to handle the proof process. The power of $\mathcal{L}_{tac}$ actually resides in the specificity of the pattern-matching operators, which offer non-linear first and second order unification and can perform backtracking over each clause of the

pattern-matching. In addition, subterm pattern-matching is also available, with the possibility of handling the contexts of the matched subterms and with a backtracking support over the several occurrences of a given subterm pattern. For the full syntax definition of $\mathcal{L}_{tac}$, the reader can refer to the reference manual of Coq [129].

The different operators of $\mathcal{L}_{tac}$ allow us to write powerful tactics quite easily and in a very compact way. Moreover, these tactics are defined directly in the proof language and the user just has to learn a very minimal set of operators. Thus, most of the constraints regarding the use of a meta-language too close to the implementation language (see above) vanish with the introduction of this more domain-specific meta-language. However, this meta-language also requires to pay attention to some of its specific features. In particular, a dedicated debugger comes with $\mathcal{L}_{tac}$, as it appears very difficult to debug manually tactics using backtracking; it is also necessary to provide appropriate means to track typing errors occurring in Coq terms, as tactics can build arbitrary and potentially not well-typed terms. Another point, we have to pay attention to, is that $\mathcal{L}_{tac}$ is not fully programmable, so that some very complex tactics may require to switch to OCaml, the implementation language of Coq. Nevertheless, $\mathcal{L}_{tac}$ still helps at this level, since it is possible to include $\mathcal{L}_{tac}$ code in OCaml code by means of a system of quotations provided by Camlp5 [41]; the converse is also possible using antiquotations which import OCaml code into $\mathcal{L}_{tac}$ code embedded in quotations.

Initially, $\mathcal{L}_{tac}$ was designed to deal with small and local automation, typically in the toplevel when the user is actually doing a proof. However, $\mathcal{L}_{tac}$ appeared much more powerful than expected and could be used to deal with non-trivial problems of automation. For example, a tactic, called "tauto" and able to prove intuitionistic propositional formulas (based on the contraction-free sequent calculi LJT* of R. Dyckhoff [65]), was developed by the author. There was already a version of this tactic written in OCaml and compared to this version, we observed several significant differences and gains. In particular, we obtained a drastic reduction in size (from 2000 lines to 40 lines of code), as well as a considerable increase in performance (up to 95% for some examples). These spectacular results can be explained by the use of some of the proof-dedicated operators (mainly the backtracking pattern-matching over proof contexts), which allows us to express Dyckhoff's algorithm quite naturally. Another non-trivial example is the development still by the author of the tactic "field" [51, 52] (see Section D.3 of Appendix D), which aims to solve equalities over fields. This tactic was coded in a total reflexive way [27, 78] and fully in $\mathcal{L}_{tac}$, even for the reification step of the reflexion process, which is usually done in OCaml (since it requires to perform logical introspection). In addition, some other significants developments were also realized by the user community, which has generally provided quite positive feedbacks w.r.t. the introduction of $\mathcal{L}_{tac}$ in Coq; see [35, 85] for instance. For other examples of complex tactics written in $\mathcal{L}_{tac}$, the reader can refer to [45, 34].

B.2.3 *Future of Meta-Languages*

The $\mathcal{L}_{tac}$ tactic language provides several evolutions in the domain of meta-languages for proof assistants. In particular, it advocates to move the meta-language closer to the proof language, even if it compels the meta-language to be distinct from the

implementation language of the proof assistant. It allows the user to write tactics in a well constrained environment, which prevents him/her from building incorrect tactics in a very convenient way and directly within the proof assistant. Another evolution provided by $\mathcal{L}_{tac}$ is the idea of making the meta-language more specific to the domain of formal proofs. This idea is essentially achieved by the introduction of backtracking pattern-matching operators, which offers the features of a functional language with a typical flavor of logic programming. Thus, if we had to summarize the foundation of $\mathcal{L}_{tac}$, we would say that ML was initially designed to write tactics for LCF-like proof assistants, but evolved in such way that it has become a quite general-purpose language which does not provide appropriate built-in primitives for the design of tactics; the aim of $\mathcal{L}_{tac}$ is to give a genuine and suitable meta-language back to proof assistants.

These different evolutions of meta-languages raise the following question: what future can we expect for meta-languages adapted to proof assistants? Here are some elements of answer we can bring in the context of $\mathcal{L}_{tac}$ (many of them are essentially experiment feedbacks and therefore deserve to be considered in the short term):

- Future proof dedicated meta-languages will have to correctly handle pattern-matching over terms in presence of binders. In $\mathcal{L}_{tac}$, this feature is not managed in a fully satisfactory way. In particular, if the pattern-matching of a term containing binders is actually straightforward, the pattern-matching of a binder or under a binder is made by means of a specific second-order operator, of which the use is restricted. This mechanism deserves to be more flexible, and even if we are dependent on the De Bruijn representation used by Coq for terms with binders, it should be of great help to study the many different binder representation strategies used to address the PoplMark challenge [12] in particular. Among these alternative solutions, there are Higher-Order Abstract Syntax (HOAS) [109], nominal syntax [69], or locally nameless syntax [73]. In addition, it should be noted that a recent contribution [117] tends to tackle this problem of pattern-matching with binders in a context similar to that of $\mathcal{L}_{tac}$.

- Complex and efficient automation procedures generally make use of tables. These tables allow us to store propositions, terms, names, and any information that may be useful during the process of building proofs. However, as the logic language is usually purely functional, these tables must be provided by the meta-language. Since $\mathcal{L}_{tac}$ is a layer over ML, this means that appropriate commands must be implemented in ML and made available at the $\mathcal{L}_{tac}$ level. Such functionalities should allow us to write more purely $\mathcal{L}_{tac}$ tactics. This is the case of the tactic "field" [51, 52] (see Section D.3 of Appendix D) for example, where a table is used to store the field theories. A formalization of the introduction of tables in a similar context is described in [117], and could certainly be implemented for $\mathcal{L}_{tac}$.

- As $\mathcal{L}_{tac}$ tends to go away from ML and to move closer to the logic layer, the question of which data types for $\mathcal{L}_{tac}$ may be asked. Even if it seems clear that ML data types cannot be used, this point is not really critical, since for a language like $\mathcal{L}_{tac}$, it is still possible to use all the data types of the logic layer and to

handle them by means of pattern-matching. For instance, the reification part of reflexive tactics [27, 78] can be implemented by means of maps of terms, which are directly expressed using the logic language (as in the tactic "field"). However, we can wonder if it is the way to go. The answer is not clear-cut, as the logic language offers a full range of data types, but also with some constraints of typing which does not actually concern the meta-language (for example, a list of terms imposes to handle terms of same type, which is very restrictive from the meta-language point of view). As a consequence, a set of predefined data types with appropriate operators might be a reasonable solution. It raises the problem of determining which data types could be worth being included in this set though.

- In $\mathcal{L}_{tac}$, tactics are either applied to a goal, or produce a term. These two kinds of behaviors cannot actually be merged, and a desirable extension could be to allow a same tactic to produce a term while also being applied to a given goal as a sort of side effect. With such an extension, we must become aware of the difficulty to deal with backtracking, since pattern-matching operators over proof contexts and terms both provide backtracking and it is not clear how these two sources of backtracking can be mixed within a same tactic.

B.3 FREE-STYLE THEOREM PROVING

[*Contribution in collaboration with B. Werner.*
Coq project (INRIA), Rocquencourt (France), 1997-2001.
Published in [47, 45].]

The third contribution consists of the description of a proof language, called $\mathcal{L}_{pdt}$ and designed in the framework of the Coq proof assistant [129]. This language is intended to be the fruit of a fusion between several proof styles, and to therefore provide a significant degree of flexibility for the user when developing his/her proofs.

B.3.1 *The Several Proof Styles*

A proof language is a language used to describe proofs in a proof assistant. Here, the word "proof" means a script (of instructions or expressions) to be presented to a machine for checking. As expected, there are several ways of explaining a proof to a proof assistant, which all of them rely on the proof engine of the proof assistant. These questions were actually not at the center of concerns when the first implementations of proof assistants appeared in the late 1960's, and we had to wait for the middle 1990's that the seminal paper of J. Harrison [80] provides some elements of comparison between the different existing proof styles. In particular, procedural and declarative styles are contrasted each other in this paper.

The procedural style consists in giving instructions to a proof machine, which stores the state of the proof. Procedural proofs are naturally backward-oriented, as the philosophy relies on tackling the goal to be proved providing information as less as possible. As a consequence, if such proofs are quite appropriate when completed

interactively, they suffer from a lack of readability though, and therefore from a lack of maintainability. This tends to make such proofs quite sensitive to the changes of the proof assistant. All the same, most of current interactive theorem provers are actually based on procedural proof languages, such as Coq [129], HOL [135], or PVS [146] for example. On the contrary, in the declarative style, the user has to declare auxiliary lemmas, which can be seen as logical cuts and which once combined are intended to prove the main goal. In this way, declarative proofs appear more forward-oriented, as the auxiliary lemmas may be as close as desired to the hypotheses used to prove the main goal. Such proofs are quite readable because the intermediate states of proof explicitly appear in the proof script, and they are therefore much more maintainable than procedural proofs. However, an immediate drawback is the verbose nature of these proofs, where parts of the initial proposition to be proved must be repeated as many times as necessary. This tends to make such proofs very fragile to the changes of specification. In addition, the user has to be very well informed of the automation procedure which combines the several auxiliary lemmas. There are actually few theorem provers based on declarative proof languages; we can mention in particular Mizar [140], TLA+ [96] (of which the proof language is described in [95]), or Focal [13, 132] (of which the proof language is inspired by that of TLA+; see Appendix A).

There is actually a third alternative to the procedural and declarative proof styles, which consists in using the language of proof terms. This language relies on the Brouwer-Heyting-Kolmogorov interpretation, which provides an interpretation of intuitionistic proofs, and by extension, on the Curry-Howard isomorphism, which considers propositions as types and proofs as terms. In such context, searching for a proof of a given proposition is then simply equivalent to building a term of such type. When building such a proof term, we are obviously guided by the goal (the type), which makes this kind of proofs more backward-oriented as in the procedural style. With appropriate tools (typically tools providing means to simplify the filling of placeholders), it is possible to write such proofs quite easily. As for readability, it probably requires a little practice when reading terms as proofs, but a term is full of information and nothing is actually hidden compared to the procedural approach for instance. Regarding maintainability, a proof term is robust to the changes of the proof assistant (if it does not concern its underlying logic), but as declarative proofs, it may be very impacted by changes of specification. There are a number of proof assistants using intuitionistic logics and in which it is possible to directly provide terms as proofs (even if it is often not the default way to build proofs), such as Coq [129], LEGO [137], or NuPRL [141]. But the best example of such proof assistants is certainly Alfa [124] (the successor of ALF), as it works primitively with proof terms, which are the only way to interact with the proof editor.

B.3.2 *The $\mathcal{L}_{pdt}$ Proof Language*

Following the previous observations, it is possible to bring out when and where these different proof styles are useful and should be used. Thus, we will procedural proofs for small proofs, known to be trivial and realized interactively in backward mode, for

which we are not interested in the formal details. These proofs must be seen as black boxes. Declarative proofs will be used for more complex proofs that we would like to build more in forward mode (as a mathematical proof in a textbook), in a batch way and very precisely, i.e. providing much information to the reader. Finally, proof terms will be also used for complex proofs, but backward-oriented, built either interactively or in batch mode (both methods are actually appropriate), and for which we can choose the level of details (it is possible to hide some type signatures). Thus, these three proof styles seem to correspond to specific needs, and actually do not deserve to be opposed.

The idea of $\mathcal{L}_{pdt}$ [47, 45] is to amalgamate the three proof styles identified above, i.e. the procedural, declarative and proof term based styles. This language was formalized in the framework of the Coq proof assistant [129], as it primitively provides both procedural proofs and proof terms. As for declarative features, they were actually easily simulated in this procedural environment by means of simple logical cuts. Beyond being the unique language proposing a fusion of three proof styles, the novelty of $\mathcal{L}_{pdt}$ also resides in the formalization of its semantics. As far as the author knows, Alfa [124] is the only system which has a formally described proof language [100]. The formalization of the $\mathcal{L}_{pdt}$ semantics actually goes further, as it also deals with procedural and declarative proofs. The corresponding semantics is a big-step semantics, which handles goals under the form of global and local contexts, and terms which may contain metavariables and which may be refined by side-effects. The reader can refer to [47, 45] for more details regarding this semantics, and also for some examples of use of $\mathcal{L}_{pdt}$. In addition, an implementation of $\mathcal{L}_{pdt}$ was also realized as a prototype for Coq (for some sub-versions of version 7).

B.3.3 *The Next Proof Languages*

The approach of the $\mathcal{L}_{pdt}$ proof language tends to show that none of the considered proof styles is actually the panacea with which it would be possible to heal the difficulty for users of interacting with proof assistants. Thus, instead of taking sides for or against a given style, $\mathcal{L}_{pdt}$ proposes not only a language merging the several proof styles, but also a sort of methodology indicating how and when using a given style. However, we can wonder if it is the good way to go, and in particular, what the current trends in terms of proof languages actually consist of.

Historically, due to the paper [80] of J. Harrison, declarative proof style impressively bounced back, with many experiments to introduce declarative features on top of procedural framework [79, 156, 155, 70, 39], and also to build purely declarative environments [120, 158]. However, this craze for declarative style was a little surprising, since the paper of J. Harrison also pointed out the difficulty of introducing too many formulas in proof scripts, which tends to increase the viscosity of the proofs; this was also noted later in [104]. As a matter of fact, we had to bow to the evidence that it was difficult to get the best of the two worlds. If $\mathcal{L}_{pdt}$ tends to bring a solution, it is also difficult to assess this solution as it was actually never used by Coq users (the corresponding prototype was not released), contrary to the $\mathcal{L}_{tac}$ language [44, 46, 45] (see Section B.2). Failing that, some of the current trends aim to still promote

declarative proofs while keeping their viscosity low. One way to do so should consist in making mechanized proofs and proofs from textbooks closer, as we know that in paper proofs, we take the liberty of making a number of shortcuts, which avoid to repeat formulas in particular. This approach is proposed in [152], where a specialized format of proof intends to reconcile published proofs and proof scripts. Another experiment quite close to this idea was conducted by the MathLang project [87], with the aim of developing an approach for computerizing mathematical texts and knowledge, which allows various degrees of formalization, and which is compatible with different logical frameworks and proof systems.

However, it is also important to become aware that the approaches above impose an appropriate automation, able to combine proof statements as desired in the initial textbooks. The problem is actually even more general, and consists in understanding what place automation should occupy in the future proof languages. Automated proofs must be handled with care, as they are black boxes and therefore break the properties of readability and maintainability. Even if the user is given the possibility of consulting an automated proof, the result is generally unreadable and of little use (see Section 3.2 of Chapter 3 for example). As for maintainability, the user has little influence over automation, and is subjected to the changes of automation. Thus, the scope of automation must be clearly delimited. The limits of this scope are actually closely related to the proof style. For instance, a user of procedural proofs may be surprised by a too powerful automation, which performs more proof steps than expected, while a user of declarative proofs may be disappointed by a too weak automation, which cannot combine proof statements as combined in his/her paper proof. Thus, depending on the proof style, automation must be well balanced or at least well controlled, as pointed out by [88, 157] for example respectively in procedural and declarative settings, and should therefore play a full role in the design of the next proof languages.

C

STUDENT SUPERVISION

C.1 PHD STUDENTS

C.1.1 *Jean-Frédéric Étienne (2004-2008)*

Supervision: David Delahaye (50%) and Véronique Donzeau-Gouge (50%, CNAM).
Time/Grant: from January 2004 to July 2008 (defended on July 7, 2008); MAE grant.
Current Job: Software engineer at SafeRiver (security and dependability).
Title: "Certifying Airport Security Regulations using the Focal Environment".
Abstract: In the framework of the EDEMOI project, this PhD thesis was aiming to integrate and apply several requirements engineering and formal methods techniques to analyze regulations in the domain of airport security. In particular, one of the objectives was to apply the Focal tool to a concrete study case, namely the formalization of airport security regulations. The idea was to answer two needs: the formalization itself for the EDEMOI project, and the assessment of the Focal environment over a real-world study case.

C.1.2 *Pierre-Nicolas Tollitte (2009-now)*

Supervision: David Delahaye (50%) and Catherine Dubois (50%, ENSIIE).
Time/Grant: from October 2009; MESR doctoral contract.
Title: "A Formal Verification Tool for Modeling Effective Mathematics".
Abstract: This PhD thesis comes within the scope of strong applicative problematics and consists in developing a complete Focal environment, which allows us to model effective mathematics. Concretely, it aims to carry on with the formalization effort of computer algebra realized in Focal, in order to bring into relief some possible limitations of the design features of the Focal language in particular. The objective is then to elaborate the corresponding extensions to Focal, while keeping these extensions semantically and formally founded, and to integrate them to the compiler.

C.1.3 *Mélanie Jacquel (2010-now)*

Supervision: David Delahaye (40%), Catherine Dubois (20%, ENSIIE), and Karim Berkani (40%, Siemens Transportation Systems).
Time/Grant: from January 2010; ANRT CIFRE contract.
Title: "A Mechanized Proof Method for Automating Proofs in the Set Theory of B".

Abstract: The goal of this PhD thesis consists in increasing the automation of the Atelier B, and therefore in significantly reducing the development costs by automatically discharging a maximum of proof obligations. More precisely, the idea is to work in the context of an external formal environment (developed in Coq), which allows us to validate rules to be added to the theory of B.

C.2 MASTER AND ENGINEERING STUDENTS

C.2.1 *Yuan Gang (2003)*

Supervision: David Delahaye (100%).
Time/Degree Course: from April to September 2003; MOCS Master 2nd year.
Title: "Compilation Models for the Focal Environment".
Abstract: This Master thesis was aiming to study several compilation models for the Focal language w.r.t. the target languages of the compiler, namely OCaml and Coq. In particular, the idea was to compare several models in terms of readability of the compiled code, which had to ensure both reusability and traceability. From this point of view, a module-based model appeared to be quite appropriate and was formalized afterwards during Nicolas Bertaux's Master thesis (see below).

C.2.2 *Nicolas Bertaux (2008)*

Supervision: David Delahaye (100%).
Time/Degree Course: from April to September 2008; MOCS Master 2nd year.
Title: "A Module-Based Model for the Focal Environment".
Abstract: The objective of this Master thesis was to elaborate a compilation scheme based on modules, and which was supposed to be an alternative to the current scheme (implemented in the compiler) using records. This new compilation model had the advantage to provide a higher level view of compiled specifications supplying in particular traceability.

C.2.3 *Pierre-Nicolas Tollitte (2009)*

Supervision: David Delahaye (50%) et Catherine Dubois (50%, ENSIIE).
Time/Degree Course: from January to June 2009; ENSIIE 3rd year.
Title: "Generating Certified Functional Code from Inductive Specifications in Focalize".
Abstract: This engineer's thesis was aiming to propose a method for generating functional code from inductive specifications in the framework of the Focalize environment (successor of Focal). This method was consisting of a preliminary mode consistency analysis, which was verifying that a computation was possible w.r.t. the selected inputs/outputs, and the code generation itself. The produced code was certified in the sense that it was systematically supported by a correctness proof also generated in Focalize.

C.2.4 *Sanaa Toumi (2009)*

Supervision: David Delahaye (50%) et Renaud Rioboo (50%, ENSIIE).
Time/Degree Course: from April to September 2009; Master MOCS 2nd year.
Title: "Using CAD for Testing First-Order Formulas over Real Closed Fields in Focal".
Abstract: The goal of this Master thesis was to develop a test procedure for verifying first-order formulas over a real closed field using a Cylindrical Algebraic Decomposition (CAD for short) of this field. The procedure had to be implemented as an interface between the Focal environment and the Axiom computer algebra system, in which there was in particular an implementation of Collins' algorithm able to produce a CAD for a given real closed field.

C.2.5 *Benjamin Lalière (2009)*

Supervision: David Delahaye (100%).
Time/Degree Course: from May to June 2009; ESILV 2nd year.
Titre: "Implementation of a Parser for OpenMath".
Abstract: This engineer's thesis was aiming to develop a parser for OpenMath, which is a language based on XML and which allows us to represent mathematical objects. The OpenMath language is typically used to exchange mathematical data between several programs, and this parser for OpenMath was intended to be used for dealing with the data exchanges occurring in the interface integrating the Axiom CAD to Focal and previously developed by Sanaa Toumi (see above).

D

PUBLICATION ADDENDUM

This appendix provides 5 publications in their entirety, and which intend to support Chapters 2, 3 and 4. More precisely, papers of Section D.1 and D.2 are related to Chapter 2, papers of Section D.3 and D.4 to Chapter 3, and finally paper of Section D.5 to Chapter 4.

D.1 PAPER 1: AIRPORT SECURITY REGULATIONS IN FOCAL

This paper is related to Chapter 2 and has been published in [54].

Certifying Airport Security Regulations using the Focal Environment

David Delahaye, Jean-Frédéric Étienne,
and Véronique Viguié Donzeau-Gouge

CEDRIC/CNAM, Paris, France,
David.Delahaye@cnam.fr, etien_je@auditeur.cnam.fr,
donzeau@cnam.fr

Abstract. We present the formalization of regulations intended to ensure airport security in the framework of civil aviation. In particular, we describe the formal models of two standards, one at the international level and the other at the European level. These models are expressed using the Focal environment, which is also briefly presented. Focal is an object-oriented specification and proof system, where we can write programs together with properties which can be proved semi-automatically. We show how Focal is appropriate for building a clean hierarchical specification for our case study using, in particular, the object-oriented features to refine the international level into the European level and parameterization to modularize the development.

1 Introduction

The security of civil aviation is governed by a series of international standards and recommended practices that detail the responsibilities of the various stake-holders (states, operators, agents, etc). These documents are intended to give the specifications of procedures and artifacts which implement security in airports, aircraft and air traffic control. A key element to enforce security is the conformance of these procedures and artifacts to the specifications. However, it is also essential to ensure the consistency and completeness of the specifications. Standards and recommended practices are natural language documents (generally written in English) and their size may range from a few dozen to several hundred pages. Natural language has the advantage of being easily understood by a large number of stake-holders, but practice has also shown that it can be interpreted in several inconsistent ways by various readers. Moreover, it is very difficult to process natural language documents automatically in the search for inconsistencies. When a document has several hundred pages, it is very difficult to ensure that the content of a particular paragraph is not contradicted by some others which may be several dozen pages from the first one.

This paper aims to present the formal models of two standards related to airport security in order to study their consistency: the first one is the international standard Annex 17 [7] (to the Doc 7300/8) produced by the International Civil Aviation Organization (ICAO), an agency of the United Nations; the second one

is the European standard Doc 2320 [2] (a public version of the Doc 30, which has a restricted access status) produced by the European Civil Aviation Conference (ECAC) and which is supposed to refine the first one at the European level. More precisely, from these models, we can expect:

1. to detect anomalies such as inconsistencies, incompleteness and redundancies or to provide evidence of their absence;
2. to clarify ambiguities and misunderstandings resulting from the use of informal definitions expressed in natural language;
3. to identify hidden assumptions, which may lead to shortcomings when additional explanations are required (e.g. in airport security programmes);
4. to make possible the rigorous assessment of satisfaction for a concrete regulation implementation and w.r.t. the requirements.

This formalization was completed in the framework of the EDEMOI[1] [8] project, which aims to integrate and apply several requirements engineering and formal methods techniques to analyze regulation standards in the domain of airport security. The methodology of this project may be considered as original in the sense that it tries to apply techniques, usually reserved to critical software, to the domain of regulations (in which no implementation is expected). The project used a two-step approach. In the first step, standards described in natural language were analyzed in order to extract security properties and to elaborate a conceptual model of the underlying system [5]. The second step, which this work is part of, consists in building a formal model and to analyze/verify the model by different kinds of formal tools. In this paper, we describe two formal models of the two standards considered above, which have been carried out using the Focal [12] environment, as well as some results that have been analyzed from these models.

Another motivation of this paper is to present the Focal [12] (previously Foc) environment, developed by the Focal team, and to show how this tool is appropriate to model this kind of application. The idea is to assess and validate the design features as well as the reasoning support mechanism offered by the Focal specification and proof system. In our case study, amongst others, we essentially use the features of inheritance and parameterization. Inheritance allows us to get a neat notion of refinement making incremental specifications possible; in particular, the refinement of the international level by the European level can be expressed naturally. Parameterization provides us with a form of polymorphism so that we can factorize parts of our development and obtain a very modular specification. Finally, regarding the reasoning support, the first-order automated theorem-prover of Focal, called Zenon, bring us an effective help by automatically discharging most of the proofs required by the specification.

The paper is organized as follows: first, we give a brief description of the Focal language with its main structures and features; next, we present our case study, i.e. the several standards regulating security in airports and in particular,

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

those we chose to model; finally, we describe the global formalization made in Focal, as well as the properties that could be analyzed and verified.

2 The Focal environment

2.1 What is Focal?

Focal [12], initiated by T. Hardin with R. Rioboo and S. Boulmé, is a language in which it is possible to build applications step by step, going from abstract specifications, called species, to concrete implementations, called collections. These different structures are combined using inheritance and parameterization, inspired by object-oriented programming; moreover, each of these structures is equipped with a carrier set, providing a typical algebraic specification flavor. Moreover, in this language, there is a neat separation between the activities of programming and proving. A compiler was developed by V. Prevosto for this language, able to produce Ocaml [11] code for execution, Coq [10] code[2] for certification, but also code for documentation (generated by means of structured comments). More recently, D. Doligez provided a first-order automated theorem prover, called Zenon, which helps the user to complete his/her proofs in Focal through a declarative-like proof language. This automated theorem prover can produce pure Coq proofs, which are reinserted in the Coq specifications generated by the Focal compiler and fully verified by Coq.

2.2 Specification: species

The first major notion of the Focal language is the structure of *species*, which corresponds to the highest level of abstraction in a specification. A species can be roughly seen as a list of attributes and there are three kinds of attributes:

- the carrier type, called *representation*, which is the type of the entities that are manipulated by the functions of the species; representations can be either abstract or concrete;
- the functions, which denote the operations allowed on the entities; the functions can be either *definitions* (when a body is provided) or *declarations* (when only a type is given);
- the properties, which must be verified by any further implementation of the species; the properties can be either simply properties (when only the proposition is given) or theorems (when a proof is also provided).

More concretely, the general syntax of a species is the following:

[2] Here, Coq is only used as a proof checker, and not to extract, from provided proofs and using its Curry-Howard isomorphism capability, Ocaml programs, which are directly generated from Focal specifications.

```
species <name> =

  rep [= <type>];                  (* abstract/concrete
                                     representation *)

  sig <name> in <type>;            (* declaration *)
  let <name> = <body>;             (* definition *)

  property <name> : <prop>;        (* property *)
  theorem <name> : <prop>          (* theorem *)
  proof : <proof>;

end
```

where <name> is simply a given name, <type> a type expression (mainly typing of core-ML without polymorphism but with inductive types), <body> a function body (mainly core-ML with conditional, pattern-matching and recursion), <prop> a (first-order) proposition and <proof> a proof (expressed in a declarative style and given to Zenon). In the type language, the specific expression **self** refers to the type of the representation and may be used everywhere except when defining a concrete representation.

As said previously, species can be combined using (multiple) inheritance, which works as expected. It is possible to define functions that were previously only declared or to prove properties which had no provided proof. It is also possible to redefine functions previously defined or to reprove properties already proved. However, the representation cannot be redefined and functions as well as properties must keep their respective types and propositions all along the inheritance path. Another way of combining species is to use parameterization. Species can be parameterized either by other species or by entities from species. If the parameter is a species, the parameterized species only has access to the interface of this species, i.e. only its abstract representation, its declarations and its properties. These two features complete the previous syntax definition as follows:

```
species <name> (<name> is <name>, <name> in <name>, ...)
        inherits <name>, <name> (<pars>), ... = ...
end
```

where <pars> is a list of <name> and denotes the names which are used as parameters. When the parameter is a species, the keyword is **is**, when it is an entity of a species, the keyword is **in**.

To better understand this notion of species, let us give a small example:

Example 1 (Finite stacks). To formalize finite stacks, an abstract way is to specify stacks (possibly infinite) first, and to refine them as finite stacks afterwards. The specification of stacks might be the following:

```
species stack (typ is setoid) inherits setoid =

  sig empty in self;
  sig push in typ -> self -> self;
  sig pop in self -> self;
  sig last in self -> typ;
  let is_empty (s) = !equal (s, !empty);

  property ie_empty : !is_empty (!empty);
  property ie_push : all e in typ, all s in self,
                          not (!is_empty (!push (e, s))); ...

end
```

where setoid is a predefined species representing a non-empty set with an equality (in the first line, the parameter and the inheritance from setoid show respectively that we want to be able to compare two items of a stack, but also two stacks), the "!" notation is equivalent to the common dot notation of message sending in object-oriented programming (**self** is the default species when there is no receiver species indicated; e.g. !empty is for **self**!empty).

Next, before specifying finite stacks, we can be more modular and formalize the notion of finiteness separately as follows:

```
species is_finite (max in int) inherits basic_object =

  sig size in self -> int;
  property size_max : all s in self, #int_leq (!size (s), max);

end
```

where basic_object is a predefined species supposed to be the root of every Focal hierarchy, int the predefined type of integers and "#int_" the prefix of operations over the type int. Here, we can remark that the species is parameterized by an entity of a species and not by a species.

Finally, we can formalize finite stacks using a multiple inheritance from the species stack and is_finite:

```
species finite_stack (typ is setoid, max in int)
          inherits stack (typ), is_finite (max) =

  let is_full (s) = #int_eq (!size (s), max);

  property size_empty : #int_eq (!size (!empty), 0);
  property size_push : all e in typ, all s in self, not (!is_full (s)) ->
     #int_eq (!size (!push (e, s)), #int_plus (!size (s), 1)); ...

end
```

2.3 Implementation: collection

The other main notion of the Focal language is the structure of *collection*, which corresponds to the implementation of a specification. A collection implements a species in such a way that every attribute becomes concrete: the representation must be concrete, functions must be defined and properties must be proved. If the implemented species is parameterized, the collection must also provide implementations for these parameters: either a collection if the parameter is a species or a given entity if the parameter denotes an entity of a species. Moreover, a collection is seen (by the other species and collections) through its corresponding interface; in particular, the representation is an abstract data type and only the definitions of the collection are able to manipulate the entities of this type. Finally, a collection is a terminal item and cannot be extended or refined by inheritance. The syntax of a collection is the following:

collection <name> **implements** <name> (<pars>) = ... **end**

We will not detail examples of collections here since our formalization (see Section 4) does not make use of them. Actually, the airport security regulations considered in this paper are rather abstract and do not expect any implementation. Regarding our previous example of finite stacks, a corresponding collection will have to provide a concrete representation (using lists for example), definitions for only declared functions (empty, push, pop, last) and proofs for properties (ie_empty, ie_push, etc). For complete examples of collections, the reader can refer to the standard library of Focal (see Section 2.5).

2.4 Certification: proving with Zenon

The certification of a Focal specification is ensured by the possibility of proving properties. To do so, a first-order automated theorem prover, called Zenon and based on the tableau method, helps us to complete the proofs. Basically, there are two ways of providing proofs to Zenon: the first one is to give all the properties (proved or not) and definitions needed by Zenon to build a proof with its procedure; the second one is to give additional auxiliary lemmas to help Zenon to find a proof. In the first option, Zenon must be strong enough to find a proof with only the provided properties and definitions; the second option must be considered when Zenon needs to be helped a little more or when the user likes to present his/her proof in a more readable form. In the first option, proofs are described as follows:

theorem <name> : <prop>
proof : **by** <props> **def** <defs>;

where <props> is a list of properties and <defs> a list of definitions.

The proof language of the second option is inspired by a proposition by L. Lamport [6], which is based on a practical and hierarchical structuring of proofs using number labels for proof depth. We do not describe this language

here but some examples of use can be found in the formalization of our case study (see Section 4.4 to get the development).

Let us describe a small proof in our example of finite stacks:

Example 2 (Finite stacks). In the species stack, we can notice that with the definition of is_empty, Property ie_empty can already be proved in the following way:

```
theorem ie_empty : !is_empty (!empty)
proof : by !equal_reflexive def !is_empty;
```

where equal_reflexive is the property of reflexivity for equality, which is inherited from the species setoid.

This proof uses the definition of is_empty, which means that any redefinition of is_empty in any further inheritance invalidates this proof (which has to be completed again using the new definition). Thus, w.r.t. usual object-oriented programming, redefinitions may have some additional effects since they directly influence the proofs in which they are involved.

2.5 Further information

For additional information regarding Focal, the reader can refer to [3], as well as to the Focal Web site: `http://focal.inria.fr/`, which contains the Focal distribution (compiler, Zenon and other tools), the reference manual, a tutorial, some FAQs and also some other references regarding, in particular, Focal's formal semantics (e.g. see S. Boulmé and S. Fechter's PhD theses).

3 Case study: airport security regulations

The primary goal of the international standards and recommended practices regulating airport security is to safeguard civil aviation against acts of unlawful interference. These normative documents detail the roles and responsibilities of the various stake-holders and pinpoint a set of security measures (as well as the ways and means to implement them) that each airport serving civil aviation has to comply with. In addition, the entire regulatory system is organized in a hierarchical way, where each level has its own set of regulatory documents that are drafted and maintained by different bodies. At the international level, Annex 17 [7] of the International Civil Aviation Organization (ICAO) lays down the general principles and recommended practices to be adopted by each member state. It is refined at the European level by the Doc 2320 [2] of the European Civil Aviation Conference (ECAC), where the standard is made more detailed and more precise. At the national level, each member state has to establish and implement a national civil aviation security programme in compliance with international standards and national laws. Finally, at the airport level, the national and international standards are implemented by an airport security programme.

All these documents are written in natural language and due to their voluminous size, it is difficult to manually assess the consistency of the entire regulatory system. Moreover, informal definitions tend to be inaccurate and may be interpreted in various inconsistent ways by different readers. Consequently, it may happen that two inspectors visiting the same airport at the same time reach contradictory conclusions about its conformity. However, these documents have the merit of being rigorously structured. Ensuring their consistency and completeness while eliminating any ambiguity or misunderstanding is a significant step towards the reinforcement of airport security.

3.1 Scope delimitation

After a deep study of the above-mentioned documents and several consultations with the ICAO and ECAC, we decided to take as a starting point the preventive security measures described in Chapter 4 of Annex 17. Chapter 4 begins by stating the primary goal to be fulfilled by each member state, which is:

> *4.1 Each Contracting State shall establish measures to prevent weapons, explosives or any other dangerous devices, articles or substances, which may be used to commit an act of unlawful interference, the carriage or bearing of which is not authorized, from being introduced, by any means whatsoever, on board an aircraft engaged in international civil aviation.*

Basically, this means that acts of unlawful interference can be avoided by preventing unauthorized dangerous objects from being introduced on board aircraft[3]. To be able to achieve this goal, the member states have to implement a set of preventive security measures, which are classified in Chapter 4 according to six specific situations that may potentially lead to the introduction of dangerous objects on board. These are namely:

- persons accessing restricted security areas and airside areas (A17, 4.2);
- taxiing and parked aircraft (A17, 4.3);
- ordinary passengers and their cabin baggage (A17, 4.4);
- hold baggage checked-in or taken in custody of airline operators (A17, 4.5);
- cargo, mail, etc, intended for carriage on commercial flights (A17, 4.6);
- special categories of passengers like armed personnel or potentially disruptive passengers that have to travel on commercial flights (A17, 4.7).

At the lower levels of the regulatory hierarchy, the security measures are refined and detailed in such a way as to preserve the decomposition presented above. This structure allowed us to easily identify the relation between the different levels of refinement. Due to the restricted access nature of some of the regulatory documents, the formalization presented in Section 4 only considers Chapter 4 of Annex 17 and some of the refinements proposed by the European Doc 2320. Moreover, for simplification reasons, we do not cover the security measures 4.3 and 4.6.

[3] Note that the interpretation given to the quoted paragraph may appear wrong to some readers. In fact, Paragraph 4.1 is ambiguous as it can be interpreted in two different ways (see Section 4.4 for more details).

3.2 Modeling challenges

Modeling the regulations governing airport security is a real world problem and is therefore a good exercise to identify the limits of the inherent features of the Focal environment. Moreover, the ultimate objective of such an application is not to produce certified code but rather to provide an automated support for the analysis of the regulatory documents. For this case study, the formalization needs to address the following modeling challenges:

1. the model has to impose a structure that facilitates the traceability and maintainability of the normative documents. Moreover, through this structure, it should be possible to easily identify the impact of a particular security measure on the entire regulatory system;
2. the model must make the distinction between the security measures and the ways and means of implementing them. Most of the security measures are fairly general and correspond to reachable objectives. However, their implementation may differ from one airport to another due to national laws and local specificities;
3. for each level of the regulatory hierarchy, the model must determine (through the use of automated reasoning support tools) whether or not the fundamental security properties can be derived from the set of prescribed security measures. This will help to identity hidden assumptions made during the drafting process. In addition, the model has to provide evidence that the security measures defined at refined levels are not less restrictive than those at higher levels.

4 Formalization

4.1 Model domain

In order to formalize the meaning of the preventive security measures properly, we first need to identity the subjects they regulate, together with their respective properties/attributes and the relationships between them. It is also essential to determine the hierarchical organization of the identified subjects in order to effectively factorize functions and properties during the formalization process. This is done by determining the dependencies between the security measures, w.r.t. the definitions of terms used in the corresponding normative document. For example, let us consider the following security measure described in Chapter 4 of Annex 17:

> *4.4.1 Each Contracting State shall establish measures to ensure that originating passengers of commercial air transport operations and their cabin baggage are screened prior to boarding an aircraft departing from a security restricted area.*

To be able to formalize this security measure, it is obvious that we will have to define the subjects *originating passenger*, *cabin baggage*, *aircraft* and *security restricted area*, together with the relations between them. Moreover, we will need to define appropriate attributes for the *originating passenger* subject to characterize the state of being *screened* and of being *on board*. Finally, to complete the formalization, we will have to specify the integrity constraints induced by the regulation (e.g. screened passengers are either in security restricted areas or on board aircraft). The hierarchies of subjects obtained after analyzing all the preventive security measures of Annex 17 are represented by a Focal model, where each subject is a species. For instance, the Focal model for airside persons is given in Figure 1 (where nodes are species and arrows inheritance relations s.t. $A \leftarrow B$ means species B inherits from A).

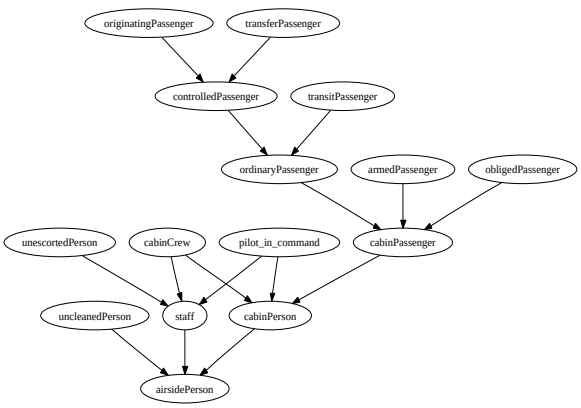

Fig. 1. Hierarchy for airside persons in Annex 17.

For possible extensions during the refinement process, the *representation* of the species is left undefined (abstract) and their functions are only declared. Moreover, since we are not concerned with code generation, our formalization does not make use of collections. For example, the following species corresponds to the specification of the *cabin person* subject:

```
species cabinPerson (obj is object, obj_set is basic_set (obj),
        do is dangerousObject, do_set is basic_set (do),
        wp is weapon, wp_set is basic_set (wp), id is identity,
        c_luggage is cabinLuggage (obj, obj_set, do, do_set, wp, wp_set),
        cl_set is basic_lset (obj, obj_set, do, do_set, wp, wp_set, c_luggage))
    inherits airsidePerson (obj, obj_set, do, do_set, wp, wp_set, id) =

sig embarked in self -> bool;
sig get_cabinLuggage in self -> cl_set;

property invariant_weapons : all w in wp, all s in self,
  wp_set!member (w, !get_weapons (s)) -> not (wp!inaccessible (w));

end
```

The species cabinPerson specifies the common functions and properties for the different types of persons who are eligible to travel on board an aircraft. In order to specify the relations between cabin persons and the different items they can have access to during flight time, the species cabinPerson is parameterized with the species object, dangerousObject, weapon and cabinLuggage. The parameters obj_set, do_set, wp_set and cl_set describe the sets of the previously

identified items; they are introduced to express the fact that a cabin person can own more than one item at a time. Since most of these relations are already specified in the species airsidePerson, they are inherited automatically. The function get_cabinLuggage is only introduced to make accessible the set of cabin luggage associated to a given instance of cabinPerson. Property invariant_weapons is a typical example of integrity constraints imposed by the regulation. It states that when weapons are carried by cabin persons, they are *by default* considered to be accessible during flight time.

4.2 Annex 17: preventive security measures

As said in Section 3.2, the formal model needs to impose a certain structure that will facilitate the traceability and maintainability of the normative documents. To achieve this purpose, our model follows the structural decomposition proposed in Chapter 4 of Annex 17 (using inheritance), while taking into account the dependencies between the preventive security measures. In our model, since most of the security measures correspond to reachable objectives, they are defined as invariants and each airport security programme must provide procedures which satisfy these invariants. However, when the security measures describe actions to be taken when safety properties are violated, a procedural approach is adopted. The consistency and completeness of the regulation are achieved by establishing that the fundamental security property, defined in Paragraph 4.1 of Annex 17, is satisfied by all the security measures, while ensuring their homogeneity. The general structure of the Annex 17 model is represented in Figure 2.

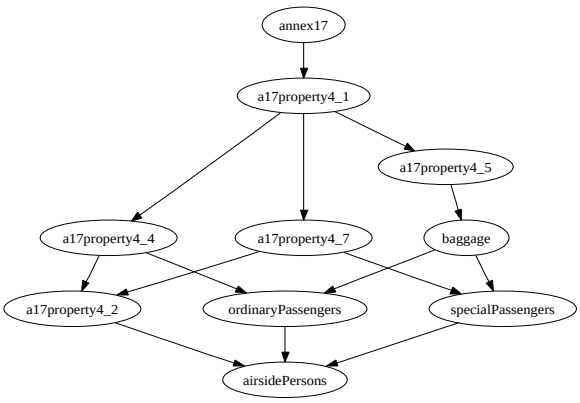

Fig. 2. Structure of Annex 17.

The species airsidePersons, ordinaryPassengers, specialPassengers and baggage introduce the set domain of the subjects presented in Section 4.1 as well as their relational constraints (e.g. two passengers cannot have the same luggage). The preventive security measures are formalized in species a17property4_2, a17property4_4, a17property4_5, a17property4_7 and their dependencies are defined according to the hierarchical organization of the subjects they regulate. The fundamental security property is defined in species a17property4_1. It is at this level that the set of on board objects is defined. Finally, the theorems establishing the consistency and completeness of the regulation are defined in the species annex17.

Security measures related to ordinary passengers As an example, we can focus on Property 4.4 of Annex 17 related to security measures for ordinary passengers. This property is divided into four sub-properties and, for example, we can describe how Property 4.4.1 (cited in Section 4.1) was formalized:

Example 3 (Property 4.4.1). Security measure 4.4.1 states that originating passengers and their cabin baggage should be screened prior to boarding an aircraft. In species a17property4_4, this statement is formalized as follows:

```
property property_4_4_1 : all p in op, all s in self,
  op_set!member (p, !originatingPassengers (s)) ->
  op!embarked (p) -> op!screened (p);
```

where p represents an originating passenger and s the current state of species a17property4_4. It should be noted that the scope of the boolean function screened extends to cabin baggage as well, since cabin baggage remains with its owners throughout the boarding process. The fact of being a screened ordinary passenger is defined in the species controlledPassengers (see Figure 1) as follows:

```
property invariant_screened : all s in self,
  !screened (s) -> wp_set!is_empty (!get_weapons (s)) and
  wp_set!is_empty (cl_set!get_weapons (!get_cabinLuggage (s))) and
  all o in do, do_set!member (o, !get_dangerousObjects (s)) or
    do_set!member (o, cl_set!get_dangerousObjects
                        (!get_cabinLuggage (s))) -> do!is_authorized (o);
```

where s represents a controlledPassenger. Property invariant_screened states that if a passenger is screened, he/she does not have any weapons and if the passenger does have a dangerous object (other than weapons), it is authorized. A similar property also exists for Property 4.4.2 (which concerns transfer passengers) and could be factorized via the parameterization mechanism of Focal.

From this property and the three others (4.4.2, 4.4.3 and 4.4.4), we can prove the global property 4.4 that ordinary passengers admitted on board an aircraft do not have any unauthorized dangerous objects. This intermediate lemma is used afterwards when proving the consistency of the fundamental security property (4.1) w.r.t. the preventive security measures.

Consistency of Annex 17 Once we completed the formalization for each of the different categories of preventive security measures and derived the appropriate intermediate lemmas, we can consider Paragraph 4.1 (see Section 3.1) of Annex 17. It is formalized as follows in species a17property4_1:

```
property property_4_1 : all a in ac, all s in self,
  ac_set!member (a, !departureAircraft (s)) ->
  (all o in do, do_set!member (o, !onboardDangerousObjects (a, s)) ->
    do!is_authorized (o)) and
  (all o in wp, wp_set!member (o, !onboardWeapons (a, s)) ->
    wp!is_authorized (o));
```

where a represents an aircraft. This states that dangerous objects are admitted on board a departing aircraft only if they are authorized. In addition, the set of on board objects for each departing aircraft is defined according to the different types of cabin persons (together with their cabin luggage) and according to the different types of hold baggage loaded into the aircraft. This correlation is necessary since it will allow us to establish the following consistency theorem:

```
theorem consistency : !property_4_2 -> !property_4_4 ->
  !property_4_5 -> !property_4_7 -> !property_4_1
proof : by do_set!union1, wp_set!union1 def !property_4_2, !property_4_4,
  !property_4_5, !property_4_7, !property_4_1;
```

where property_4_2, property_4_4, property_4_5 and property_4_7 correspond to the intermediate lemmas defined for each category of preventive security measures. The purpose of Theorem consistency is to verify whether the fundamental security property can be derived from the set of preventive security measures. This allowed us to identify some hidden assumptions done during the drafting process (see Section 4.4). However, this theorem does not guarantee the absence of contradictions in the regulation. A way to tackle this problem 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.

4.3 Doc 2320: some refinements

The document structure of Doc 2320 follows the decomposition presented in Chapter 4 of Annex 17. Refinement in Doc 2320 appears at two levels. At the subject level, the refinement consists in enriching the characteristics of the existing subjects or in adding new subjects. At the security property level, the security measures become more precise and sometimes more restrictive. The verification of the consistency and completeness of Doc 2320 is performed in the same way as for Annex 17 (see the modeling described in Section 4.2). However, since Doc 2320 refines Annex 17, an additional verification is required to show that the security measures that it describes do not invalidate the ones defined in Annex 17. Thus, in addition to consistency proofs, another kind of proofs appears, that are refinement proofs. The model structure obtained for Doc 2320 is described in Figure 3 (where the existing species coming from

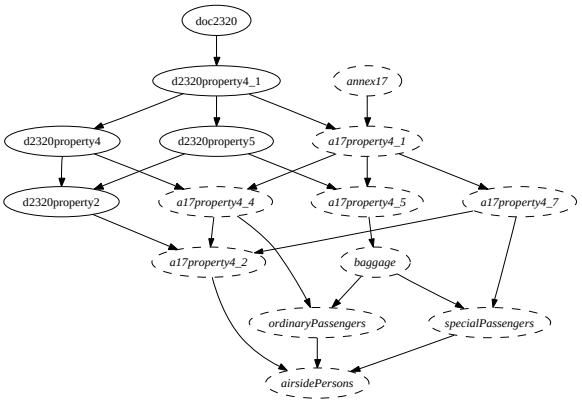

Fig. 3. Structure of Doc 2320.

Annex 17 are distinguished with dashed nodes). As can be seen, the refinement is performed in such a way as to preserve the dependencies between the security measures. Moreover, it can be observed that unlike species a17property4_2, a17property4_4 and a17property4_5, species a17property4_7 does not have a Doc 2320 counterpart. This is because, in Doc 2320, no mention to special categories of passengers is made. We assume that in this case, the international standard still prevails.

A refinement example In Doc 2320, Property 4.4.1 of Annex 17 is refined by Property 4.1.1, which states that originating passengers are either searched by hand or screened prior to boarding an aircraft. In species d2320property4, this statement is formulated as follows:

```
property d2320property_4_1_1 : all p in op, all s in self,
  op_set!member (p, !originatingPassengers (s)) ->
  op!embarked (p) -> op!screened (p) or op!handSearched (p);
```

To prove that Property d2320property_4_1_1 does not invalidate Property property_4_4_1, the following theorem is used:

```
theorem refinement : !d2320property_4_1_1 -> !property_4_4_1
proof : by op!invariant_handSearched, op!invariant_screened
        def !d2320property_4_1_1, !property_4_4_1;
```

The above theorem is provable since in species controlledPassenger2320, which is a refined version of species controlledPassenger, the boolean function handSearched is characterized by the same properties than the boolean function screened (e.g. Property invariant_screened).

4.4 Analyses and results

An example of ambiguity As seen in Section 3, Paragraph 4.1 of Annex 17 is very important as it states the primary goal of the preventive security measures to be implemented by each member state. However, it appears to be ambiguous since it can be interpreted in two different ways: either dangerous objects are *never* authorized on board or they are admitted on board *only if* they are authorized. According to the ICAO, the second interpretation is the correct one as Paragraph 4.1 needs to be considered in the general context of the regulation to clarify this ambiguity.

Hidden assumptions In trying to demonstrate that Paragraph 4.1 of Annex 17 is consistent w.r.t. the set of preventive security measures, we discovered, for instance, the following hidden assumptions:

1. since disruptive passengers who are obliged to travel are generally escorted by law enforcement officers, they are considered not to have any dangerous objects in their possession;

2. unlike other passengers, transit passengers are not subjected to any specific security control but should be protected from unauthorized interference at transit spots. This implies that they are considered to be secure and hence do not have any unauthorized dangerous objects.

Development The entire formalization takes about 10000 lines of Focal code, with in particular, 150 species and 200 proofs. It took about 2 years to be completed. The development is freely available (sending a mail to the authors) and can be compiled with the latest version of Focal (0.3.1).

5 Conclusion

Summary A way to improve security is to produce high quality standards. The formal models of Annex 17 and Doc 2320 regulations, partially described in this paper, tend to bring an effective solution in the specific framework of airport security. From these formalizations, some properties could be analyzed and in particular, the notion of consistency. This paper also aims to emphasize the use of the Focal language, which provides a strongly typed and object-oriented formal development environment. The notions of inheritance and parameterization allowed us to build the specifications in an incremental and modular way. Moreover, the Zenon automated theorem prover (provided in Focal) discharged most of the proof obligations automatically and appeared to be very appropriate when dealing with abstract specifications (i.e. with no concrete representation).

Related work Currently, models of the same regulations, by D. Bert and his team, are under development using B [1] in the framework of the EDEMOI project. In the near future, it could be interesting to compare the two formal models (in Focal and B) rigorously in order to understand if and how the specification language influences the model itself. It should be noted that the same results (see Section 4.4) were obtained from this alternative formalization, since some of these results were already analyzed before the formalization itself (during the conception step). Very close to the EDEMOI project is the SAFEE project [9], funded by the 6th Framework Programme of the European Union (FP6) and which aims to use similar techniques for security but on board the aircraft. Regarding similar specifications in Focal, we must keep in mind that the compiler is rather recent (4/5 years at most) and efforts have been essentially provided, by R. Rioboo, to build a Computer Algebra library, which is currently the standard library of Focal. However, some more applicative formalizations are under development like certified implementations of security policies [4] by M. Jaume and C. Morisset.

Future work We plan to integrate a test suite into this formalization using an automatic generation procedure (working from a Focal specification) and using stubs for abstract functions (i.e. only declared). Amongst other things, this will

allow us to imagine and build attack scenarios which, at least in this context, appear to be quite interesting for official certification authorities. Such an automatic procedure is currently work in progress, by C. Dubois and M. Carlier, but is still limited (to universally quantified propositions) and needs to be extended to be applied to our development. We also plan to produce UML documents automatically generated from the Focal specifications and which is an effective solution to interact with competent organizations (ICAO, ECAC). Such a tool has been developed by J. F. Étienne but has to be completed to deal with all the features of Focal. Regarding the Focal language itself, some future evolutions might be appropriate, in particular, the notion of subtyping (there is a notion of subspecies but it does not correspond to a relation of subtyping), but which still needs to be specified in the case of properties. Also, it might be necessary to integrate temporal features in order to model behavioral properties, since in fact, our formalization, described in this paper, just shows a static view of the specified regulations.

References

1. J. R. Abrial. *The* B *Book, Assigning Programs to Meanings.* Cambridge University Press, Cambridge (UK), 1996. ISBN 0521496195.
2. The European Civil Aviation Conference. *Regulation (EC) N° 2320/2002 of the European Parliament and of the Council of 16 December 2002 establishing Common Rules in the Field of Civil Aviation Security*, December 2002.
3. C. Dubois, T. Hardin, and V. Viguié Donzeau-Gouge. Building Certified Components within Focal. In *Symposium on Trends in Functional Programming (TFP)*, volume 5, pages 33–48, Munich (Germany), November 2004. Intellect (Bristol, UK).
4. M. Jaume and C. 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.
5. R. Laleau, S. Vignes, Y. Ledru, M. Lemoine, D. Bert, V. Viguié Donzeau-Gouge, and F. Peureux. Application of Requirements Engineering Techniques to the Analysis of Civil Aviation Security Standards. In *International Workshop on Situational Requirements Engineering Processes (SREP), in conjunction with the 13th IEEE International Requirements Engineering Conference*, Paris (France), August 2005.
6. L. Lamport. How to Write a Proof. *American Mathematical Monthly*, 102(7):600–608, August 1995.
7. The International Civil Aviation Organization. *Annex 17 to the Convention on International Civil Aviation, Security - Safeguarding International Civil Aviation against Acts of Unlawful Interference, Amendement 11*, November 2005.
8. The EDEMOI project, 2003. `http://www-lsr.imag.fr/EDEMOI/`.
9. The SAFEE project, 2004. `http://www.safee.reading.ac.uk/`.
10. The Coq Development Team. Coq, *version 8.0.* INRIA, January 2006. Available at: `http://coq.inria.fr/`.
11. The Cristal Team. Objective Caml, *version 3.09.1.* INRIA, January 2006. Available at: `http://caml.inria.fr/`.
12. The Focal Development Team. Focal, *version 0.3.1.* CNAM/INRIA/LIP6, May 2005. Available at: `http://focal.inria.fr/`.

D.2 PAPER 2: EXECUTING INDUCTIVE RELATIONS

This paper is related to Chapter 2 and has been published in [49].

Extracting Purely Functional Contents from Logical Inductive Types

David Delahaye, Catherine Dubois,
and Jean-Frédéric Étienne

CEDRIC/CNAM-ENSIIE, Paris, France,
David.Delahaye@cnam.fr, dubois@ensiie.fr,
etien_je@auditeur.cnam.fr

Abstract. We propose a method to extract purely functional contents from logical inductive types in the context of the Calculus of Inductive Constructions. This method is based on a mode consistency analysis, which verifies if a computation is possible w.r.t. the selected inputs/outputs, and the code generation itself. We prove that this extraction is sound w.r.t. the Calculus of Inductive Constructions. Finally, we present some optimizations, as well as the implementation designed in the Coq proof assistant framework.

1 Introduction

The main idea underlying extraction in proof assistants like Isabelle or Coq is to automatically produce certified programs in a correct by construction manner. More formally it means that the extracted program realizes its specification. Programs are extracted from types, functions and proofs. Roughly speaking, the extracted program only contains the computational parts of the initial specification, whereas the logical parts are skipped. In Coq, this is done by analyzing the types: types in the sort *Set* (or *Type*) are computationally relevant while types in sort *Prop* are not. Consequently, inductively defined relations, implemented as Coq logical inductive types, are not considered by the extraction process because they are exclusively dedicated to logical aspects. However such constructs are widely used to describe algorithms. For example, when defining the semantics of a programming language, the evaluation relation embeds the definition of an interpreter.

Although inductive relations are not executable, they are often preferred because it is often easier to define a relational specification than its corresponding functional counterpart involving pattern-matching and recursion. For example, in Coq, it is easier to define the relation "the terms t and u unify with s as a most general unifier" than defining the function that computes the most general unifier of t and u if it exists. In this case, the difficulty is to prove the termination of the function while simultaneously defining it. Moreover, proof assistants offer many tools to reason about relational specifications (e.g. elimination, inversion tactics) and the developer may prefer the relational style rather than the functional style,

even though recent work (e.g. in Coq, functional induction or recursive definition) provide a better support for defining functions and reasoning about them.

Based on these observations, our aim is to translate logical inductive specifications into functional code, with an intention close to the one found in the Centaur [3] project, that is extracting tools from specifications. Another motivation for extracting code from logical inductive specifications is to get means to execute these specifications (if executable), in order to validate or test them. Better still, in a formal testing framework, the extracted code could be used as an oracle to test a program independently written from the specification.

In this paper, we propose a mechanism to extract functional programs from logical inductive types. Related work has been done on this subject around semantics of programming languages, for example [3, 8, 1, 4, 12] and [2] in a more general setting. Like [2], our work goes beyond semantic applications, even though it is a typical domain of applications for such a work. In addition, our extraction is intended to only deal with logical inductive types that can be turned into purely functional programs. This is mainly motivated by the fact that proof assistants providing an extraction mechanism generally produce code in a functional framework. Thus, we do not want to manage logical inductive specifications that would require backtracking, that is more a Prolog-like paradigm. In that sense, our work separates from Centaur [3], Petterson's RML translator [8] and Berghofer and Nipkow's approach [2]. The first one translates Typol specifications, which are inductively defined semantic relations, into Prolog programs. RML is also a formalism to describe natural semantics of programming languages, and the corresponding compiler produces C programs that may backtrack if necessary. Finally, Berghofer and Nipkow's tool produces code that can compute more than one solution, if any. We can also mention the use of Maude [12] to animate executable operational semantic specifications, where these specifications are turned into rewriting systems.

To turn an inductive relation into a function that can compute some results, we need additional information. In particular, we need to know which arguments are inputs and which arguments are outputs. This information is provided by the user using the notion of modes. Furthermore, these modes are used to determine if a functional computation is possible, in which case we say that the mode is consistent. Otherwise, the functional extraction is not possible and is rejected by our method. In order to make functional computations possible, some premises in the types of constructors may have to be reordered. The notion of mode, going back actually to attribute grammars [1], is fairly standard, especially in the logical programming community. For example, the logical and functional language Mercury [7] requires mode declarations to produce efficient code. Similar mode systems have already been described in [4, 2, 9].

The paper is organized as follows: first, we informally present our extraction mechanism with the help of some examples; next, we formalize the extraction method itself (in particular, the mode consistency analysis and the code generation) and prove its soundness; finally, we describe our prototype developed in the Coq proof assistant framework and discuss some optimizations.

2 Informal presentation

In this section, we present how our functional extraction must work on some examples. For these examples, we use the Coq framework with, in particular, its syntax and OCaml [11] as one of its target languages for extraction. Our general approach is the following:

1. the user annotates his/her logical inductive type with a mode that specifies which arguments are inputs, the others being considered as outputs;
2. a mode consistency analysis is performed to determine if the extraction is possible w.r.t. the provided mode;
3. if the previous analysis is successful, the logical definition is translated into a functional program.

This process may be recursive and may call the regular extraction mechanism to extract code from functions or proofs.

A mode can be seen as the computational behavior of a logical inductive type. It is defined as a set of indices denoting the inputs of the relation. For example, let us consider the predicate add that specifies the addition of two natural numbers, i.e. given three natural numbers n, m and p, (add n m p) defines that p is the result of the addition of n and m. This predicate is defined as follows:

```
Inductive add : nat → nat → nat → Prop :=
  | addO : forall n, add n O n
  | addS : forall n m p, add n m p → add n (S m) (S p).
```

The mode $\{1, 2\}$ indicates that we consider n and m as inputs and we would like to compute p. The extracted function is the expected one, defined by pattern-matching on both arguments (actually only the second one is significant):

```
let rec add p0 p1 = match p0, p1 with
  | n, O → n
  | n, S m → let p = add n m in S p
  | _ → assert false
```

We can also propose to extract a function with the mode $\{2, 3\}$. Thus, we obtain a function that performs subtraction:

```
let rec add p0 p1 = match p0, p1 with
  | n, O → n
  | S p, S m → add p m
  | _ → assert false
```

Finally the mode $\{1, 2, 3\}$ means that the three arguments are known and that we want to produce a function that checks if the triple constitutes a possible computation or not (as a boolean result):

```
let rec add p0 p1 p2 = match p0, p1, p2 with
  | n, O, m when n = m → true
  | n, S m, S p → add n m p
  | _ → false
```

However, with the mode $\{1, 3\}$, the extraction is refused, not because of the mode analysis (which succeeds) but because it would produce a function with two overlapping patterns, (n,n) (obtained from the type of the first constructor addO) and (n,p) (obtained from the type of the second constructor addS). With such a configuration, more than one result might be computed and therefore the function would not be deterministic, which is incompatible with a proper notion of function. Extraction with modes involving only one input are refused for the same reason.

As a last example, let us consider a case where constraints are put on the results, e.g. in the eval_plus constructor the evaluation of a1 and a2 must map to values built from the N constructor:

```
Inductive Val : Set := N : Z → Val | ...
Inductive Expr : Set := V : Var → Expr | Plus : Expr → Expr → Expr.

Inductive eval : Sigma → Expr → Val → Prop :=
  | eval_v : forall (s : Sigma) (v : Var), eval s (V v) (valof s v)
  | eval_plus : forall (s : Sigma) (a1 a2 : Expr) (v w : Z),
      eval s a1 (N v) → eval s a2 (N w) → eval s (Plus a1 a2) (N (v + w)).
```

where Sigma is an evaluation context and valof is a function looking for the value of a variable in an evaluation context.

With the mode $\{1, 2\}$, the extracted function is the following:

```
let rec eval s e = match s, e with
  | s, V v → valof s v
  | s, Plus (a1, a2) →
    (match eval s a1 with
       | N v →
         (match eval s a2 with
            | N w → N (zplus v w)
            | _ → assert false)
       |_ → assert false)
```

where valof and zplus are respectively the extracted functions from the definitions of valof and the addition over Z.

Regarding mode consistency analysis, detailed examples will be given in the next section.

3 Extraction of logical inductive types

The extraction is made in two steps: first, the mode consistency analysis tries to find a permutation of the premises of each inductive clause, which is compatible w.r.t. the annotated mode; second, if the mode consistency analysis has been successful, the code generation produces the executable functional program. Before describing the extraction method itself, we have to specify which inductive types we consider and in particular, what we mean exactly by logical inductive types. We must also precise which restrictions we impose, either to ensure a purely functional and meaningful extraction, or to simplify the presentation of

this formalization, while our implementation relaxes some of these restrictions (see Section 5).

3.1 Logical inductive types

The type theory we consider is the Calculus of Inductive Constructions (CIC for short; see the documentation of Coq [10] to get some references regarding the CIC), i.e. the Calculus of Constructions with inductive definitions. This theory is probably too strong for what we want to show in this paper, but it is the underlying theory of the Coq proof assistant, in which we chose to develop the corresponding implementation. An inductive definition is noted as follows (inspired by the notation used in the documentation of Coq):

$\mathsf{Ind}(d : \tau, \Gamma_c)$

where d is the name of the inductive definition, τ a type and Γ_c the context representing the constructors (their names together with their respective types). In this notation, two restrictions have been made: we do not deal with parameters[1] and mutual inductive definitions. Actually, these features do not involve specific technical difficulties. Omitting them allows us to greatly simplify the presentation of the extraction, as well as the soundness proof in particular. Also for simplification reasons, dependencies, higher order and propositional arguments are not allowed in the type of an inductive definition; more precisely, this means that τ has the following form:

$\tau_1 \to \ldots \tau_n \to \mathsf{Prop}$

where τ_i is of type Set or Type, and does not contain any product or dependent inductive type. In addition, we suppose that the types of constructors are in prenex form, with no dependency between the bounded variables and no higher order; thus, the type of a constructor is as follows:

$\prod_{i=1}^{n} x_i : X_i.T_1 \to \ldots \to T_m \to (d\ t_1\ \ldots\ t_p)$

where $x_i \notin X_j$, X_i is of type Set or Type, T_i is of type Prop and does not contain any product or dependent inductive type, and t_i are terms. In the following, the terms T_i will be called the premises of the constructor, whereas the term $(d\ t_1\ \ldots\ t_p)$ will be called the conclusion of the constructor. We impose the additional constraint that T_i is a fully applied logical inductive type, i.e. T_i has the following form:

$d_i\ t_{i1}\ \ldots\ t_{ip_i}$

[1] In CIC, parameters are additional arguments, which are shared by the type of the inductive definition (type τ) and the types of constructors (defined in Γ_c).

where d_i is a logical inductive type, t_{ij} are terms, and p_i is the arity of d_i.

An inductive type verifying the conditions above is called a logical inductive type. We aim to propose an extraction method for this kind of inductive types.

3.2 Mode consistency analysis

The purpose of the mode consistency analysis is to check whether a functional execution is possible. It is a very simple data-flow analysis of the logical inductive type. We require the user to provide a mode for the considered logical inductive type, and recursively for each logical inductive type occurring in this type.

Given a logical inductive type I, a mode md is defined as a set of indices denoting the inputs of I. The remaining arguments are the output arguments. Thus, $md \subseteq \{1, \ldots, a_I\}$, where a_I is the arity of I. Although a mode is defined as a set, the order of the inputs in the logical inductive type is relevant. They will appear in the functional translation in the same order.

In practice, it is often the case to use the extraction with, either a mode corresponding to all the arguments except one, called the computational mode, or a mode indicating that all the arguments are inputs, called the fully instantiated mode. The formalization below will essentially deal with these two modes with no loss of generality (if more than one output is necessary, we can consider that the outputs are gathered in a tuple).

In order to make functional computations possible, some premises in a constructor may have to be reordered. It will be the case when a variable appears first in a premise as an input and as an output in another premise written afterwards. For a same logical inductive type, some modes may be possible whereas some others may be considered inconsistent. Different mode declarations give different extracted functions.

Given $\mathcal{M}$, the set of modes for I and recursively for every logical inductive type occurring in I, a mode md is consistent for I w.r.t. $\mathcal{M}$ iff it is consistent for Γ_c (i.e. all the constructors of I) w.r.t. $\mathcal{M}$. A mode md is consistent for the constructor c of type $\Pi_{i=1}^{n} x_i : X_i . T_1 \rightarrow \ldots \rightarrow T_n \rightarrow T$ w.r.t. $\mathcal{M}$, where $T = d\ t_1\ \ldots\ t_p$, iff there exist a permutation π and the sets of variables S_i, with $i = 0 \ldots m$, s.t.:

1. $S_0 = \mathsf{in}(md, T)$;
2. $\mathsf{in}(m_{\pi j}, T_{\pi j}) \subseteq S_{j-1}$, with $1 \leq j \leq m$ and $m_{\pi j} = \mathcal{M}(\mathsf{name}(T_{\pi j}))$;
3. $S_j = S_{j-1} \cup \mathsf{out}(m_{\pi j}, T_{\pi j})$, with $1 \leq j \leq m$ and $m_{\pi j} = \mathcal{M}(\mathsf{name}(T_{\pi j}))$;
4. $\mathsf{out}(md, T) \subseteq S_m$.

where $\mathsf{name}(t)$ is the name of the logical inductive type applied in the term t (e.g. $\mathsf{name}(add\ n\ (S\ m)\ (S\ k)) = add$), $\mathsf{in}(m, t)$ the set of variables occurring in the terms designated as inputs by the mode m in the term t (e.g. $\mathsf{in}(\{1,2\}, (add\ n\ (S\ m)\ (S\ k))) = \{n, m\}$), and $\mathsf{out}(m, t)$ the set of variables occurring in the terms designated as outputs by the mode m in the term t (e.g. $\mathsf{out}(\{1,2\}, (add\ n\ (S\ m)\ (S\ k))) = \{k\}$).

The permutation π denotes a suitable execution order for the premises. The set S_0 denotes the initial set of known variables and S_j the set of known variables after the execution of the πj^{th} premise (when $T_{\pi 1}$, $T_{\pi 2}$, ..., $T_{\pi j}$ have been executed in this order). The first condition states that during the execution of T (for the constructor c) with the mode md, the values of all the variables used in the terms designated as inputs have to be known. The second condition requires that the execution of a premise T_i with a mode m_i will be performed only if the input arguments designated by the mode are totally computable. It also requires that m_i is a consistent mode for the logical inductive type related to T_i (we have constrained T_i in the previous section to be a fully applied logical inductive type). According to the third condition, all the arguments of a given premise T_i are known after its execution. Finally, a mode md is said to be consistent for c w.r.t. $\mathcal{M}$ if all the arguments in the conclusion of c (i.e. T) are known after the execution of all the premises.

We have imposed some restrictions on the presence of functions in the terms appearing in the type of a constructor. To relax these conditions, such as accepting functions in the output of the premises (see Section 5), in the step j, we should verify that the function calls are computable, that is to say their arguments only involve known variables (belonging to S_{j-1}).

To illustrate the mode consistency analysis, let us consider the logical inductive type that specifies the big step semantics of a small imperative language. The evaluation of commands is represented by the relation $s \vdash_c i : s'$, which means that the execution of the command i in the store s leads to the final store s'. This relation has the type $store \to command \to store \to Prop$, where $store$ and $command$ are the corresponding types for stores and imperative commands. For example, the types of the constructors for the $while$ loop are the following:

$$\begin{array}{l} while_1 : (s \vdash_e b : true) \to (s \vdash_c i : s') \to (s' \vdash_c while\ b\ do\ i : s'') \to \\ \qquad\qquad (s \vdash_c while\ b\ do\ i : s'') \\ while_2 : (s \vdash_e b : false) \to (s \vdash_c while\ b\ do\ i : s) \end{array}$$

where given a store s, an expression t and a value v, $s \vdash_e t : v$ represents the evaluation of expressions, i.e. the expression t evaluates to v in the store s. For clarity reasons, we do not indicate the universally quantified variables, which are the stores s, s' and s'', the expression b and the command i.

If the mode $\{1, 2\}$ is consistent for $\vdash_e$ then the mode $\{1, 2\}$ is consistent for both $while$ constructors of $\vdash_c$. But considering the typing relation of the simply typed λ-calculus $\Gamma \vdash t : \tau$, denoting that t is of type τ in context Γ and where Γ is a typing context, t a term and τ a type, the mode $\{1, 2\}$ is not consistent for this relation. Actually, this mode is not consistent for the typing of the abstraction; the type τ_1 in the premise, considered here as an input, is not known at this step ($\tau_1 \notin S_0$):

$$abs : (\Gamma, (x : \tau_1) \vdash e : \tau_2) \to (\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2)$$

In the following, we will ignore the permutation of the premises and will assume that all the premises are correctly ordered w.r.t. to the provided mode.

3.3 Code generation

The functional language we consider as target for our extraction is defined as follows (mainly a functional core with recursion and pattern-matching):

$$\begin{array}{rcl} e & ::= & x \mid c^n \mid C^n \mid \mathsf{fail} \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \mid e_1\ e_2 \mid \mathsf{fun}\ x \to e \\ & \mid & \mathsf{rec}\ f\ x \to e \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid (e_1, \ldots, e_n) \\ & \mid & \mathsf{match}\ e\ \mathsf{with} \mid gpat_1 \to e_1\ \ldots \mid gpat_n \to e_n \end{array}$$

$$gpat ::= pat \mid pat\ \mathsf{when}\ e$$

$$pat\ ::= x \mid C^n \mid C^n\ pat \mid (pat_1, \ldots, pat_n) \mid \_$$

where c^n is a constant of arity n, to be distinguished from C^n, a constructor of arity n. Both constants and constructors are uncurrified and fully applied. Moreover, we use the notations $e_1\ e_2\ \ldots\ e_n$ for $((e_1\ e_2) \ldots e_n)$, and $\mathsf{fun}\ x_1\ \ldots\ x_n \to e$ for $\mathsf{fun}\ x_1 \to \ldots \mathsf{fun}\ x_n \to e$ (as well as for rec functions).

Given $I = \mathsf{Ind}(d : \tau, \Gamma_c)$, a logical inductive type, well-typed in a context Γ, and $\mathcal{M}$, the set of modes for I and recursively for every logical inductive type occurring in I, we consider that each logical inductive type is ordered according to its corresponding mode (with the output, if required, at the last position), i.e. if the mode of a logical inductive type J is $\mathcal{M}(J) = \{n_1, \ldots, n_J\}$, then the n_1^{th} argument of J becomes the first one and so on until the n_J^{th} argument, which becomes the c_J^{th} one, with $c_J = \mathsf{card}(\mathcal{M}(J))$. The code generation for I, denoted by $[\![I]\!]_{\Gamma,\mathcal{M}}$, begins as follows (we do not translate the type τ of I in Γ since this information is simply skipped in the natural semantics we propose for the target functional language in Section 4):

$$[\![I]\!]_{\Gamma,\mathcal{M}} = \begin{cases} \mathsf{fun}\ p_1\ \ldots\ p_{c_I} \to [\![\Gamma_c]\!]_{\Gamma,\mathcal{M},P}, & \text{if } d \notin \Gamma_c, \\ \mathsf{rec}\ d\ p_1\ \ldots\ p_{c_I} \to [\![\Gamma_c]\!]_{\Gamma,\mathcal{M},P}, & \text{otherwise} \end{cases}$$

where $c_I = \mathsf{card}(\mathcal{M}(I))$ and $P = \{p_1, \ldots, p_{c_I}\}$.
Considering $\Gamma_c = \{c_1, \ldots, c_n\}$, the body of the function is the following:

$$\begin{array}{rcl} [\![\Gamma_c]\!]_{\Gamma,\mathcal{M},P} & = & \mathsf{match}\ (p_1, \ldots, p_{c_I})\ \mathsf{with} \\ & & \mid [\![c_1]\!]_{\Gamma,\mathcal{M}} \\ & & \mid \ldots \\ & & \mid [\![c_n]\!]_{\Gamma,\mathcal{M}} \\ & & \mid \_ \ \to\ \mathsf{default}_{I,\mathcal{M}} \end{array}$$

where $\mathsf{default}_{I,\mathcal{M}}$ is defined as follows:

$$\mathsf{default}_{I,\mathcal{M}} = \begin{cases} \mathsf{false}, & \text{if } c_I = a_I, \\ \mathsf{fail}, & \text{if } c_I = a_I - 1 \end{cases}$$

where $c_I = \mathsf{card}(\mathcal{M}(I))$ and a_I is the arity of I.

The translation of a constructor c_i is the following (the name of c_i is skipped and only its type is used in this translation):

$$[\![c_i]\!]_{\Gamma,\mathcal{M}} = [\![\prod_{j=1}^{n_i} x_{ij} : X_{ij}.T_{i1} \to \ldots \to T_{im_i} \to (d\ t_{i1}\ \ldots\ t_{ip_i})]\!]_{\Gamma,\mathcal{M}}$$
$$= \begin{cases} ([\![t_{i1}]\!], \ldots, [\![t_{ic_I}]\!]) \to [\![T_{i1} \to \ldots \to T_{im_i}]\!]_{\Gamma,\mathcal{M},\mathsf{cont}_{I,\mathcal{M}}}, \\ \quad \text{if } t_{i1}, \ldots, t_{ic_I} \text{ are linear,} \\ ([\![\sigma_i(t_{i1})]\!], \ldots, [\![\sigma_i(t_{ic_I})]\!])\ \mathsf{when}\ \mathsf{guard}(\sigma_i) \to \\ \quad [\![T_{i1} \to \ldots \to T_{im_i}]\!]_{\Gamma,\mathcal{M},\mathsf{cont}_{I,\mathcal{M}}}, \text{ otherwise} \end{cases}$$

where σ_i is a renaming s.t. $\sigma_i(t_{i1}), \ldots, \sigma_i(t_{ic_I})$ are linear, $\mathsf{guard}(\sigma_i)$ is the corresponding guard, of the form $x_{ij_1} = \sigma_i(x_{ij_1})\ \mathsf{and}\ \ldots\ \mathsf{and}\ x_{ij_k} = \sigma_i(x_{ij_k})$, with $\mathsf{dom}(\sigma_i) = \{x_{ij_1}, \ldots, x_{ij_k}\}$ and $1 \leq j_l \leq n_i$, $l = 1 \ldots k$, and $\mathsf{cont}_{I,\mathcal{M}}$ is defined as follows:

$$\mathsf{cont}_{I,\mathcal{M}} = \begin{cases} \mathsf{true}, \text{ if } c_I = a_I, \\ [\![t_{ip_i}]\!], \text{ if } c_I = a_I - 1 \end{cases}$$

The terms $t_{i1}, \ldots, t_{ic_I}$ must only contain variables and constructors, while t_{ip_i} (if $c_I = a_I - 1$) can additionally contain symbols of functions. The corresponding translation is completely isomorphic to the structure of these terms and uses the regular extraction, presented in [6]. Moreover, we consider that the terms t_{ik} and t_{jk} are not unifiable, for $i, j = 1 \ldots n$ and $k = 1 \ldots c_I$ (otherwise, a functional extraction may not be possible, i.e. backtracking is necessary or there are several results).

The right-hand side of the pattern rules is generated with the following scheme:

$$[\![T_{ij} \to \ldots \to T_{im_i}]\!]_{\Gamma,\mathcal{M},\mathsf{cont}_{I,\mathcal{M}}} =$$
$$\begin{cases} \mathsf{cont}_{I,\mathcal{M}}, \text{ if } j > m_i, \\ \\ \mathsf{if}\ [\![d_{ij}\ t_{ij1} \ldots t_{ijc_{ij}}]\!]\ \mathsf{then}\ [\![T_{i(j+1)} \to \ldots \to T_{im_i}]\!]_{\Gamma,\mathcal{M},\mathsf{cont}_{I,\mathcal{M}}} \\ \mathsf{else}\ \mathsf{default}_{I,\mathcal{M}}, \text{ if } j \leq m_i \text{ and } c_{ij} = a_{ij}, \\ \\ \mathsf{match}\ [\![d_{ij}\ t_{ij1} \ldots t_{ijc_{ij}}]\!]\ \mathsf{with} \\ \mid\ [\![t_{ija_{ij}}]\!] \to [\![T_{i(j+1)} \to \ldots \to T_{im_i}]\!]_{\Gamma,\mathcal{M},\mathsf{cont}_{I,\mathcal{M}}} \\ \mid\ \_ \ \to\ \mathsf{default}_{I,\mathcal{M}}, \text{ if } j \leq m_i \text{ and } c_{ij} = a_{ij} - 1 \end{cases}$$

where $T_{ij} = d_{ij}\ t_{ij1} \ldots t_{ija_{ij}}$, a_{ij} is the arity of d_{ij} and $c_{ij} = \mathsf{card}(\mathcal{M}(\Gamma(d_{ij})))$. We consider that the term $t_{ija_{ij}}$ is linear and does not contain computed variables or symbols of functions when $c_{ij} = a_{ij} - 1$ (in this way, no guard is required in the produced pattern).

4 Soundness of the extraction

Before proving the soundness of the extraction of logical inductive types, we need to specify the semantics of the functional language we chose as a target for our extraction and presented in Section 3. To simplify, we adopt a pure Kahn style

big step natural semantics (with call by value) and we introduce the following notion of value:

$$v ::= c^0 \mid C^0 \mid \Delta_C \mid \mathsf{fail} \mid <x \rightsquigarrow e, \Delta> \mid <x \rightsquigarrow e, \Delta>_{\mathsf{rec}(f)} \mid (v_1, \ldots, v_n)$$

where Δ is an evaluation context, i.e. a list of pairs (x, v), and Δ_C is a set of values of the form $C^n\ (v_1, \ldots, v_n)$. In addition, we introduce a set Δ_c of tuples of the form $(c^n, v_1, \ldots, v_n, v)$, with $n > 0$.

An expression e evaluates to v in an environment Δ, denoted by the judgment $\Delta \vdash e \triangleright v$, if and only if there exists a derivation of this judgment in the system of inference rules described in Appendix A (to save space, we do not include the corresponding error rules returning fail).

Given I, a logical inductive type, well-typed in a context Γ, and $\mathcal{M}$, the set of modes for I and recursively for every logical inductive type occurring in I, we introduce the translation (and evaluation) of the context Γ w.r.t. the logical inductive type I and the set of modes $\mathcal{M}$. A context is a set of assumptions $(x : \tau)$, definitions $(x : \tau := t)$ and inductive definitions $\mathsf{Ind}(d : \tau, \Gamma_c)$, where x and d are names, τ a type, t a term and Γ_c the set of constructors. The translation of Γ w.r.t. I and $\mathcal{M}$, denoted by $[\![\Gamma]\!]_{I,\mathcal{M}}$, consists in extracting and evaluating recursively each assumption, definition or inductive definition occurring in I (thus, this translation provides an evaluation context). For assumptions, definitions and inductive definitions which are not logical inductive types, we use the regular extraction, presented in [6]. Regarding logical inductive types, we apply the extraction described previously in Section 3.

The soundness of our extraction is expressed by the following theorem:

Theorem (Soundness). *Given $I = \mathsf{Ind}(d : \tau, \Gamma_c)$, a logical inductive type, well-typed in a context Γ, and $\mathcal{M}$, the set of modes for I and recursively for every logical inductive type occurring in I, we have the two following cases:*

- $c_I = a_I$*: if* $[\![\Gamma]\!]_{I,\mathcal{M}} \vdash [\![I]\!]_{\Gamma,\mathcal{M}}\ [\![t_1]\!] \ldots [\![t_{c_I}]\!] \triangleright \mathsf{true}$ *then the statement* $\Gamma \vdash d\ t_1 \ldots t_{c_I}$ *is provable;*
- $c_I = a_I - 1$*: if* $[\![\Gamma]\!]_{I,\mathcal{M}} \vdash [\![I]\!]_{\Gamma,\mathcal{M}}\ [\![t_1]\!] \ldots [\![t_{c_I}]\!] \triangleright v \neq \mathsf{fail}$ *then there exists t s.t.* $[\![t]\!] = v$ *and the statement* $\Gamma \vdash d\ t_1 \ldots t_{c_I}\ t$ *is provable.*

where $c_I = \mathsf{card}(\mathcal{M}(I))$, a_I is the arity of I, and $t_1 \ldots t_{c_I}$, t are terms.

Proof. The theorem is proved by induction over the extraction. We suppose that $\Delta \vdash [\![I]\!]_{\Gamma,\mathcal{M}}\ [\![t_1]\!] \ldots [\![t_{c_I}]\!] \triangleright v$, with $\Delta = [\![\Gamma]\!]_{I,\mathcal{M}}$ and either $v = \mathsf{true}$ if $c_I = a_I$, or $v \neq \mathsf{fail}$ if $c_I = a_I - 1$. Using the definition of $[\![I]\!]_{\Gamma,\mathcal{M}}$ given in Section 3 and the rules of Appendix A, this expression is evaluated as follows:

$$\Delta \vdash [\![I]\!]_{\Gamma,\mathcal{M}} \triangleright \begin{cases} <p_1\ \ldots\ p_{c_I} \rightsquigarrow [\![\Gamma_c]\!]_{\Gamma,\mathcal{M},P}, \Delta>, \text{ if } d \notin \Gamma_c, \\ <p_1\ \ldots\ p_{c_I} \rightsquigarrow [\![\Gamma_c]\!]_{\Gamma,\mathcal{M},P}, \Delta>_{\mathsf{rec}(d)} = c, \text{ otherwise} \end{cases}$$

where $P = \{p_1, \ldots, p_{c_I}\}$.

The arguments are also evaluated: $\Delta \vdash [\![t_1]\!] \triangleright v_1, \ldots,\ \Delta \vdash [\![t_{c_I}]\!] \triangleright v_{c_I}$. Using the definition of $[\![\Gamma_c]\!]_{\Gamma,\mathcal{M},P}$, we have the following evaluation:

$$\Delta_b \vdash \begin{pmatrix} \mathsf{match}\ (v_1, \ldots, v_{c_I})\ \mathsf{with} \\ \mid [\![c_1]\!]_{\Gamma,\mathcal{M}} \\ \mid \ldots \\ \mid [\![c_n]\!]_{\Gamma,\mathcal{M}} \\ \mid \_ \ \rightarrow\ \mathsf{default}_{I,\mathcal{M}} \end{pmatrix} \triangleright v$$

where Δ_b is defined as follows:

$$\Delta_b = \begin{cases} \Delta, (p_1, v_1), \ldots, (p_{c_I}, v_{c_I}), \text{ if } d \notin \Gamma_c, \\ \Delta, (d, c), (p_1, v_1), \ldots, (p_{c_I}, v_{c_I}), \text{ otherwise} \end{cases}$$

We know that either $v = \mathsf{true}$ or $v \neq \mathsf{fail}$ (according to c_I); this means that there exists i s.t. the pattern of $[\![c_i]\!]_{\Gamma,\mathcal{M}}$ matches the value $(v_1, \ldots, v_{c_I})$. Using the definition of $[\![c_i]\!]_{\Gamma,\mathcal{M}}$, we have to evaluate:

$$\Delta_p \vdash [\![T_{i1} \rightarrow \ \ldots\ \rightarrow T_{im_i}]\!]_{\Gamma,\mathcal{M},\mathsf{cont}_{I,\mathcal{M}}} \triangleright v \tag{1}$$

with $\Delta_p = \Delta_b, \Delta_i$, where Δ_i has the following form (by definition of $\mathsf{filter}_{\Delta_b}$):

$$\Delta_i = \begin{cases} \mathsf{mgu}_{\Delta_b}(v_t, ([\![t_{i1}]\!], \ldots, [\![t_{ic_I}]\!])), \text{ if } t_{i1}, \ldots, t_{ic_I} \text{ are linear} \\ \mathsf{mgu}_{\Delta_b}(v_t, ([\![\sigma_i(t_{i1})]\!], \ldots, [\![\sigma_i(t_{ic_I})]\!])), \text{ otherwise} \end{cases}$$

where $v_t = (v_1, \ldots, v_{c_I})$. In addition, we have $\Delta_p \vdash \mathsf{guard}(\sigma_i) \triangleright \mathsf{true}$ if $t_{i1}, \ldots, t_{ic_I}$ are not linear.

The reduction of our extraction language is weaker than the one defined for CIC; in particular, this means that we have: given $\Delta = [\![\Gamma]\!]_{I,\mathcal{M}}$, a term t and a value $v \neq \mathsf{fail}$, if $\Delta \vdash [\![t]\!] \triangleright v$ then there exists a term t' s.t. $[\![t']\!] = v$ and $\Gamma \vdash t \equiv t'$, where $\equiv$ is the convertibility relation for CIC. Moreover, considering $\Delta = [\![\Gamma]\!]_{I,\mathcal{M}}$, a term t and a value $v \neq \mathsf{fail}$, if $\sigma = \mathsf{mgu}_{\Delta}(v, [\![t]\!])$ then there exist t' and $\bar{\sigma}$ s.t. $[\![t']\!] = v$, $\mathsf{dom}(\bar{\sigma}) = \mathsf{dom}(\sigma)$, $[\![\bar{\sigma}(x)]\!] = \sigma(x)$ for all $x \in \mathsf{dom}(\bar{\sigma})$, and $\bar{\sigma} = \mathsf{mgu}_{\Gamma}(t', t)$. Using these two remarks, there exists $\bar{\Delta}_i$ as described above s.t.:

$$\Gamma \vdash \bar{\Delta}_i(d\ t_{i1} \ldots t_{ic_I}) \equiv (d\ t_1 \ldots t_{c_I}) \tag{2}$$

Note that we can consider $\Delta_b = \Delta$, since the variables p_i, $i = 1 \ldots c_I$, do not occur in I; actually, these variables are just used for the curryfication of $[\![I]\!]_{\Gamma,\mathcal{M}}$. Note also that this unification between $(d\ t_{i1} \ldots t_{ic_I})$ and $(d\ t_1 \ldots t_{c_I})$ may be total or partial according to c_I (if $c_I = a_I$ or $c_I = a_I - 1$).

Regarding the arguments of c_i, we have to consider another property: given a context Δ_a, if $\Delta_p, \Delta_a \vdash [\![T_{i1} \rightarrow \ \ldots\ \rightarrow T_{im_i}]\!]_{\Gamma,\mathcal{M},\mathsf{cont}_{I,\mathcal{M}}} \triangleright v$ then there exists a context $\Delta'_a \supseteq \Delta_a$ s.t. $\Gamma \vdash \bar{\Delta}'_a \bar{\Delta}_i T_{ij}$ is provable for $j = 1 \ldots m_i$, and $\Delta_p, \Delta'_a \vdash \mathsf{cont}_{I,\mathcal{M}} \triangleright v$. This property is proved by induction over the product type. Using the definition of $[\![T_{i1} \rightarrow \ \ldots\ \rightarrow T_{im_i}]\!]_{\Gamma,\mathcal{M}}$, we have three cases:

- $j > m_i$: $\Delta_p, \Delta_a \vdash \mathsf{cont}_{I,\mathcal{M}} \triangleright v$ and $\Delta'_a = \Delta$.

– $j \leq m_i$, $c_{ij} = a_{ij}$:

$$\Delta_p, \Delta_a \vdash \begin{pmatrix} \mathsf{if}\ [\![d_{ij}\ t_{ij1} \ldots t_{ijc_{ij}}]\!]\ \mathsf{then} \\ \quad [\![T_{i(j+1)} \to \ldots \to T_{im_i}]\!]_{\Gamma,\mathcal{M},\mathsf{cont}_{I,\mathcal{M}}} \\ \mathsf{else}\ \mathsf{default}_{I,\mathcal{M}} \end{pmatrix} \triangleright v$$

Since either $v = \mathsf{true}$ or $v \neq \mathsf{fail}$, we have $\Delta_p, \Delta_a \vdash [\![d_{ij}\ t_{ij1} \ldots t_{ijc_{ij}}]\!] \triangleright \mathsf{true}$ and the then branch is selected. By hypothesis of induction (over the soundness theorem), this means that $\Gamma \vdash \bar{\Delta}_a \bar{\Delta}_i T_{ij}$ is provable. Next, we have the evaluation $\Delta_p, \Delta_a \vdash [\![T_{i(j+1)} \to \ldots \to T_{im_i}]\!]_{\Gamma,\mathcal{M},\mathsf{cont}_{I,\mathcal{M}}} \triangleright v$ and by hypothesis of induction, there exists $\Delta'_a \supseteq \Delta_a$ s.t. $\Gamma \vdash \bar{\Delta}'_a \bar{\Delta}_i T_{ik}$ is provable for $k = j+1 \ldots m_i$, and $\Delta_p, \Delta'_a \vdash \mathsf{cont}_{I,\mathcal{M}} \triangleright v$. As $\Delta'_a \supseteq \Delta_a$, $\Gamma \vdash \bar{\Delta}'_a \bar{\Delta}_i T_{ij}$ is also provable.

– $j \leq m_i$, $c_{ij} = a_{ij} - 1$:

$$\Delta_p, \Delta_a \vdash \begin{pmatrix} \mathsf{match}\ [\![d_{ij}\ t_{ij1} \ldots t_{ijc_{ij}}]\!]\ \mathsf{with} \\ \mid [\![t_{ija_{ij}}]\!] \to [\![T_{i(j+1)} \to \ldots \to T_{im_i}]\!]_{\Gamma,\mathcal{M},\mathsf{cont}_{I,\mathcal{M}}} \\ \mid \_ \ \to\ \mathsf{default}_{I,\mathcal{M}} \end{pmatrix} \triangleright v$$

Since either $v = \mathsf{true}$ or $v \neq \mathsf{fail}$, we have $\Delta_p, \Delta_a \vdash [\![d_{ij}\ t_{ij1} \ldots t_{ijc_{ij}}]\!] \triangleright v' \neq \mathsf{fail}$ and the pattern $[\![t_{ija_{ij}}]\!]$ matches v'. By hypothesis of induction (over the soundness theorem), this means that $\Gamma \vdash \bar{\Delta}_a \bar{\Delta}_i T_{ij}$ is provable. Next, we have the evaluation $\Delta_p, \Delta'_p \vdash [\![T_{i(j+1)} \to \ldots \to T_{im_i}]\!]_{\Gamma,\mathcal{M},\mathsf{cont}_{I,\mathcal{M}}} \triangleright v$, with $\Delta'_p = \Delta_a, \Delta_m$ and $\Delta_m = \mathsf{mgu}_{\Delta_p,\Delta_a}(v', [\![t_{ija_{ij}}]\!])$. By hypothesis of induction, there exists $\Delta'_a \supseteq \Delta'_p$ s.t. $\Gamma \vdash \bar{\Delta}'_a \bar{\Delta}_i T_{ik}$ is provable for $k = j+1 \ldots m_i$, and $\Delta_p, \Delta'_a \vdash \mathsf{cont}_{I,\mathcal{M}} \triangleright v$. As $\Delta'_a \supseteq \Delta'_p$, $\Gamma \vdash \bar{\Delta}'_a \bar{\Delta}_i T_{ij}$ is also provable.

Using (1), (2) and the above property (with the empty context for Δ_a), there exists a context Δ'_i s.t. $\Gamma \vdash \bar{\Delta}'_i \bar{\Delta}_i T_{ij}$ is provable for $j = 1 \ldots m_i$, $\Gamma \vdash \bar{\Delta}'_i \bar{\Delta}_i (d\ t_{i1} \ldots t_{ic_I}) \equiv (d\ t_1 \ldots t_{c_I})$ and $\Delta_p, \Delta'_i \vdash \mathsf{cont}_{I,\mathcal{M}} \triangleright v$. We distinguish two cases:

– $c_I = a_I$: $\mathsf{cont}_{I,\mathcal{M}} = \mathsf{true}$ and the constructor c_i can be directly applied to prove $\Gamma \vdash d\ t_1 \ldots t_{c_I}$.

– $c_I = a_I - 1$: we have $\Delta_p, \Delta'_i \vdash [\![t_{ip_i}]\!] \triangleright v$. There exists a term t'_{ip_i} s.t. $[\![t'_{ip_i}]\!] = v$ and $\Gamma \vdash \bar{\Delta}'_i \bar{\Delta}_i t_{ip_i} \equiv t'_{ip_i}$. Thus, $\Gamma \vdash \bar{\Delta}'_i \bar{\Delta}_i (d\ t_{i1} \ldots t_{ic_I}\ t_{ip_i}) \equiv (d\ t_1 \ldots t_{c_I}\ t'_{ip_i})$ and the constructor c_i can be applied to prove $\Gamma \vdash d\ t_1 \ldots t_{c_I}\ t'_{ip_i}$.

5 Implementation and optimizations

5.1 Implementation

A prototype has been implemented and integrated to the Coq [10] proof assistant. This implementation follows what has been described in Section 3 and is plugged into the regular extraction mechanism of Coq so that it can produce code for the

same (functional) languages, i.e. OCaml, Haskell and Scheme. As an example, let us consider the addition predicate presented in Section 2 with an extraction to the OCaml language [11]:

```
Coq < Inductive add : nat → nat → nat → Prop :=
Coq <   | addO : forall n, add n O n
Coq <   | addS : forall n m p, add n m p → add n (S m) (S p).

Coq < Extraction Logical add [1 2].
(** val add : nat → nat → nat **)

let rec add p0 p1 =
  match (p0, p1) with
    | (n, O) → n
    | (n, S m) → let p = add n m in S p
    | _ → assert false
```

This implementation should be part of the forthcoming version of Coq, and currently, the development in progress is available on demand (sending a mail to the authors).

5.2 Optimizations

Conclusion inputs In Section 3, we described the translation of a logical inductive type when rules do not overlap, that is when the types of the conclusions do not unify. However, we can implement some heuristics to overcome some of these cases. Let us consider the example of the while loop, seen in Section 3:

```
Inductive exec : store → command → store → Prop := ...
  | while1 : forall (s s1 s2 : Sigma) (b : expr) (c : command),
      (eval s b true) → (exec s c s1) → (exec s1 (while b do c) s2) →
      (exec s (while b do c) s2)
  | while2 : forall (s : Sigma) (b : expr) (c : command), (eval s b false) →
      (exec s (while b do c) s).
```

These two constructors overlap: the types of their conclusion are identical up to renaming. A Prolog-like execution would try to apply the first rule by computing the evaluation of the boolean expression b and matching it with the value true. If the matching fails, the execution would backtrack to the second rule. However, in this case, the execution is completely deterministic and no backtracking is necessary. In fact, we can discriminate the choice between both constructors thanks to their first premise. We introduce a heuristic to handle such cases efficiently. It requires the ability to detect common premises between both overlapping rules and to discriminate w.r.t. syntactic exclusive premises (p and $\neg p$, values constructed with different constructors of an inductive type, for example).

For the while loop example, the extracted function with the mode $\{1, 2\}$ involves only one case in the global pattern-matching to be able to handle correctly the execution:

```
let rec exec s c = match s, c with ...
  | s, while(b,c) →
    (match (eval s b) with
       | true → s
       | false →
         let s1 = exec s c in
         let s2 = exec s1 (while (b, c)) in s2)
```

Premise outputs In the formalization, we also assumed that the outputs of the premises do not contain computed variables. Consequently, we cannot translate rules where constraints exist on these outputs, which is the case for the following constructor that describes the typing of a conditional expression in a logical inductive type named typecheck, when the mode $\{1, 2\}$ is specified:

```
Inductive typecheck : env → expr → type → Prop := ...
  | if : forall (g : env) (b, e1, e2 : expr) (t : type),
      (typecheck g b bool) → (typecheck g e1 t) → (typecheck g e2 t) →
      (typecheck g (if b then e1 else e2) t).
```

There is no difficulty to adapt the translation for such cases. Once the non-linearity between premise outputs has been detected, we use fresh variables and guards as follows:

```
let rec typecheck g e = match g, e with ...
  | g, if (b, e1, e2) →
    (match typecheck g b with
       | bool → let t = typecheck g e1 in
         (match typecheck g e2 with
           | t' when t' = t → t
           | _ → assert false)
       | _ → assert false)
```

In the same way, it is also possible to deal with nonlinearity or symbols of functions in the output of a premise.

6 Conclusion

In this paper, we have presented an extraction mechanism in the context of CIC, which allows us to derive purely functional code from relational specifications implemented as logical inductive types. The main contributions are the formalization of the extraction itself (as a translation function) and the proof of its soundness. In addition, a prototype has been implemented and integrated to the Coq proof assistant, whereas some optimizations (relaxing some limitations) are under development.

Regarding future work, we have several perspectives. First, we aim to prove the completeness of our extraction (the mode consistency analysis should be used in this proof). Concerning our implementation, the next step is to manage large scale specifications, with, for example, the extraction of an interpreter from

a development of the semantics of a programming language (in Coq, there are many developments in this domain). Another perspective is to adapt our mechanism to produce Coq functions, taking benefit from the new facilities offered by Coq to define general recursive functions [10]. These new features rely on the fact that the user provides a well-founded order establishing the termination of the described function. Provided the mode and this additional information, we could extract a Coq function from a logical inductive type, at least for a large class of logical inductive types (e.g. first order unification, strongly normalizable calculi, etc). Finally, the mode consistency analysis should be completed by other analyses like determinism or termination. The logical programming community has investigated abstract interpretation to check this kind of operational properties [5]. Similar analyses could be reproduced in our case. We could also benefit from results coming from term rewriting system tools.

References

1. Isabelle Attali and Didier Parigot. Integrating Natural Semantics and Attribute Grammars: the Minotaur System. Technical Report 2339, INRIA, 1994.
2. Stefan Berghofer and Tobias Nipkow. Executing Higher Order Logic. In Paul Callaghan, Zhaohui Luo, James McKinna, and Randy Pollack, editors, *TYPES*, volume 2277 of *Lecture Notes in Computer Science (LNCS)*, pages 24–40. Springer, December 2000.
3. Patrick Borras, Dominique Clément, Thierry Despeyroux, Janet Incerpi, Gilles Kahn, Bernard Lang, and Valérie Pascual. Centaur: the System. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (PSDE)*, volume 24(2) of *SIGPLAN Notices*, pages 14–24, Boston (MA, USA), November 1988. ACM Press.
4. Catherine Dubois and Richard Gayraud. Compilation de la sémantique naturelle vers ML. In Pierre Weis, editor, *Journées Francophones des Langages Applicatifs (JFLA)*, Morzine-Avoriaz (France), February 1999.
5. Manuel V. Hermenegildo, Germán Puebla, Francisco Bueno, and Pedro López-García. Integrated Program Debugging, Verification, and Optimization using Abstract Interpretation (and the Ciao System Preprocessor). *Science of Computer Programming*, 58(1-2):115–140, 2005.
6. Pierre Letouzey. A New Extraction for Coq. In *TYPES*, volume 2646 of *Lecture Notes in Computer Science (LNCS)*, pages 200–219, Berg en Dal (The Netherlands), April 2002. Springer.
7. David Overton, Zoltan Somogyi, and Peter J. Stuckey. Constraint-based Mode Analysis of Mercury. In *Principles and Practice of Declarative Programming (PPDP)*, pages 109–120, Pittsburgh (PA, USA), October 2002. ACM Press.
8. Mikael Pettersson. A Compiler for Natural Semantics. In Tibor Gyimóthy, editor, *Compiler Construction (CC)*, volume 1060 of *Lecture Notes in Computer Science (LNCS)*, pages 177–191, Linköping (Sweden), April 1996. Springer.
9. Robert F. Stärk. Input/Output Dependencies of Normal Logic Programs. *Journal of Logic and Computation*, 4(3):249–262, 1994.
10. The Coq Development Team. Coq, *version 8.1*. INRIA, November 2006. Available at: `http://coq.inria.fr/`.

11. The Cristal Team. Objective Caml, *version 3.09.3*. INRIA, September 2006. Available at: `http://caml.inria.fr/`.
12. Alberto Verdejo and Narciso Martí-Oliet. Executable Structural Operational Semantics in Maude. *Journal of Logic and Algebraic Programming*, 67(1-2):226–293, 2006.

A Semantic rules of the extraction language

$$\frac{(x,v) \in \Delta}{\Delta \vdash x \triangleright v}\ \mathsf{Var} \qquad \frac{}{\Delta \vdash \mathsf{fail} \triangleright \mathsf{fail}}\ \mathsf{fail}$$

$$\frac{}{\Delta \vdash c^0 \triangleright c^0}\ \mathsf{const}_0 \qquad \frac{}{\Delta \vdash C^0 \triangleright C^0}\ \mathsf{constr}_0$$

$$\frac{\Delta \vdash e_1 \triangleright v_1 \quad \ldots \quad \Delta \vdash e_n \triangleright v_n \quad (c^n, v_1, \ldots, v_n, v) \in \Delta_c}{\Delta \vdash c^n\ (e_1, \ldots, e_n) \triangleright v}\ \mathsf{const}_n$$

$$\frac{\Delta \vdash e_1 \triangleright v_1 \quad \ldots \quad \Delta \vdash e_n \triangleright v_n \quad C^n\ (v_1, \ldots, v_n) \in \Delta_C}{\Delta \vdash C^n\ (e_1, \ldots, e_n) \triangleright C^n\ (v_1, \ldots, v_n)}\ \mathsf{constr}_n$$

$$\frac{\Delta \vdash e_1 \triangleright \mathsf{true} \quad \Delta \vdash e_2 \triangleright v_2}{\Delta \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \triangleright v_2}\ \mathsf{if}_{\mathsf{true}} \qquad \frac{\Delta \vdash e_1 \triangleright \mathsf{false} \quad \Delta \vdash e_3 \triangleright v_3}{\Delta \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \triangleright v_3}\ \mathsf{if}_{\mathsf{false}}$$

$$\frac{}{\Delta \vdash \mathsf{fun}\ x \rightarrow e \triangleright <x \rightsquigarrow e, \Delta>}\ \mathsf{fun} \qquad \frac{}{\Delta \vdash \mathsf{rec}\ f\ x \rightarrow e \triangleright <x \rightsquigarrow e, \Delta>_{\mathsf{rec}(f)}}\ \mathsf{rec}$$

$$\frac{\Delta \vdash e_1 \triangleright v_1 \quad \Delta, (x, v_1) \vdash e_2 \triangleright v_2}{\Delta \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \triangleright v_2}\ \mathsf{let} \qquad \frac{\Delta \vdash e_1 \triangleright v_1 \quad \ldots \quad \Delta \vdash e_n \triangleright v_n}{\Delta \vdash (e_1, \ldots, e_n) \triangleright (v_1, \ldots, v_n)}\ \mathsf{Tuple}$$

$$\frac{\Delta \vdash e_1 \triangleright <x \rightsquigarrow e_3, \Delta> \quad \Delta \vdash e_2 \triangleright v_2 \quad \Delta, (x, v_2) \vdash e_3 \triangleright v_3}{\Delta \vdash e_1\ e_2 \triangleright v_3}\ \mathsf{App}$$

$$\frac{\begin{array}{c}\Delta \vdash e_1 \triangleright <x \rightsquigarrow e_3, \Delta>_{\mathsf{rec}(f)} = c \quad \Delta \vdash e_2 \triangleright v_2 \\ \Delta, (f, c), (x, v_2) \vdash e_3 \triangleright v_3\end{array}}{\Delta \vdash e_1\ e_2 \triangleright v_3}\ \mathsf{App}_{\mathsf{rec}}$$

$$\frac{\begin{array}{c}\Delta \vdash e \triangleright v \quad \mathsf{filter}_\Delta(v, gpat_i) = \Delta_i \\ \mathsf{filter}_\Delta(v, gpat_j) = \mathsf{fail},\ 1 \leq j < i \quad \Delta, \Delta_i \vdash e_i \triangleright v_i\end{array}}{\Delta \vdash \mathsf{match}\ e\ \mathsf{with} \mid gpat_1 \rightarrow e_1\ \ldots \mid gpat_n \rightarrow e_n \triangleright v_i}\ \mathsf{match}$$

$$\text{with } \mathsf{filter}_\Delta(v, gpat) = \begin{cases} \Delta_p, & \text{if } gpat = pat \text{ and } \mathsf{mgu}_\Delta(v, pat) = \Delta_p, \\ \Delta_p, & \text{if } gpat = pat\ \mathsf{when}\ e,\ \mathsf{mgu}_\Delta(v, pat) = \Delta_p, \\ & \text{and } \Delta, \Delta_p \vdash e \triangleright \mathsf{true}, \\ \mathsf{fail}, & \text{otherwise} \end{cases}$$

D.3 PAPER 3: A MAPLE MODE FOR COQ

This paper is related to Chapter 3 and has been published in [52].

Article Submitted to Journal of Symbolic Computation

Dealing with Algebraic Expressions over a Field in Coq using Maple*

David Delahaye[†1] and Micaela Mayero[‡2]

[1]CPR, CNAM, *Département d'Informatique, Paris, France*
[2]PPS, *Université Denis Diderot (Paris 7), Paris, France*

Abstract

We describe an interface between the Coq proof assistant and the Maple symbolic computation system, which mainly consists in importing, in Coq, Maple computations regarding algebraic expressions over fields. These can either be pure computations, which do not require any validation, or computations used during proofs, which must be proved (to be correct) within Coq. These correctness proofs are completed automatically thanks to the tactic `Field`, which deals with equalities over fields. This tactic, which may generate side conditions (regarding the denominators) that must be proved by the user, has been implemented in a reflexive way, which ensures both efficiency and certification. The implementation of this interface is quite light and can be very easily extended to get other Maple functions (in addition to the four functions we have imported and used in the examples given here).

1. Introduction

Computer Algebra and Theorem Proving are two distinct domains of theoretical Computer Science, which are quite wide and involved in many distinct applications. These two domains have their own scientific communities, which seem to be also rather distinct from each other and have little interaction. Actually, Computer Algebra focuses on (symbolic) computation whereas Theorem Proving focuses on validation and, basically, that does not seem to be related. However, over the last 5 years, the situation has begun to change and some people are increasingly interested in making these two worlds communicate. The fact is that

[†]David.Delahaye@cnam.fr, http://cedric.cnam.fr/~delahaye/.

[‡]Micaela.Mayero@pps.jussieu.fr, http://www.pps.jussieu.fr/~mayero/.

*This work has been realized within the Programming Logic Group of the Chalmers University of Technology (Department of Computer Science, Gothenburg, Sweden).

Computer Algebra and Theorem Proving are quite complementary, especially regarding their respective weak points. If we consider a Computer Algebra System (CAS), there is no notion of consistency (we are only interested in the power of computation) and it is quite trivial to perform *false* computations, which obviously have no sense. For example, in the Maple system [8], if we consider the equation $a = 0$, we can derive directly that $1 = 0$ asking for the division and the simplification by a of the equation. Here, the first equation is valid, but we cannot divide it by a because a is equal to 0. Maple does not make any remark and performs the division. Maple supposes that the user knows what he/she is doing and in this trivial example, it seems quite clear that we could have been a little more careful before asking for this division by 0. However, in some more significant examples, this kind of false computations could very easily occur without being noticed by the user and obtaining a wrong result could have more serious consequences. Of course, this is not specific to Maple and could apply to many other CASs. In such a case (as well as in many others), Deduction Systems (DSs) can clearly bring a solution. For example, in the Coq proof assistant [24], it is impossible to derive $1 = 0$ from $a = 0$. Indeed, the division by a is performed by means of a lemma which makes appear explicitely the side condition over a. Thus, a must not be equal to 0 and this cannot be proved.

Conversely, efficient computations are, in general, rather difficult to perform in a proof system. In a DS, it is as easy to crash the system during a *big* computation as it was to derive false propositions in a CAS. As an example, still in Coq, if we try to compute 100×1000, we can notice that it is quite time-consuming (about 20s on a Sun UltraSPARC 450 MHz with a native compiled version of Coq), whereas the computation of 1000×1000 saturates the stack. Again, this is not specific to Coq and this could apply to some other DSs[1]. In the same way as for the previous wrong derivation, CASs can bring a solution to these problematic computations. For example, Maple realizes these two computations immediately (less than a second) on the same machine.

All these examples emphasize that, as said previously, CASs are dedicated to computation whereas DSs are dedicated to validation. But, these examples also show that these domains are complementary because computation (resp. validation) is clearly a weak point of DSs (resp. CASs). So, there is a real interest to be gained in making them interact. This could be done in several ways: to import validation in CASs, to import computation in DSs or, more ambitiously, to build a unique system with both (efficient) computation and validation. In this paper, we focus on the second way, i.e. to import computation in DSs, and we want to share our experience of an interface between Coq and Maple. This idea of interface was proposed some time after the design of the tactic `Field` [12] for Coq, which can automatically prove equalities over fields. Beyond the fact

[1]Actually, in Coq and in most of DSs, there exist more efficient numbers (based on a binary coding) which can be seen as dedicated to computations. However, the recursion scheme is not the usual one and the proofs are less *natural* with these numbers (which are not really appropriate for reasoning).

that this tactic has considerably improved the possibility of Computer Algebra development in Coq, it has also made it possible to import computations from some external CASs (not only Maple) and to validate them automatically, so that it is almost transparent for the user, who has the impression of performing only a computation. Thus, with this interface, we can use, in Coq, some Maple functions regarding expressions over fields and the computations coming from these functions are systematically proved (to be correct) by the tactic `Field`. This work presents a concrete realization and thus contributes to showing the effectiveness of general methods describing the interaction between CASs and DSs (see [4], for example). It should be noted that up to now, in this specific context of CAS/DS, very few interfaces have been designed or at least, formally presented in publications. Compared to some of these interfaces, the main novelty of this paper consists in providing the corresponding automation in the DS to *automatically* prove the external computations from the CAS. Indeed, we claim that such a method is worth being applied in practice only if the automation (at least partial) to verify the correctness of the imported computations has been also developed.

2. The computation/proof paradigm

2.1. Computation vs proof

In Mathematics, there are two distinct traditions: an operational tradition and a denotational tradition. For example, if we want to show that $2 + 2 = 4$, the question does not seem to be the same according to the two traditions. In an operational way, the previous equation is actually not symmetric and this means that we have to compute (or reduce) $2 + 2$ to 4. In a denotational way, the equation is really symmetric and this is a proposition that we have to prove using the axioms of Arithmetics. Historically, Mathematics of Antiquity, coming from Egyptian, Babylonian or Greek knowledge, was centered around operational methods and, little by little, these methods have been replaced by the denotational view of Modern Mathematics. However, nowadays, with the growing development of Computer Science, we can no longer ignore the operational tradition.

Thus, in a *modern* mathematical language, we must be able to show that $2 + 2 = 4$ but also to transform the expression $2 + 2$ into the expression 4. So, this language must not only provide deduction rules, but also computation rules, which are called *rewriting rules*. One of the simplest rewriting rules, which is, in general, implicitly used, is the β-reduction, which simply consists in replacing the formal arguments by their effective arguments. For example, if we apply the function $x, y \mapsto x + y$ to 3 and 4, we obtain the expression $(x, y \mapsto x + y)(3, 4)$, which is computed by replacing x by 3 and y by 4 to get the expression $3 + 4$. If we want to further reduce the expression $3 + 4$, we need other computation rules (regarding the symbol $+$, in particular). Currently, one of the most interesting theoretical frameworks which allows us to handle both deductions and

computations is certainly the *deduction modulo* [13]. In this formalism, thanks to a congruence on propositions, deductions (the undecidable part) and computations (the decidable part) are clearly separated in a clean way. However, as far as the authors are aware, no proof system, based on such a theory, has yet been implemented. In the current other proof systems, the user has to apply explicit computation rules when building proofs and it is more difficult to distinguish computations from deductions.

2.2. Importing computation

In general, especially compared to programming languages, computations are a weak point of proof systems, even if, in these tools, computations are realized considerably faster than deduction checkings, i.e. means which allow us to build a proof tree structure w.r.t. the underlying logic. This can be explained by the fact that proof systems use essentially a pure functional specification language. In particular, it is impossible to handle imperative features like, for example, mutable objects or exceptions, which can be of great help when writing efficient code. Moreover, to ensure consistency, proof systems must only deal with functions which terminate. In some systems, the termination argument must be explicitly given (as in PVS [19]) and in some others, the syntactical class of functions are constrained in order to get the proof termination automatically (as in Coq [24]). In every case, the user is a bit limited in the way of writing his/her code in a language which is clearly not Turing-complete. In the same way, it is quite difficult to handle partial functions, which can only be simulated.

Actually, all the previous features express that we cannot add any computation rule to a logic system. We have to take care of the consistency of the system because these rules are used not only for pure computations but also in proofs. However, this is a real limitation because we may want to perform computations outside a proof and, in this case, there is no special need for restriction regarding the computation rules. To do so, a natural idea is to call external functions (w.r.t. the proof system) which are only used for pure (i.e. not in a proof) and local computations. In this case, the proof system is only interested in the result and not in the function which makes the computation.

This method is quite general and the proof system can use functions from any external tool. Moreover, the proof system only needs a very superficial knowledge of the external tool in order to build the interface. In particular, this can be applied to Computer Algebra which can provide very efficient procedures to any proof system whereas it is quite useless, for the proof system developers, to know how these procedures actually work. Thus, this principle can easily build bridges between Theorem Proving and other very different domains, which may, in turn, lead to a new computational behavior.

2.3. Validating computation

The previous method is a general idea to perform external computations in a proof system but this cannot be used inside proofs. As said previously, to extend these computations to proofs, we have to ensure that they do not break the consistency of the proof system. To do so, we have to prove that every external computation is correct w.r.t. the computation and deduction rules of the proof system. This idea is currently well known and for example, in the context of CA, [4] essentially proposes two approaches regarding this process of validation: a *believing* approach, where the correctness of the external computation is assumed, i.e. added as an axiom, and a *skeptical* approach, where we have to establish the correctness of the computation. Actually, there is also a third approach, called the *autarkic* approach, where the proof assistant makes the computation on its own. This last approach will not be discussed in this paper since we want to import computations. As claimed in [3], the believing approach is unsatisfactory because CASs may have some bugs but especially, as can be seen in the introduction, some necessary side conditions may not be required. Thus, we will have a clearly skeptical view in this paper. More precisely, let us consider the following sequent:

$$\Gamma \vdash P(t)$$

where Γ is a context of hypotheses, P is a predicate and t is a term. If we want to transform t using an external function f_{ext}, we have to use the contextual rule of the equational logic, i.e.:

$$\frac{\Gamma \vdash t = t' \qquad \Gamma \vdash P(t')}{\Gamma \vdash P(t)}\,(=_{\mathsf{Cont}})$$

where $=$ is an equality (it could be either the syntactical equality, also called Leibniz's equality, or more generally a setoid equality) and $t' = f_{\mathsf{ext}}(t)$. Once this rule has been applied, we can go on to the proof with the sequent $\Gamma \vdash P(t')$, but we have also to prove that $t = t'$, i.e. $t = f_{\mathsf{ext}}(t)$, which is the required validation. To do so, there are two alternatives. Either there is a function of the proof system, called g_{prf}, s.t. we can obtain, by computation $g_{\mathsf{prf}}(t) = f_{\mathsf{ext}}(t)$ and conclude using the reflexivity axiom of the equality, but in this case, the use of f_{ext} is completely useless because we can use g_{prf} directly to transform t (as said previously, we are not interested in autarkic computations in this paper). Or there is no function of the proof system which is extensionally equal to f_{ext} and we have to prove $t = t'$ using the deduction rules. As already mentioned, the first alternative will generally not occur because the proof system will call external functions which do not already belong to its environment. The second option is more realistic and shows very clearly the duality computation/proof. Indeed, here, we make an external computation with $f_{\mathsf{ext}}(t)$ and also a corresponding proof $t = f_{\mathsf{ext}}(t)$.

Regarding the proof of $t = t'$, it would be particularly interesting if it could

be completed automatically, even partially, so that the user really has the impression of making a computation using an external function. However, even if this proposition cannot be automatically proved, it is always worth calling the external function which returns t' because, in every case, we use the external tool as an *oracle* which provides a result.

2.4. Application to Coq and Maple

As said previously, even if it is easier, in a proof system, to verify that a computation is correct than to make the computation itself, it is quite important to automate this verification so that the user only sees the computation side of this paradigm. Thus, this restricts the application of Theorem Proving to domains which are decidable or at least partially decidable. Here, in the context of the Coq proof assistant [24] (a direct descendent of LCF, one of the first proof systems), we have designed a strategy (or a *tactic*), called `Field` [12], for reasons which will be explained in section 3, which automatically proves equalities between algebraic expressions over a field. As the problem is not decidable in general, the tactic generates some conditions (typically that some expressions occurring in inverses must not be equal to 0), that the user must prove manually. Even though it was not one of the initial goals, this tactic has built quite a direct bridge toward Computer Algebra Systems, which can perform various symbolic computations over algebraic expressions, because it is now possible to call any computation procedure of a CAS to get a result and to verify the correctness of this result with this new tactic. As a Computer Algebra System, we have chosen Maple [8] because, beyond the fact that Maple is both popular and easy to use, at least for a beginner, it provides all the functions we would like to have in Coq and it allows us to implement very quickly a light interface between the two systems (see section 4). We will see that this cooperation between Coq and Maple makes usual but also certified operations over algebraic expressions possible very easily (the list of available operations can be extended quite quickly by the user) and opens up, as a bonus, new possibilities regarding the automation of Coq in the domain of Computer Algebra.

3. Presentation of the tactic `Field`

Before describing the interface between Coq and Maple, let us focus on the tactic `Field`, which has made it possible to increase the automation of Coq in the domain of Computer Algebra.

3.1. History and motivations

Initially, the tactic `Field` [12] was designed to automate many *small* parts of proofs over the real numbers using the field structure. These parts of proofs were small in the sense that they were quite trivial when considered from the usual and informal mathematical point of view, but they turned out to be quite

tedious to build in a formal proof system. In particular, the idea was to deal with the ε/δ proofs[2] involved in the theorems about limits and derivatives. As a typical example, let us consider the derivative of the addition: given two functions f and g, as well as their derivatives at x_0, denoted $f'(x_0)$ and $g'(x_0)$ (we assume that these two functions are both side differentiable at x_0). By definition, the two derivatives are expressed as follows:

$$f'(x_0) = \lim_{x \to x_0} \frac{f(x) - f(x_0)}{x - x_0}$$

$$g'(x_0) = \lim_{x \to x_0} \frac{g(x) - g(x_0)}{x - x_0}$$

Using the theorem of limit addition, we obtain directly:

$$f'(x_0) + g'(x_0) = \lim_{x \to x_0} \left(\frac{f(x) - f(x_0)}{x - x_0} + \frac{g(x) - g(x_0)}{x - x_0} \right)$$

Using the limit definition[3], we have, for every domain D of $\mathbb{R}$:

$$\forall \varepsilon > 0,\ \exists \delta > 0,\ \forall x \in D \backslash x_0,\ \text{if } |x - x_0| < \delta \text{ then}$$

$$\left| \frac{f(x) - f(x_0)}{x - x_0} + \frac{g(x) - g(x_0)}{x - x_0} - (f'(x_0) + g'(x_0)) \right| < \varepsilon$$

Finally, we must show the following equality:

$$\frac{f(x) - f(x_0)}{x - x_0} + \frac{g(x) - g(x_0)}{x - x_0} = \frac{f(x) + g(x) - (f(x_0) + g(x_0))}{x - x_0}$$

The equality above is quite trivial but the formal proof requires much more work than expected. Indeed, we must reduce the left-hand side with the same denominator and then we can show the equality over the numerators using distributivity, commutativity, etc. In Coq, 10 rewritings are needed to complete this proof.

Initially, the tactic `Field` was implemented to deal with that kind of proofs over the real numbers. Next, the tactic was generalized to handle every field (not only $\mathbb{R}$, but also $\mathbb{C}$ for example). The tactic is currently available in the latest version of Coq (V7.3).

[2] ε/δ proofs are proofs using the explicit definition of limit, i.e. of the form: $\forall \varepsilon > 0. \exists \delta \ldots$.

[3] Here, we have to get to the ε/δ level even if it seems sufficient to perform the algebraic manipulations in the argument of the limit because in the formalization made in Coq, the x occurring in the limit is an abstract variable and it is not allowed to rewrite *under* an abstraction.

3.2. Algorithm

The algorithm is based on the fact that there is already a decision procedure for Abelian rings in the Coq system (tactic `Ring` [5; 6]). So, the idea is to minimize the simplification operations in order to be plugged into the decision procedure on Abelian rings as soon as possible. This essentially means that we simply have to get rid of all the inverses involved in the equality to be solved.

To do so, we propose the following steps:

1. To transform the expressions $x - y$ into $x + (-y)$ and x/y into $x * 1/y$.
2. To look for the inverses involved in the equality in order to build a product of these inverses.
3. To perform a full distribution in the left-hand side and the right-hand side of the equality, except in the inverses.
4. To associate to the right each monomial, except in the inverses.
5. To multiply the left-hand side and the right-hand side of the equality by the product of inverses (built in step 2), generating the side condition that all the inverses must not be equal to 0.
6. To distribute only the product of inverses on the sum of monomials in the left-hand and right-hand side without re-associating to the right.
7. To simplify the inverses in the monomials using the field rule $x*1/x = 1$, if $x \neq 0$ and performing permutations of the monomial components if necessary, that is to say if there are some inverses remaining and that the field rule cannot be applied[4].
8. To loop to step 2 if there are some inverses remaining.

The first step allows us to limit the operations we have to deal with (we get rid of the binary minus and the division which are not primitive in the field definition) before reducing the problem to the ring level. Step 4 is clearly not necessary, but rather more efficient, because this avoids a double recursive call in the functions which handle these expressions. The last step, which may involve further iterations, is justified by the possibility of having other inverses in the inverses. After all these steps, we obtain an expression without inverses and we only have to call the decision procedure for Abelian rings to conclude. The correctness and the complexity of this algorithm will be discussed later in subsection 3.4.

3.3. A complete example

Let us consider a small example of the evaluation of the previous algorithm where it is needed to make a recursive call. Given x, y and z, three variables over a field, we propose to show the following equality under the hypotheses that $x \neq 0$, $y \neq 0$ and $z \neq 0$:

[4]Here, we do not have to verify that $x \neq 0$, because this condition has already been generated during the step of multiplication by the product of all the inverses.

$$\frac{1}{x} * (x - \frac{x}{\frac{y}{z}}) = 1 - \frac{z}{y}$$

First, we transform the binary minus and the divisions:

$$\frac{1}{x} * (x + (-x) * \frac{1}{y * \frac{1}{z}}) = 1 + (-z) * \frac{1}{y}$$

We build the product of inverses, we call p:

$$p = x * ((y * \frac{1}{z}) * y)$$

We perform a full distribution in the left-hand and right-hand sides of the equality, except in the inverses:

$$\frac{1}{x} * x + \frac{1}{x} * ((-1) * x * \frac{1}{y * \frac{1}{z}}) = 1 + (-1) * z * \frac{1}{y}$$

We associate to the right each monomial, except in the inverses:

$$\frac{1}{x} * x + \frac{1}{x} * ((-1) * (x * \frac{1}{y * \frac{1}{z}})) = 1 + (-1) * (z * \frac{1}{y})$$

We multiply the left-hand side and the right-hand side of the equality by p generating a correctness condition:

$$x * ((y * \tfrac{1}{z}) * y) * (\tfrac{1}{x} * x + \tfrac{1}{x} * ((-1) * (x * \tfrac{1}{y * \frac{1}{z}}))) = x * ((y * \tfrac{1}{z}) * y) * (1 + (-1) * (z * \tfrac{1}{y}))$$

with $x * ((y * \frac{1}{z}) * y) \neq 0$.

We distribute this product on the monomials without re-associating to the right:

$$x * ((y * \tfrac{1}{z}) * y) * (\tfrac{1}{x} * x) + x * ((y * \tfrac{1}{z}) * y) * (\tfrac{1}{x} * ((-1) * (x * \tfrac{1}{y * \frac{1}{z}}))) = x * ((y * \tfrac{1}{z}) * y) * 1 + x * ((y * \tfrac{1}{z}) * y) * ((-1) * (z * \tfrac{1}{y}))$$

We simplify the inverses performing permutations if necessary:

$$(y * \frac{1}{z}) * y * x + y * ((-1) * x) = x * ((y * \frac{1}{z}) * y) * 1 + x * y * (-1)$$

Some inverses are remaining (two occurrences of $\frac{1}{z}$) and we have to apply the previous steps again. The new product of inverses, which we call p', is the following:

$$p' = z$$

Here, each member of the equality is already fully distributed and there is no need to apply step 3. Next, we associate to the right each monomial:

$$y*(\frac{1}{z}*(y*x))+y*((-1)*x) = x*(y*(\frac{1}{z}*(y*1)))+x*(y*(-1))$$

We multiply both sides of the equality by p' generating another side condition:

$$z*(y*(\frac{1}{z}*(y*x))+y*((-1)*x)) = z*(x*(y*(\frac{1}{z}*(y*1)))+x*(y*(-1)))$$

with $z \neq 0$.

We distribute p' on the monomials without re-associating to the right:

$$z*(y*(\frac{1}{z}*(y*x)))+z*(y*((-1)*x)) = z*(x*(y*(\frac{1}{z}*(y*1))))+z*(x*(y*(-1)))$$

We simplify the inverses:

$$y*(y*x)+z*(y*((-1)*x)) = x*(y*(y*1))+z*(x*(y*(-1)))$$

Thus, we obtain an equality over an Abelian ring structure, which can be solved calling the corresponding decision procedure. We have also two side conditions (there are as many conditions as recursive calls of the algorithm), $p \neq 0$ and $p' \neq 0$, which have to be proved manually (more precisely, they are not proved automatically by the algorithm).

3.4. Implementation

An originality of the tactic `Field` is that it has been implemented in a reflexive way. This particular coding ensures both efficiency and soundness. To fully understand how this is possible, let us briefly recall what a reflection is.

To code a tactic in a formal proof system, there are globally two possible options. An explicit coding using rewriting or a reflexive coding using reduction. An explicit coding, also called LCF's approach, may be very inefficient due to the use of rewriting, which may be quite time-consuming but also space-consuming in some systems based on Curry-Howard's isomorphism[5], where proofs are λ-terms, like in Coq, Lego [21] or Alfa [11], for example. A reflexive coding is an alternative method, which is quite satisfactory according to these two criteria. Indeed, rewritings are replaced by more efficient reduction steps and proof term size is the same as the size of the goals to be proved.

[5]This isomorphism consists in the fact that it is possible to build a bijection between propositions and types of a typed λ-calculus, which implies a second bijection between proofs and λ-terms.

The reflection principle is the following: given a language of concrete terms (typically any type of the system), say C, and a language of abstract terms (typically an inductive type), say A. As it is not possible, in general, to handle the terms of C as we would like to (we may not be able to perform pattern-matching, for example), the idea is to reflect a part of C into the language A, which is supposed to be isomorphic to this part. The first step, called metaification[6], consists in converting terms of C into terms of A. More precisely, this consists, for a term c of C, in building the term a of A such that $[\![a]\!]_v = c$, where $[\![.]\!]$ is the interpretation function from A to C (which can be coded in the system), v is the canonical map of the parts of C which are not reflected and $=$ is the syntactical equality.

Next, we can handle the converted terms of A and we can write some transformation functions on A. To use these functions, we only have to prove correctness lemmas, which, for example, given a transformation function φ from A to A, have the following form: $\forall a \in A.[\![\varphi(a)]\!]_v = [\![a]\!]_v$. In particular, this point means that, in the reflection process, the tactic and its correctness proof cannot be dissociated, they are built simultaneously.

Finally, once this lemma has been applied, we only have to fully reduce the expression in order to perform the transformation on the abstract term of A and to re-obtain a concrete term of C.

The situation can be summarized by figure 1 and here is a small example to better understand how this general scheme of reflection actually works:

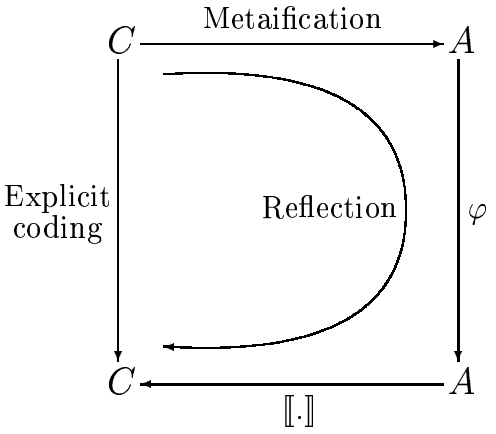

Figure 1: Reflection principle.

EXAMPLE 3.1 (HANDLING PROPOSITIONS): *The probably best example of objects which can only be handled at the meta-logic level are basic propositions. For*

[6]This word was coined by Samuel Boutin [6], but, in the literature, this step is also called quotation or reification.

instance, let us consider the strategy which replaces $\Phi \wedge \top$ *with* Φ*, where* Φ *is a proposition and* $\top$ *is the universally true proposition. To write such a strategy according to a reflective approach consists in reflecting a part of propositions, actually* $\wedge$ *as well as* $\top$*, using a concrete type (an inductive type). Let us call* T *this concrete type, which is made of three constructors:* and *corresponding to* $\wedge$*,* true *to* $\top$ *and a last one to variables (these variables are mapped to propositions which are not reflected, i.e. propositions formed from* $\vee$*,* $\Rightarrow$ *, etc). Now, the strategy can be written as a function, we call* $\mathcal{S}$*, over terms of* T*; this is a quite trivial recursive function working by usual pattern-matching. To be applied, this strategy needs its correctness lemma (that is one of the main points of reflection's principle: the correctness of the strategy is established during the implementation process), which is:*

$$\forall t \in T. [\![\mathcal{S}(t)]\!]_v = [\![t]\!]_v$$

where $[\![.]\!]$ *is the interpretation function of abstracted propositions of* T *into usual propositions (as* $\mathcal{S}$*, this is also a quite trivial recursive function working by usual pattern-matching) and* v *is the canonical map of the parts which have not been reflected during the metaification pass. Here, since we use equality between propositions, we must remark that we need an extensionality axiom to prove this lemma.*

If we consider the expression $\Phi \wedge \top \Rightarrow \Phi$*, where* Φ *is an arbitrary complicated proposition, let us see how to apply the previous strategy. First, we have to metaified the expression (this is the only pass which requires to switch to the meta-logic level) and we obtain:*

$$[\![\mathsf{and}(p, \mathsf{true})]\!]_{[(p,\Phi)]} \Rightarrow \Phi$$

Next, we apply the correctness lemma of $\mathcal{S}$ *(a simple rewriting):*

$$[\![\mathcal{S}(\mathsf{and}(p, \mathsf{true}))]\!]_{[(p,\Phi)]} \Rightarrow \Phi$$

We can apply the definition of $\mathcal{S}$*:*

$$[\![p]\!]_{[(p,\Phi)]} \Rightarrow \Phi$$

Finally, we apply the definition of $[\![.]\!]$ *to get* $\Phi \Rightarrow \Phi$*.*

For further information regarding the use of reflection, see [5; 6; 15; 22; 18].

Thus, with this very specific implementation, the proofs using `Field` can be verified faster using only reductions and are smaller. Moreover, this reflexive coding ensures the correctness of the tactic and the built procedure is then *certified*. It is important to realize how this last point is fundamental because, with a direct coding, it is not even possible to state the correctness *inside* the proof system (indeed, this is a meta-statement which can only be verified outside the system). Here, we provide an efficient but also valid tactic. For further details regarding the correctness lemmas and the complexity of `Field`, see [12].

4. Interfacing Coq and Maple

4.1. Method

According to the paradigm described in section 2, we have imported, into Coq, computations from Maple to build terms, in definitions for example, but also to be used in proofs of propositions, called *goals*. Pure computations, i.e. not used in proofs, can be realized with the following expression (we will see some examples in section 5):

`Eval` < Maple *function* > `in` < Coq *term* >

which simply consists in applying the Maple function to the Coq term and returning a new Coq term. Obviously, there is interfacing work to do between Coq terms and Maple terms, which will be described in subsection 4.2. As said in section 2, computations of this kind do not require any verification from Coq (that is why they are called pure computations) and no tactic is called to build any proof (even if the Maple function does not do what it was coded for, it cannot break the consistency of Coq).

Computations which occur in proofs are called with the usual syntax of Coq's tactics as follows:

< Maple *function* > < Coq *term* >.

where we assume that we are in the proof editing mode and that the Coq term has an occurrence in the goal we try to prove. If we call f, the Maple function, and t, the Coq term, the effect of this "Maple tactic" on a goal of the form $\Gamma \vdash P(t)$ is to generate a subgoal $\Gamma \vdash P(f(t))$, the user has to complete next, and a subgoal $\Gamma \vdash t = f(t)$, to verify, as seen in section 2, that the Maple function has given a correct result. The latter subgoal is automatically solved by the application of the tactic `Field` since, as said in subsection 2.4, we deal with Maple functions which handle algebraic expressions over fields. As seen in section 3, applying `Field` may generate side conditions (typically, that some expressions must not be equal to 0) and, after applying this Maple tactic, the user may have to prove more than one subgoal. Figure 2 gives a global view of how Maple computations are used in Coq proofs.

The Maple functions dealing with algebraic expressions over fields which have been exported are the following (we also give their informal and brief semantics coming from Maple's documentation):

Maple function	Action
`simplify`	Apply simplification rules to an expression
`factor`	Factorize a multivariate polynomial
`expand`	Expand an expression
`normal`	Normalize a rational expression

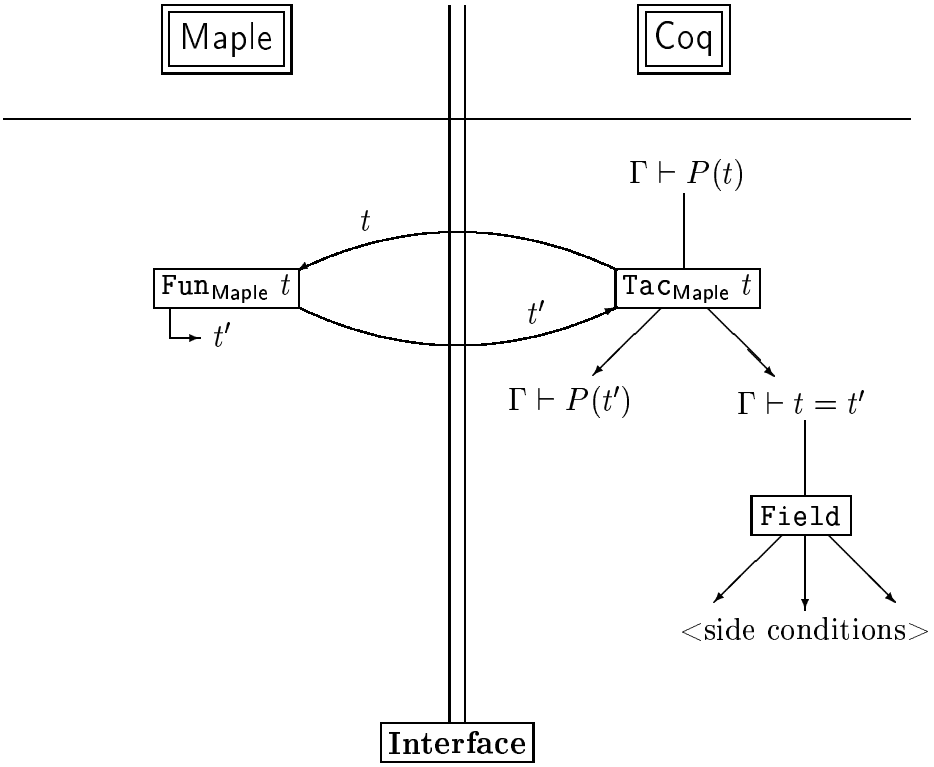

Figure 2: Using Maple computations in Coq proofs.

In section 5, we will see some examples which give a more precise idea regarding the behavior of these functions, but for further information, the reader can refer to the Maple reference manual [8]. Syntactically, these functions are imported in Coq with the same names where the first character has been capitalized giving `Simplify`, `Factor`, etc. Currently, we only have functions with an arity of 1 (we will see in subsection 4.2 that adding other functions is quite direct and very easy for the user), but there is no specific difficulty in extending this interface to functions with higher arities.

4.2. Implementation

Implementing the interface between Coq and Maple has been done quite quickly and almost for free, essentially due to the reflexive coding of the tactic `Field`. When we want to transform a Coq term t with a Maple function, we use the reflexive structure of field which has been created for `Field` and we perform a metaification (see figure 1) to get a term t_{abs}. With this new term t_{abs}, which is only an algebraic expression over a field, we build another concrete term t_{Maple}, expressed in the Maple syntax and which is completely isomorphic to t_{abs}. Next, we can call the corresponding Maple function to obtain a term t'_{Maple}, which can be directly transformed into a term t'_{abs} of the reflexive structure. Finally, the

t'_{abs} is interpreted by the function $[\![.]\!]$ of figure 1 to re-obtain a Coq term t', which can be used either as the result of a pure computation or in a proof.

To use the previous result, i.e. t' coming from t, in a proof, we simply make a cut of the equality $t = t'$ and we rewrite t into t' in the goal where t occurs. The equality $t = t'$ is then proved by `Field`. This is done by the following short Coq script:

```
Replace t with t';[Idtac|Field].
```

where the tactic `Replace` makes the cut of `t=t'` (we obtain two subgoals) to rewrite `t` into `t'` in the main subgoal, and the tactic `Idtac` leaves the goal unchanged. The notation `Tac;[`$\mathtt{Tac}_1$`|...|`$\mathtt{Tac}_n$`]` means that we apply the tactic `Tac` to the current goal, $\mathtt{Tac}_1$ to the first subgoal, ..., and $\mathtt{Tac}_n$ to the nth subgoal. Here, the first subgoal is the initial goal where `t` has been replaced by `t'`, which is left identical by `Idtac` (this is the expected subgoal, which must be completed by the user afterwards), and the second subgoal is the equality `t=t'`, which is automatically proved by `Field`.

The whole implementation has been realized in a very light way. The interface between Coq and Maple does not require a specific architecture and, in particular, there is no transfer protocol, no use of sockets or other sophisticated communication systems, but only a simple pipe created by Coq for each call of a Maple function. Of course, neither of the two systems has any information regarding the state of the other one, but this is quite useless and when Maple is called, there is no need for a specific context. Incidentally,we can also notice that the roles of the two tools are asymmetric. Indeed, Coq is in charge and deals with the interface. Maple is only called when Coq needs a computation, but Maple cannot call Coq to verify a result in its own context.

The implementation of this interface is available as the Coq contribution Chalmers/MapleMode, which can be found on the Coq web site: http://coq.inria.fr. As just said, the implementation is quite light and the corresponding code is very short with about 300 lines of ML. The contribution contains also some examples of use for each imported Maple function.

5. Examples

Here, we give some examples of the new tactics `Simplify`, `Factor`, `Expand` and `Normal`, now available in Coq. As seen previously, these tactics call the corresponding Maple functions and we propose some examples where they can be used either to prove a goal or only to compute a result. These examples are rather small in order to show quite clearly how the new corresponding tactics work and in particular, to understand when some side conditions are generated. A more complete and realistic example can be found in appendix A.

To use Maple in Coq, we have to import the specific module `Maple` (the in-

terface) with the Coq command `Require`. If Maple is available on the machine where Coq is running then the following heading appears:

```
Welcome to Coq 7.3 (May 2002)

Coq < Require Maple.
[Loading ML file maple.cmo ...
Coq is now powered by Maple
[Maple V Release 5 (Chalmers Tekniska Hogskola)]

    |\^/|              v
._|\|   |/|_.  ====>  <O___,,
 \  MAPLE  /   ====>   \VV/
 <____ ____>            //
      |
done]
```

The examples we give are over the field of real numbers. We suppose that the module `Reals` has been imported, which allows us to use the whole real number library and, in particular, to call the tactic `Field` instantiated for this field. This module has a specific syntax and the real expressions are parenthesized by ``. The opposite of x, the inverse of x and $x \neq y$ are respectively noted `-x`, `/x` and `x<>y`. Additionally, there are also a binary minus and a division, which are represented by `-` and `/`.

5.1. Maple tactics

5.1.1. `Simplify`

We want to prove the following lemma, we call `simp1`, stating that for every real number x and y, if $x \neq 0$ and $y \neq 0$ then:

$$(\frac{x}{y} + \frac{y}{x})\ x.y - (x.x + y.y) + 1 > 0$$

After having entered the goal and assumed the hypotheses (tactic `Intros`), we obtain the following goal:

```
simp1 < Intros.
1 subgoal

  x : R
  y : R
  H : ``x <> 0``
  H0 : ``y <> 0``
  ============================
   ``(x/y+y/x)*x*y-(x*x+y*y)+1 > 0``
```

The most direct way of completing this goal is to simplify the left-hand side term $(\frac{x}{y} + \frac{y}{x})\ x.y - (x.x + y.y) + 1$ with the new tactic `Simplify` as follows:

```
simp1 < Simplify ``(x/y+y/x)*x*y-(x*x+y*y)+1``.
2 subgoals

  x : R
  y : R
  H : ``x <> 0``
  H0 : ``y <> 0``
  ============================
   ``1 > 0``

subgoal 2 is:
 ``y*x <> 0``
```

The tactic `Simplify` calls the Maple function `simplify` to get the simplified term `1` and replaces it in the goal to obtain `1 > 0`. We can notice that another subgoal appears. This subgoal is generated by applying the tactic `Field`, which is called to validate the previous replacement by proving the equality between the initial and simplified terms, i.e. $(\frac{x}{y} + \frac{y}{x})\ x.y - (x.x + y.y) + 1 = 1$. This automatic proof creates an additional goal requiring that the denominator product must not be equal to 0, that is to say $y \times x \neq 0$. These two subgoals are then completed by very simple tactics on real numbers, respectively `Sup0` and `SplitRmult`.

5.1.2. `Factor`

Regarding the tactic `Factor`, we can give the following example in which we want to factorize the term:

$$a^3 + 3a^2b + 3ab^2 + b^3$$

```
fact1 < Intros.
1 subgoal

  a : R
  b : R
  H : ``a+b > 0``
  ============================
   ``a*a*a+3*a*a*b+3*a*b*b+b*b*b > 0``

fact1 < Factor ``a*a*a+3*a*a*b+3*a*b*b+b*b*b``.
1 subgoal

  a : R
  b : R
  H : ``a+b > 0``
```

```
  ============================
   ``(a+b)*(a+b)*(a+b) > 0``
```

5.1.3. `Expand`

For the tactic `Expand`, let us consider the following example in which we want to expand the term:

$$(a - b)(a + b)$$

```
expd1 < Intros.
1 subgoal

  a : R
  b : R
  H : ``a <> b``
  ============================
   ``(a-b)*(a+b) <> 0``

expd1 < Expand ``(a-b)*(a+b)``.
1 subgoal

  a : R
  b : R
  ============================
   ``a*a+ -(b*b) <> 0``
```

We can notice that the tactic `Expand` (actually, the Maple function `expand`) expands and also simplifies the expression.

5.1.4. `Normal`

We may use the tactic `Normal` in the following example in which we want to normalize the term:

$$\frac{x^2}{(x-y)^2} - \frac{y^2}{(x-y)^2}$$

In Maple, the `normal` function provides a basic form of simplification. It recognizes those expressions equal to zero which lie in the domain of "rational functions". This includes any expression constructed from sums, products and integer powers of integers and variables. The expression is converted into the "factored normal form". This is the form numerator/denominator, where the numerator and denominator are relatively prime polynomials with integer coefficients. Thus, in Coq, let us try to prove that if $x - y \neq 0$ then the previous expression is normalized into $\frac{x+y}{x-y}$:

```
norm1 < Intros.
1 subgoal

  x : R
  y : R
  H : ``x-y <> 0``
  ============================
   ``x*x/((x-y)*(x-y))-y*y/((x-y)*(x-y)) == (x+y)/(x-y)``

norm1 < Normal ``x*x/((x-y)*(x-y))-y*y/((x-y)*(x-y))``.
2 subgoals

  x : R
  y : R
  H : ``x-y <> 0``
  ============================
   ``(x+y)*/(x+ -y) == (x+y)/(x-y)``

subgoal 2 is:
 ``(x+ -y)*((x+ -y)*(x+ -y)) <> 0``
```

where the notation /a is just a syntactical abbreviation for the inverse, i.e. 1/a.

The left-hand side term of the equality in the first subgoal is the result returned by Maple. The second subgoal has been generated by `Field`. Actually, we can notice that, applying the algorithm of subsection 3.2, two denominators must not be equal to 0, that is to say $x - y$ and $(x - y)^2$, even if it is redundant.

5.2. Maple evaluation

As seen previously in subsection 4.1, we can also perform only pure Maple computations thanks to the command `Eval...in`. These computations do not need to be proved and there is no guarantee that the results are correct but it is not necessary to generate side conditions because the consistency of the system is not endangered in any way. In this case, we just want to use Coq as a calculator with the help of Maple.

For example, let us consider the computation of the following characteristic polynomial:

$$\begin{vmatrix} -1-x & 4 & 2 \\ -5 & 7-x & 5 \\ 7 & -6 & -6-x \end{vmatrix}$$

We define the characteristic polynomial, we call `car1`, by expanding the determinant w.r.t. the first column:

```
Coq < Definition car1 [x:R] := Eval Factor in
```

```
Coq <   ``(-1-x)*((7-x)*(-6-x)+30)+5*(4*(-6-x)+12)+
Coq <     7*(20-2*(7-x))``.
car1 is defined
```

As we have factorized (with `Factor`) the previous development, we have a definition `car1` which can be directly used to build the corresponding diagonal matrix:

```
Coq < Print car1.
car1 = [x:R]`` -(x+ -1)*(x+ -2)*(x+3)``
     : R->R
```

We can perform other computations, in exactly the same way, with the other Maple functions that have been imported.

6. Conclusion

6.1. Summary

In this paper, several goals have been achieved:

- We have emphasized the paradigm computation/proof, which appears, for example, in the opposition CAS/DS. In particular, it is apparent that no concrete system (not only formalisms), which could handle both computation and deduction in a clean way, has been ever designed.
- In the context of the Coq proof assistant, we have presented the tactic `Field`, which can automatically prove the equalities over fields. Regarding the implementation, this tactic has been realized in a reflexive way, which ensures both efficiency and certification.
- We have described a very light interface between Coq and Maple, which allows us to perform Maple computations in Coq. These can be pure computations, which do not require any validation, or computations used during proofs, which are then *automatically* certified using the tactic `Field`.
- We have provided some examples of use (for each new Coq computation based on a Maple function) with either pure computations (`Eval...in`) or computations used during proofs (usual tactic call), which may generate side conditions arising from the application of the tactic `Field`.

6.2. Related work

The idea of making CASs and DSs interact has been also explored by some other authors. A study of this interaction can be found in [4] where, as seen in section 2, several approaches (believing, skeptical and autarkic) are discussed. This is a very general characterization of the communication of mathematical contents between CASs and DSs so that any concrete interface between a CAS and DS should, *a priori*, fall into the scope of one of these approaches.

Other approaches, which differ from that put forward in this paper, consist in providing CAS users with an interface with a DS to verify side conditions. For example, in [1], the idea is to track the conditions, which are implicit in the CAS, but which can make the result go wrong if they are ignored. In this case, the CAS is Maple and the DS is PVS [19]. This view is quite different because it is dedicated to CAS users and there is a graphic interface which is used to build some proofs interactively without interacting directly with PVS. Moreover, the PVS proofs must be completed manually (no automation is provided for a specific Computer Algebra domain). In a similar way, another work [10] aims at using the automated theorem prover Otter to discard trivial conjectures generated by the HR theory generation tool [9] about Maple functions. Once these conjectures proved, Otter leaves the *interesting* conjectures, which cannot be easily proved.

In more closely related approaches, the CAS is devoted to help a DS. For instance, in [2], Maple is used to extend the power of rewriting of the Isabelle theorem prover [20]. No proof obligation is generated and according to [4], the interface is based on a believing principle (i.e., each rewriting using a Maple computation is set as an axiom; obviously, this is managed by the Isabelle simplifier at the meta-logic level and this is quite transparent for the user). Another example is [16] consisting in an interface between HOL [14] and Maple, where Maple is supposed to help HOL users to perform certain computations. This tool is mainly dedicated to the domain of real numbers and Maple is called with the two methods `SIMPLIFY`/`FACTORIZE`. The results returned by Maple are proved in HOL (as mentioned in the title of the paper, this is a skeptical approach) but not systematically so. In contrast to our approach, the imported functions are not embedded in specific HOL tactics, which call Maple for computations and prove the correctness of these computations automatically. In fact, the user calls the Maple functions in HOL and, if he/she wants to use the results of these computations in proofs, he/she has to prove their correctness either manually or automatically (if there is a corresponding tactic). Finally, in the idea of integrating Computer Algebra into Proof Planning, [23] presents a concrete implementation where Maple is used to enhance the computational power of the Ωmega system [25]. The computations are checked by means of an external tool, which provide protocol information to aid the verification in Ωmega. This external tool deals essentially with verifications involving arithmetic and for other kind of verifications, it must be modified by integrating an appropriate algorithm.

6.3. Future work

An extension of this work could be to deal with other kinds of computations such as limits, derivatives, etc. (these notions have already been formalized in Coq). With this interface, these computations can already be quite easily imported into Coq and if we also want to deal with them in proofs, it is always possible to prove their correctness, at least manually. However, in the same way as for the tactic `Field`, the idea would be to also develop the corresponding automation.

For example, if we denote f', the derivative of f, we would like to provide a new tactic which can automatically prove in Coq that f' is the derivative of f, for some specific forms of f, precisely as is done for algebraic expressions over a field with the tactic `Field`.

Another application of this interface could be to use Maple to compute polynomial gcd's. Currently, we are developing a decision procedure for first order formulae over algebraic closed fields based on a method of quantifier elimination, and as such methods require many gcd computations, we plan to use Maple to do so. The correctness proofs will also be automated using the tactic `Field` and Bezout's relation (the coefficients will be given by Maple). Once realized, a large part of this procedure could be also reused to deal with real closed fields. It is particularly interesting to finalize this work because it will mean that CASs can be used in DSs not only to carried out some computations but also to enhance the automation of such systems.

Finally, we would like to use a more general language to express the output data coming from Coq. The idea would be to interface Coq with other CASs. Currently, the data produced by the interface are quite *ad hoc* and can only be parsed by Maple. A good choice could be to adopt languages like OpenMath [7] or OMDoc [17], which seem to be emerging standards and already have bindings with many existing CASs.

A. A complete example: quadratic forms

In this example, we propose to show the equality between two quadratic forms where the right-hand side form is expressed in such a way that we can compute the rank of the quadratic form trivially (i.e. a linear combination of squares of independent linear forms). To do so, we use Gauss's algorithm[7] on the left-hand side quadratic form to get the right-hand side form. Actually, we could rather expand and reduce the right-hand side to get the left-hand side expression, but we suppose that we are in the situation where we try to compute the rank of the left-hand side quadratic form (which is usually the case). The equality we consider is the following:

$$x^2 + y^2 - xy - yz - zx + z^2 = (x - \frac{y+z}{2})^2 + \frac{3}{4}(y-z)^2$$

After some simple algebraic modifications (associativity and commutativity), we have to prove (the lemma has been named `quadratic`):

```
quadratic <
1 subgoal
```

[7] In Coq and some other proof assistants based on the Curry-Howard isomorphism, a typical proof style can be to use an algorithm to guide a proof. Afterwards, a program implementing the given algorithm can be automatically extracted from this proof.

```
  x : R
  y : R
  z : R
  ============================
   ``x*x-x*y-z*x+y*y-y*z+z*z ==
     (x-(y+z)/2)*(x-(y+z)/2)+3/4*(y-z)*(y-z)``
```

The first step consists in partially factorizing $x^2-xy-zx$ into $(x-\frac{y+z}{2})^2-\frac{(y+z)^2}{4}$:

```
quadratic < Replace ``x*x-x*y-z*x`` with
quadratic <         ``(x-(y+z)/2)*(x-(y+z)/2)-(y+z)*(y+z)/4``.
2 subgoals

  x : R
  y : R
  z : R
  ============================
   ``(x-(y+z)/2)*(x-(y+z)/2)-(y+z)*(y+z)/4+y*y-y*z+z*z ==
     (x-(y+z)/2)*(x-(y+z)/2)+3/4*(y-z)*(y-z)``

subgoal 2 is:
 ``(x-(y+z)/2)*(x-(y+z)/2)-(y+z)*(y+z)/4 == x*x-x*y-z*x``
```

The second goal has been generated by the tactic `Replace`. To prove this goal, we use the new tactic `Simplify`, coming from Maple, to simplify the left-hand side term and to get the right-hand side term. The idea is to conclude using the tactic `Reflexivity` which applies the reflexivity property of the equality. As `Simplify` calls the tactic `Field`, another subgoal is also generated, i.e. the side condition that $2 \times (2 \times 4) \neq 0$. This condition is simply proved by the tactic `DiscrR` which deals with equalities and inequalities over reals involving constants:

```
quadratic < 2:Simplify ``(x-(y+z)/2)*(x-(y+z)/2)-
quadratic <              (y+z)*(y+z)/4``;
quadratic <   [Unfold Rminus;Rewrite (Rmult_sym z x);
quadratic <    Reflexivity |DiscrR].
1 subgoal

  x : R
  y : R
  z : R
  ============================
   ``(x-(y+z)/2)*(x-(y+z)/2)-(y+z)*(y+z)/4+y*y-y*z+z*z ==
     (x-(y+z)/2)*(x-(y+z)/2)+3/4*(y-z)*(y-z)``
```

where `Unfold Rminus` replaces the binary minus with the unary minus (in

the right-hand side part) and `Rewrite (Rmult_sym z x)` commutates `z` and `x` (also in the right-hand side part).

After some associativity manipulations, we can perform a full distribution in $-\frac{(y+z)^2}{4}+y^2-yz+z^2$ thanks to the tactic `Expand` (the side condition $4\times 2\neq 0$ is generated and proved by `DiscrR`):

```
quadratic < Expand ``-((y+z)*(y+z)/4)+y*y-y*z+z*z``;
quadratic <   [Idtac|DiscrR].
1 subgoal

  x : R
  y : R
  z : R
  ============================
   ``(x-(y+z)/2)*(x-(y+z)/2)+(3*/4*y*y+
     -(3*/2*y*z)+3*/4*z*z) ==
     (x-(y+z)/2)*(x-(y+z)/2)+3/4*(y-z)*(y-z)``
```

Now, we can factorize to get the last square using `Factor` (again a side condition, i.e. $4\times 2\neq 0$ is generated and proved by `DiscrR`):

```
quadratic < Factor ``3*/4*y*y+ -(3*/2*y*z)+3*/4*z*z``;
quadratic <   [Idtac|DiscrR].
1 subgoal

  x : R
  y : R
  z : R
  ============================
   ``(x-(y+z)/2)*(x-(y+z)/2)+3*/4*(y+ -z)*(y+ -z) ==
     (x-(y+z)/2)*(x-(y+z)/2)+3/4*(y-z)*(y-z)``
```

Finally, once `3*/4` and `y+-z` have been respectively transformed into `3/4` and `y-z`, we can conclude using the tactic `Reflexivity`.

The Coq proof script is given in detail below:

```
Lemma quadratic:(x,y,z:R)
  ``x*x+y*y-x*y-y*z-z*x+z*z==
    (x-(y+z)/2)*(x-(y+z)/2)+3/4*(y-z)*(y-z)``.
Proof.
  Intros.
  Replace ``x*x+y*y-x*y-y*z-z*x+z*z`` with
          ``x*x-x*y-z*x+y*y-y*z+z*z``;[Idtac|Ring].
  Replace ``x*x-x*y-z*x`` with
          ``(x-(y+z)/2)*(x-(y+z)/2)-(y+z)*(y+z)/4``.
  2:Simplify ``(x-(y+z)/2)*(x-(y+z)/2)-(y+z)*(y+z)/4``;
```

```
    [Unfold Rminus;Rewrite (Rmult_sym z x);Reflexivity
    |DiscrR].
  Unfold 2 Rminus.
  Replace ``(x-(y+z)/2)*(x-(y+z)/2)+
            -((y+z)*(y+z)/4)+y*y-y*z+z*z`` with
          ``(x-(y+z)/2)*(x-(y+z)/2)+
            (-((y+z)*(y+z)/4)+y*y-y*z+z*z)``;[Idtac|Ring].
  Expand ``-((y+z)*(y+z)/4)+y*y-y*z+z*z``;[Idtac|DiscrR].
  Factor ``3*/4*y*y+ -(3*/2*y*z)+3*/4*z*z``;[Idtac|DiscrR].
  Fold ``3/4``;Fold ``y-z``.
  Reflexivity.
Save.
```

References

1. Andrew Adams, Martin Dunstan, Hanne Gottliebsen, Tom Kelsey, Ursula Martin, and Sam Owre. Computer Algebra Meets Automated Theorem Proving: Integrating Maple and PVS. In R.J. Boulton and P.B. Jackson, editors, *TPHOLs 2001*, volume 2152 of *LNCS*, pages 27–42. Springer-Verlag, 2001.

2. Clemens Ballarin, Karsten Homann, and Jacques Calmet. Theorems and Algorithms: An Interface between Isabelle and Maple. In *International Symposium on Symbolic and Algebraic Computation*, pages 150–157, 1995.

3. Henk Barendregt and Erik Barendsen. Autarkic Computations in Formal Proofs. *Journal of Automated Reasoning (JAR)*, 28(3):321–336, April 2002.

4. Henk Barendregt and Arjeh M. Cohen. Electronic Communication of Mathematics and the Interaction of Computer Algebra Systems and Proof Assistants. *Journal of Symbolic Computation (JSC)*, 32(1/2):3–22, July/August 2001.

5. Samuel Boutin. *Réflexions sur les quotients.* PhD thesis, Université Paris 7, April 1997.

6. Samuel Boutin. Using Reflection to Build Efficient and Certified Decision Procedures. In *Theoretical Aspects of Computer Software*, pages 515–529, 1997.

7. Olga Caprotti and Arjeh M. Cohen. On the Role of OpenMath in Interactive Mathematical Documents. *Journal of Symbolic Computation (JSC)*, 32(4):351–364, September 2001.

8. Bruce W. Char, Keith O. Geddes, Gaston H. Gonnet, Benton L. Leong, Michael B. Monagan, and Stephen M. Watt. *The* Maple V *Language Reference Manual.* Springer-Verlag, New York, 1991. ISBN 0387976221.

9. Simon Colton. The HR Program for Theorem Generation. In *Proceedings of the 18th International Conference on Automated Deduction*, LNCS, pages 285–289. Springer-Verlag, 2002.

10. Simon Colton. Making Conjectures about Maple Functions. In *Proceedings of the Joint International Conferences on Artificial Intelligence, Automated Reasoning, and Symbolic Computation*, pages 259–274. Springer-Verlag, 2002.

11. Thierry Coquand, Catarina Coquand, Thomas Hallgren, and Aarne Ranta. The Alfa Home Page, 2001.
http://www.md.chalmers.se/~hallgren/Alfa/.

12. David Delahaye and Micaela Mayero. `Field`: une procédure de décision pour les nombres réels en Coq. In *Journées Francophones des Langages Applicatifs, Pontarlier (France)*. INRIA, January 2001.
ftp://ftp.inria.fr/INRIA/Projects/coq/David.Delahaye/papers/JFLA2000-Field.ps.gz.

13. Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Theorem Proving Modulo. Technical Report RR-3400, INRIA-Rocquencourt, France, April 1998.
ftp://ftp.inria.fr/INRIA/publication/RR/RR-3400.ps.gz.

14. M. J. C. Gordon and T. F. Melham. *Introduction to* HOL*: a Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

15. John Harrison. Metatheory and Reflection in Theorem Proving: a Survey and Critique. Technical Report CRC-053, SRI Cambridge, UK, 1995.
http://www.cl.cam.ac.uk/users/jrh/papers/reflect.dvi.gz.

16. John Harrison and Laurent Théry. A Skeptic's Approach to Combining HOL and Maple. *Journal of Automated Reasoning*, 21:279–294, 1998.

17. Michael Kohlhase. OMDoc: an Open Markup Format for Mathematical Documents. Technical Report SEKI SR-00-02, Universität des Saarlandes, Saarebrucken (DE), 2000.
http://www.mathweb.org/omdoc.

18. Martijn Oostdijk and Herman Geuvers. Proof by Computation in the Coq System. *Theoretical Computer Science*, 272(1–2):293–314, 2002.

19. Sam Owre, Natarajan Shankar, and John Rushby. PVS: A Prototype Verification System. In *Proceedings of CADE 11, Saratoga Springs, New York*, June 1992.

20. Larry Paulson and Tobias Nipkow. The Isabelle Home Page, 2003.
http://www.cl.cam.ac.uk/Research/HVG/Isabelle/index.html.

21. Randy Pollack. *The Theory of* Lego*: A Proof Checker for the Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1994. http://www.dcs.ed.ac.uk/home/rap/export/thesis.ps.gz.

22. Harald Rueß. Computational Reflection in the Calculus of Constructions and its Application to Theorem Proving. In R. Hindley, editor, *Proceedings for the Third International Conference on Typed Lambda Calculus and Applications (TLCA'97)*, Lecture Notes in Computer Science, Nancy, France, April 1997. Springer-Verlag.

23. Volker Sorge. Non-Trivial Computations in Proof Planning. In Hélène Kirchner and Christophe Ringeissen, editors, *Frontiers of Combining Systems : Third International Workshop, FroCoS 2000*, volume 1794 of *LNCS*, pages 121–135, Nancy, France, 22–24 March 2000. Springer-Verlag, Berlin, Germany.

24. The Coq Development Team. *The Coq Proof Assistant Reference Manual Version 7.3*. INRIA-Rocquencourt, May 2002. http://coq.inria.fr/doc-eng.html.

25. the Ωmega Group. Ωmega: Towards a Mathematical Assistant. In *Proceedings of CADE-14*, volume 1249 of *LNAI*, pages 252–255. Springer-Verlag, 1997.

D.4 PAPER 4: THE ZENON AUTOMATED THEOREM PROVER

This paper is related to Chapter 3 and has been published in [24].

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

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

$$\frac{P \wedge Q}{P, Q}\ \alpha_{\wedge} \qquad \frac{\neg(P \vee Q)}{\neg P, \neg Q}\ \alpha_{\neg\vee} \qquad \frac{\neg(P \Rightarrow Q)}{P, \neg Q}\ \beta_{\neg\Rightarrow}$$

$$\frac{P \vee Q}{P \mid Q}\ \beta_{\vee} \qquad \frac{\neg(P \wedge Q)}{\neg P \mid \neg Q}\ \beta_{\neg\wedge} \qquad \frac{P \Rightarrow Q}{\neg P \mid Q}\ \beta_{\Rightarrow}$$

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

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

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

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:

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

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:

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

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:

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

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

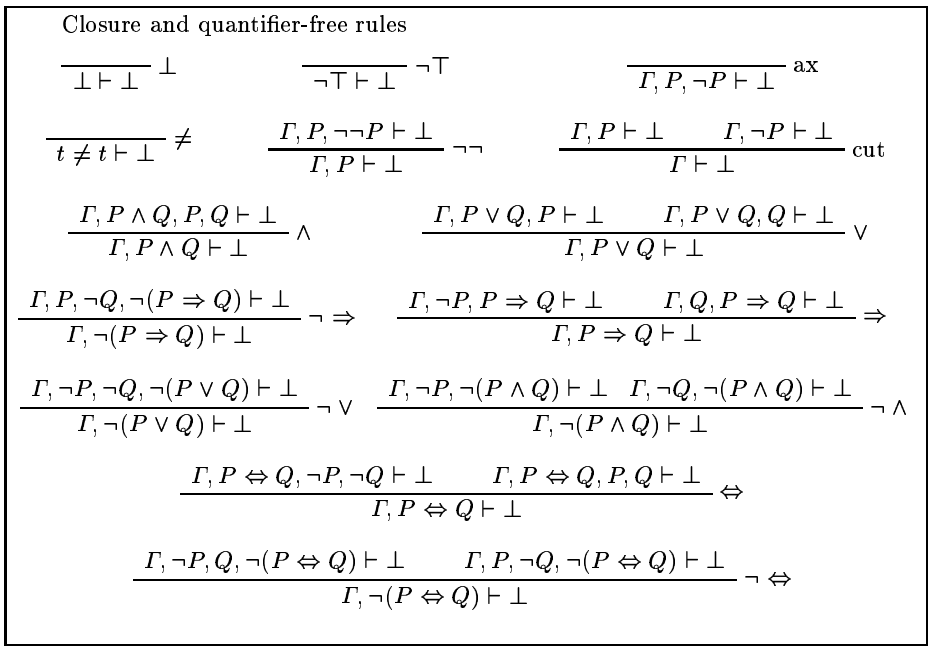

Fig. 3. LLproof rules (part 1)

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

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

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

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

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

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

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

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

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

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

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 short-cut), 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:

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

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:

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

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/`.

D.5 PAPER 5: FROM FOCAL TO UML

This paper is related to Chapter 4 and has been published in [57].

Noname manuscript No.
(will be inserted by the editor)

A Formal and Sound Transformation from Focal to UML

An Application to Airport Security Regulations

David Delahaye · Jean-Frédéric Étienne · Véronique Viguié Donzeau-Gouge

the date of receipt and acceptance should be inserted later

Abstract We propose an automatic transformation of Focal specifications to UML class diagrams. The main motivation for this work lies within the framework of the EDEMOI project, which aims to integrate and apply several requirements engineering and formal methods techniques to analyze airport security regulations. The idea is to provide a graphical documentation of formal models for developers, and in the long-term, for certification authorities. The transformation is formally described and an implementation has been designed. We also show how the soundness of our approach can be achieved.

Keywords Formal Methods · Graphical Documentation · Focal · UML · Airport Security Regulations

1 Introduction

Even though formal methods offer a systematic approach for verification, the validation process still relies on a high degree of interaction between the various stake-holders (developers, customers, end-users, certification authorities, etc) involved in a critical project. In addition, the use of formal methods requires a certain level of expertise in mathematics, which usually hinders communication. In fact, the mathematical notations used are often too obscure for inexperienced users to properly understand the exact meaning. As a result, the validation of requirements is difficultly achievable. This may even jeopardize the entire project as misinterpretations or specification errors may lead to the validation of a totally wrong implementation.

A widely adopted solution to these problems is the integration of formal and graphical specifications. In general, the use of graphical notations is quite useful when interacting with end-users. In fact, these tend to be more intuitive and are easier to grasp than their formal (or textual) counterparts. During the last few years, UML [16] has emerged as a standard in industry for modeling software systems. It provides a set of graphical constructs, which enables the modeling of systems in an object-oriented style. Currently, it is supported by a wide variety of tools, ranging from analysis, testing, simulation to code generation and transformation. Interoperability between these tools is generally achieved by exporting the UML models using the XMI interchange format.

There have been several researches devoted to establishing the link between UML and formal methods. One of the approaches that has been largely studied is the translation of UML diagrams into formal specifications [7, 11, 12], which attempts to benefit from the formal methods tools and techniques while still having control over the UML-based industrial practice. The converse approach is a rather new area of interest [9]. It is here considered to generate UML models as a means to provide a graphical documentation for Focal specifications.

The main motivation for this work lies within the framework of the EDEMOI[1] [14] project, which aims

D. Delahaye
CEDRIC/CNAM, Paris, France
E-mail: David.Delahaye@cnam.fr

J.-F. Étienne
CEDRIC/CNAM, Paris, France
E-mail: etiennje@cnam.fr

V. Viguié Donzeau-Gouge
CEDRIC/CNAM, Paris, France
E-mail: donzeau@cnam.fr

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

to integrate and apply several requirements engineering and formal methods techniques to analyze airport security regulations. For this project, we used Focal to realize the formal models of two regulations, namely the international standard Annex 17 and the European directive Doc 2320. The formalization is described in [3], while the certification part is presented in [4]. Within the project, the purpose of the UML diagrams is twofold. First, to provide a graphical documentation of the formal models produced for developers. Second, to generate higher-level views of the formal models that would be more appealing to certification authorities.

For our concern, the choice of UML as a graphical notation mainly resides in the fact that most of the Focal design features can seamlessly be represented in UML. The creation of a domain specific language for Focal could be a better approach, as it avoids us from having to deal with the intricacies of the UML semantics. In fact, text-to-model tools [8], such as xText or TCS, generally facilitates such a process, whereby the target language is taken as input and the corresponding metamodel, parser and editor is generated as output. However, we still have to develop a graphical concrete syntax for each concept. The corresponding semantics might be intuitive to developers but not necessarily to end-users or certification authorities (which is our long-term objective). Finally, the choice for UML also allows us to have access to a wide variety of tools ranging from analysis to code generation and transformation. For instance, the UML models produced can be used to map Focal specifications to other object-oriented languages, e.g. Java or C#.

This paper is complementary to the work presented in [5] and completes the formal schema established for the translation of Focal specifications into UML diagrams. Here, the objective is to provide a graphical documentation for developers. Our major concern is not only to make our transformation automatic but also to prove the soundness of our approach. In this paper, we refine the abstract syntax proposed in [5] for a subset of the UML 2.1 static structure constructs [16]. The new syntax intends to facilitate reasoning. We here also consider all the different aspects of the Focal specification language and show how the UML metamodel can be tailored to consider some of its semantic specificities. We also describe how the soundness of our transformation can be achieved. This consists in showing that the UML model generated from a well-typed Focal specification preserves both the well-formedness rules of the UML metamodel and the constraints specified in the UML profile defined on purpose. Through this work, we also contribute to the formalization of the semantics relative to the template binding construct.

The paper is organized as follows: first, we present the Focal specification language; next, we propose a formal description for a subset of the UML 2.1 static structure constructs; we then show how the UML metamodel can be extended via the profile mechanism (lightweight extension) to cater for the semantic specificities of the Focal language; we afterwards formally describe our transformation rules and expose how the soundness of our approach can be achieved; finally, we introduce our implementation and illustrate our transformation with a concrete example.

2 The Focal Environment

2.1 What is Focal?

Focal[2] [6, 15], initiated by T. Hardin and R. Rioboo with S. Boulmé, is a language in which it is possible to build certified applications step by step, going from abstract specifications, called species, to concrete implementations, called collections. These different structures are combined using inheritance and parameterization, inspired by object-oriented programming. Moreover, each of these structures is equipped with a carrier set, providing a typical algebraic specification flavor. Next, V. Prevosto developed a compiler for this language, able to produce OCaml code for execution, Coq code for certification, but also FocDoc code [13] for documentation. More recently, D. Doligez provided a first-order automated theorem prover, called Zenon [1], which helps the user to complete his/her proofs in Focal through a declarative-like proof language. This automated theorem prover can produce pure Coq proofs, which are reinserted in the Coq specifications generated by the Focal compiler and fully verified by Coq.

2.2 Specification: Species

The first main notion of the Focal language is the structure of *species*, which corresponds to the highest level of abstraction in a specification. A species can roughly be seen as a list of attributes of three kinds:

- the carrier type, called *representation*, which is the type of the entities that are manipulated by the functions of the species; the representation can be either abstract or concrete;
- the functions, which denote the operations allowed on the entities of the representation; the functions can be either *definitions* (when a body is provided) or *declarations* (when only a type is given);

[2] `http://focal.inria.fr/`.

– the properties, that must be verified by any further implementation of the species; the properties can be either simply *properties* (when only the proposition is given) or *theorems* (when a proof is also provided).

The syntax of a species is the following:

```
species <name> =

  rep [= <type>]; (* abstract/concrete
                     representation *)

  sig <name> in <type>; (* declaration *)
  let <name> = <body>;  (* definition *)

  property <name> : <prop>; (* property *)
  theorem <name> : <prop>   (* theorem *)
  proof : <proof>;

end
```

where <name> is simply a given name, <type> a type expression (mainly typing of core-ML without polymorphism but with concrete data types), <body> a function body (mainly core-ML with conditional, pattern-matching and recursion), <prop> a (first-order) proposition and <proof> a proof (expressed in a declarative style and given to Zenon). In the type language, the specific expression "self" refers to the type of the representation and may be used everywhere except when defining a concrete representation.

As said previously, species can be combined using (multiple) inheritance, which works as expected. It is possible to define functions that were previously only declared or to prove properties which had no provided proof. It is also possible to redefine functions previously defined or to reprove properties already proved. However, the representation cannot be redefined and functions as well as properties must keep their respective types and propositions all along the inheritance path. Another way of combining species is to use parameterization. Species can be parameterized either by other species or by entities from species. If the parameter is a species, the parameterized species only has access to the interface of this species, i.e. only its abstract representation, its declarations and its properties. These two features complete the previous syntax definition as follows:

```
species <name> (<name> is <name>[(<pars>)],
                <name> in <name>, ...)
        inherits <name>, <name> (<pars>),
                 ... = ...
end
```

where <pars> is a list of <name>, which denotes the names used as effective parameters. When the parameter is a species parameter declaration, the "is" keyword is used. When it is an entity parameter declaration, the "in" keyword is used.

2.3 Implementation: Collection

The other main notion of the Focal language is the structure of *collection*, which corresponds to the implementation of a specification. A collection implements a species in such a way that every attribute becomes concrete: the representation must be concrete, functions must be defined and properties must be proved. If the implemented species is parameterized, the collection must also provide implementations for these parameters: either a collection if the parameter is a species or a given entity if the parameter denotes an entity of a species. Moreover, a collection is seen (by the other species and collections) through its corresponding interface; in particular, the representation is an abstract data type and only the definitions of the collection are able to manipulate the entities of this type. Finally, a collection is a terminal item and cannot be extended or refined by inheritance. The syntax of a collection is the following:

```
collection <name> implements <name>
           (<pars>) = ... end
```

3 UML Syntax

In order to establish a formal framework for our transformation, we propose in [5] an abstract syntax for a subset of the UML 2.1 static structure constructs [16]. The syntax was mainly derived from the UML 2.1/XMI schema to reflect as much as possible our implementation. In this section, we present a new syntax that hides some of the complexities inherent to the UML metamodel and thus less dependent on the XMI format. This not only allows us to increase the readability of our transformation rules but also to facilitate reasoning. Another less tedious approach can be to make use of a text-to-model tool [8], e.g. xText or TCS, to obtain a metamodel of the Focal specification language instead of defining an abstract syntax for UML. The automatic transformation from Focal to UML may then be realized at a metamodel level through the use of a model-to-model transformation language [10], such as ATL or QVT. Nevertheless, even though such an approach can be considered during the implementation phase, it does not allow us to prove the soundness of our transformation.

The new syntax is shown in Figure 1 and is given using a BNF-like notation, where: terminal symbols are written in bold, while non-terminal ones are in italic; square brackets [...] are used to denote optional components, while curly brackets {...} denote grouping; a trailing star sign * denotes zero, one or several occur-

rences, while a trailing plus sign $^+$ denotes one or several occurrences; and the non-terminal symbol *ident* is used to designate the identifier of each nameable UML construct. In our syntax, anonymous bound classes are denoted using the same notation as defined for template bindings. The class type *class-type* is defined accordingly to reflect that these classes may also be referenced as type.

4 From Focal to UML

4.1 Extending the UML Metamodel

In order to properly visualize Focal models using UML notations, there is a need to extend the UML metamodel to cater for the semantic specificities of the Focal language. These extensions are realized through the creation of a profile, whereby appropriate stereotypes are defined to reflect the semantics of each Focal construct, namely «Species», «Collection», «FocalType», «Method», «In», «Is», «ParameterizedInheritance», «Inheritance» and «Implements». To validate our transformation, we also encode the semantics relative to the template binding construct via the introduction of intermediate stereotypes declared as *required* (i.e. mandatory when the corresponding profile is applied). Here, we base ourselves on the OCL formalization realized by Caron et al in [2], which we extend to handle nested *bound* classes and inherited members. To formally represent the application of our profile to a UML model, the abstract syntax given in Figure 1 is slightly extended. In fact, each keyword (e.g. **class inherits**, etc) representing a given UML construct is replaced by a non-terminal node to reflect the stereotypes that can be applied to the corresponding construct. For example, keyword **class** is replaced by the non-terminal node *class-head*, which is defined as follows:

$$\textit{class-head} ::= \textbf{class} \mid \textbf{focalType} \mid \textbf{species} \mid \textbf{collection}$$

The syntax is also extended to consider the attributes characterizing each stereotype (e.g., see attributes *substitutes* and *bound* of stereotype «ParameterizedInheritance» in rule $[\![\Gamma_h]\!]^{\mathrm{DE}}_{\Gamma,\mathcal{P},rep,s}$ of Figure 3).

4.2 Transformation Rules

Despite their similarities, Focal species and UML classes are based on two different concepts. In Focal, the functions defined in a species are intended to manipulate entities of a given representation, which are static items having a unique value. Hence, we model a species as an abstract factory class (stereotyped with «Species»), which defines an interface for manipulating immutable value objects of a given type. Let S denote a species: $S =$ **species** s $(\mathcal{P})$ **inherits** $\Gamma_h = rep$; $\mathcal{M}$; $\mathcal{R}$ **end**, where s is the name of the species, $\mathcal{P}$ a list of parameters, Γ_h a list of species from which we inherit, rep the representation declaration, $\mathcal{M}$ the declared/defined functions, and $\mathcal{R}$ the properties/theorems defined in S. Given the context Γ, in which S is well typed, the corresponding UML model is obtained by applying the transformation rule denoted by $[\![S]\!]_\Gamma$ in Figures 2 and 3. Our transformation captures every aspect of the Focal specification language. Due to space limitations, we here only focus on the representation, parameter declarations and inheritance. In Figures 2 and 3, $\perp$ is used to denote undefinedness and "$\cdot$" the concatenation operator on identifiers. We also write $c :: \mathit{Self}$ to designate the inner class *Self* defined within c.

The representation of a given species (rule $[\![rep]\!]^{\mathrm{PA}}_{\Gamma,s}$), is characterized by two type parameters T and TSelf (stereotyped with «FocalType»), where T represents the type of the entities and TSelf the class in which T is encapsulated. The latter is used to represent the type of the immutable value objects. Parameter T is generated only if the representation is abstract. The correlation between T and TSelf is specified by the factory methods makeSelf and getRep, which are introduced only if the given species is a root node (rule $[\![rep]\!]^{\mathrm{OP}}_{\Gamma,\mathcal{P},\Gamma_h,\alpha}$).

Inheritance between species is modeled as a dependency relation stereotyped with «ParameterizedInheritance» (rule $[\![\Gamma_h]\!]^{\mathrm{DE}}_{\Gamma,\mathcal{P},rep,s}$), which specifies an intermediate bound class that instantiates the formal parameters of the target factory class. The specializing class inherits from this bound class via a generalization relation stereotyped with «Inheritance» (rule $[\![\Gamma_h]\!]^{\mathrm{GE}}_{\Gamma,s}$).

Function declarations are translated into class operations stereotyped with «Method», which are defined as function object types (using the parameterized class Fun). As for property/theorem declarations, they are represented by UML constraints specified as invariants.

Collections are modeled as concrete singleton factory classes stereotyped accordingly. This allows us to ensure that no method invocation is possible on species, as is the case in Focal. The abstraction of the concrete representation is achieved through the declaration of an inner class Self. This class is declared with a private constructor and a private read-only attribute to obtain the desired encapsulation. The type of the immutable value objects is fixed definitely through the use of the «Implements» stereotype. In essence, the type parameters T and TSelf are instantiated such that T is substituted for a concrete type and TSelf is substituted for the inner class Self created on purpose.

Um ::= *decl**
decl ::= *class* | *constraint* | *opaque* | *dep*
class ::= *option* **class** *ident* [(*cl-param* {, *cl-param*}*)]
[**binds** *bind* {, *bind*}*] [**inherits** *ident* {, *ident*}*] =
constraint* *attr** *opr** *class** **end**
option ::= [*visibility*] [**final** | **abstract**]
visibility ::= **public** | **private** | **protected**
cl-param ::= *ident* : **class** [> *class-type*] | *ident* : **opaqueExpr** [> *type*]
class-type ::= *ident* | *bind*
type ::= *class-type* | **Integer** | **Boolean** | **UnlimitedNatural** | **String**
bind ::= *ident*<*subs* [, *subs**]>
subs ::= *ident* → *ident*
opr ::= *option* [**static**] **operation** *ident* ([*op-param* {, *op-param*}*])
[**redefines** *ident* {, *ident*}*]
op-param ::= *dir ident* [: *type*]
dir ::= **in** | **inout** | **out** | **return**
attr ::= *at-option* **property** *ident* [: *type*] [**redefines** *ident* {, *ident*}*]
at-option ::= [*visibility*] [**static**] [**final**] [**readOnly**]
opaque ::= [*visibility*] **opaqueExpr** *ident* =
[*body*] [: *type*] [**in** *lang*] **end**
constraint ::= [*visibility*] **constraint** *ident*
[**restricts** *ident* {, *ident*}*] = *opaque* **end**
dep ::= [*visibility*] **dependency** *ident*
(*ident* {, *ident*}* --→ *ident* {, *ident*}*)

Fig. 1 Syntax for UML Static Constructs

4.3 Implementation

Our implementation consists of two parts. In the first part, we define a UML profile for the Focal specification language through the use of the UML2 Eclipse plug-in. This plug-in provides an implementation of the UML 2.1 metamodel and its integrated OCL checker allows us to validate the constraints defined in our profile. The ability to specify statically defined profiles also facilitates the definition of the operations and derived attributes characterizing each stereotype constituting our profile. This step is essential as it provides the necessary tool to validate the UML models to which our profile is applied. In fact, each OCL constraint specified in our profile is parsed and evaluated at runtime. This mechanism offers a convenient way to validate the soundness of our transformation. The second part concerns the development of an XSLT stylesheet that specifies the rules to transform a Focal specification generated in FocDoc format [13] (an XML schema used by the compiler for documentation) into a UML model expressed in the XMI interchange format.

5 Soundness

In this section, we present how the soundness of our transformation can be established. Here, by soundness, we mean that the transformation of a well-typed Focal specification results in a well-formed UML model. We write Δ_p to denote the UML profile established for the Focal specification language. To simplify, we consider Δ_p to be a list of UML constraints $\Phi_1, \ldots, \Phi_m$. Symbol $\mathcal{U}$ is used to denote a UML model, which represents a list of construct declarations $\mathcal{D}_1, \ldots, \mathcal{D}_n$ as described by the abstract syntax shown in Figure 1. We write $\Delta_p(\mathcal{D}_i)$ to denote the list of constraints that relates to the current declaration $\mathcal{D}_i$ when profile Δ_p is applied. Finally, we write Δ_m to denote the UML metamodel, which is considered to be a list of UML constraints $\Omega_1, \ldots, \Omega_q$. Similarly, $\Delta_m(\mathcal{D}_i)$ denotes the list of constraints relative to a given construct declaration $\mathcal{D}_i$. The soundness theorem is the following (due to space restrictions, we omit the corresponding proof):

Theorem 1 (Soundness) *Let $\mathcal{F}$ a well-typed Focal specification within the context Γ s.t. $\mathcal{F} = \mathcal{E}_1, \ldots, \mathcal{E}_n$ and where each $\mathcal{E}_i$ is either a species S or a collection C. Let $\mathcal{U}$ be the UML model obtained when applying the transformation rule $[\![\mathcal{F}]\!]_\Gamma$. Our transformation is sound if the following conditions hold:*

1. $\Delta_p, \Delta_m \nvdash \bot$;
2. *For each $\mathcal{D}_i \in \mathcal{U}$,*
   - $\forall \Omega_j \in \Delta_m(\mathcal{D}_i)$, $\Gamma \vdash \Omega_j$;
   - $\forall \Phi_k \in \Delta_p(\mathcal{D}_i)$, $\Gamma \vdash \Phi_k$.

The first condition specifies that the constraints within profile Δ_p must not introduce any inconsistency w.r.t. the well-formedness rules of the UML metamodel Δ_m. The second condition states that $\mathcal{U}$ must satisfy both the well-formedness rules of the UML metamodel and the constraints within profile Δ_p.

The previous theorem essentially states that typing is preserved from Focal to UML (even if the well-

Focal Species:

$$[\![S]\!]_\Gamma = \begin{cases} \textbf{public abstract species } s\ [\![\mathcal{P}]\!]^{\text{RE}}_{\Gamma,rep,s}\ [\![\Gamma_h]\!]^{\text{GE}}_{\Gamma,s} = [\![\mathcal{R}]\!]_{\Gamma,s}\ [\![\mathcal{P}]\!]^{\text{AT}}_{\Gamma,s}\ [\![rep]\!]^{\text{OP}}_{\Gamma,\mathcal{P},\Gamma_h,\perp}\ [\![\mathcal{M}]\!]_{\Gamma,\mathcal{P},\Gamma_h,s,\perp}\ \textbf{end} \\ [\![\Gamma_h]\!]^{\text{DE}}_{\Gamma,\mathcal{P},rep,s} \end{cases}$$

Representation and Parameter Declarations:

Given $\mathcal{P} = p_1 \odot I_1, \ldots, p_n \odot I_n$, with $\odot \in \{\textbf{is}, \textbf{in}\}$: $[\![\mathcal{P}]\!]^{\text{RE}}_{\Gamma,rep,s} = ([\![p_1 \odot I_1]\!]_{\Gamma,P_1,s}\ ,\ \ldots,\ [\![p_n \odot I_n]\!]_{\Gamma,P_n,s}\ ,\ [\![rep]\!]^{\text{PA}}_{\Gamma,s})$ with $P_1 = \emptyset$ and $P_n = p_1 \odot I_1, \ldots, p_{n-1} \odot I_{n-1}$

$$[\![p_i \odot I_i]\!]_{\Gamma,P_i,s} = [\![e_i\ \textbf{in}\ \tau]\!]_{\Gamma,P_i,s} \mid [\![c_i\ \textbf{is}\ S_i]\!]_{\Gamma,P_i,s}$$

$$[\![e_i\ \textbf{in}\ \tau]\!]_{\Gamma,P_i,s} = e_i : \textbf{opaqueExpr in } [\![\tau]\!]^{\text{type}}_{\Gamma,P_i,\perp}$$

$$[\![c_i\ \textbf{is}\ S_i]\!]_{\Gamma,P_i,s} = \begin{cases} \left.\begin{array}{l} c_i \cdot T : \textbf{focalType}, \\ \mathsf{selfType}_{\Gamma,P_i,c_i\ \textbf{is}\ S_i} \end{array}\right\} & \text{if } \ \Gamma(S_i).rep = \perp \\ \mathsf{selfType}_{\Gamma,P_i,c_i\ \textbf{is}\ S_i} & \text{otherwise} \end{cases}$$

$$[\![rep]\!]^{\text{PA}}_{\Gamma,s} = \begin{cases} T : \textbf{focalType},\ TSelf : \textbf{focalType} & \text{if } \ rep = \perp \\ TSelf : \textbf{focalType} & \text{otherwise} \end{cases}$$

$$\mathsf{selfType}_{\Gamma,P_i,c_i\ \textbf{is}\ S_i} = \begin{cases} c_i \cdot Self : \textbf{focalType}, \\ c_i : \textbf{collection is } [\![S_i]\!]^{\text{const}}_{\Gamma,P_i,c_i} \end{cases}$$

$$[\![S_i]\!]^{\text{const}}_{\Gamma,P_i,c_i} = [\![s_{q_i}]\!]^{\text{const}}_{\Gamma,P_i,c_i} \mid [\![s_{q_i}(a_1,\ldots,a_f)]\!]^{\text{const}}_{\Gamma,P_i,c_i} \qquad [\![s_{q_i}]\!]^{\text{const}}_{\Gamma,P_i,c_i} = \begin{cases} s_{q_i} < T \to c_i \cdot T,\ TSelf \to c_i \cdot Self >_{\mathbf{s}} & \text{if } \Gamma(s_{q_i}).rep = \perp \\ s_{q_i} < TSelf \to c_i \cdot Self >_{\mathbf{s}} & \text{otherwise} \end{cases}$$

$$[\![s_{q_i}(a_1,\ldots,a_f)]\!]^{\text{const}}_{\Gamma,P_i,c_i} = \begin{cases} \left.\begin{array}{l} s_{q_i}\ [\![a_1]\!]_{\Gamma,P_i,\mathcal{S}_1,p_{q_1}\odot I_{q_1}}, \ldots,\ [\![a_f]\!]_{\Gamma,P_i,\mathcal{S}_f,p_{q_f}\odot I_{q_f}}, \\ \quad < T \to c_i \cdot T,\ TSelf \to c_i \cdot Self >_{\mathbf{s}} \end{array}\right\} & \text{if } \Gamma(s_{q_i}).rep = \perp \\ \left.\begin{array}{l} s_{q_i}\ [\![a_1]\!]_{\Gamma,P_i,\mathcal{S}_1,p_{q_1}\odot I_{q_1}}, \ldots,\ [\![a_f]\!]_{\Gamma,P_i,\mathcal{S}_f,p_{q_f}\odot I_{q_f}}, \\ \quad < TSelf \to c_i \cdot Self >_{\mathbf{s}} \end{array}\right\} & \text{otherwise} \end{cases}$$

with $p_{q_1} \odot I_{q_1}, \ldots, p_{q_f} \odot I_{q_f} = \Gamma(s_{q_i}).\mathcal{P}$, $\mathcal{S}_1 = \emptyset$ and $\mathcal{S}_f = \{(a_1, p_{q_1}), \ldots, (a_{f-1}, p_{q_{f-1}})\}$

$$[\![a_k]\!]_{\Gamma,P_i,\mathcal{S}_k,p_{q_k}\odot I_{q_k}} = \begin{cases} [\![e_k]\!]_{\Gamma,P_i,\mathcal{S}_k,e_{q_k}\ \textbf{in}\ \tau_{q_k}} & \text{if } a_k = e_k\ \wedge\ p_{q_k} \odot I_{q_k} = e_{q_k}\ \textbf{in}\ \tau_{q_k} \\ [\![c_k]\!]_{\Gamma,P_i,c_{q_k}\ \textbf{is}\ S_{q_k}} & \text{if } a_k = c_k\ \wedge\ p_{q_k} \odot I_{q_k} = c_{q_k}\ \textbf{is}\ S_{q_k} \end{cases}$$

$$[\![e_k]\!]_{\Gamma,P_i,\mathcal{S}_k,e_{q_k}\ \textbf{in}\ \tau_{q_k}} = \begin{cases} < e_{q_k} \to e_j > & \text{if } \exists\ e_j\ \textbf{in}\ \tau_j \in P_i\ s.t.\ e_j = e_k \\ < e_{q_k} \to id_k > & \text{otherwise} \end{cases}$$

where id_k is a new identifier s.t.,

$\textbf{opaqueExpr}\ id_k = e_k : [\![\tau_{q_k}]\!]^{\text{subs}}_{\Gamma,P_i,\mathcal{S}_k}\ \textbf{in}$ "Focal" $\textbf{end}$

$$[\![c_k]\!]_{\Gamma,P_i,c_{q_k}\ \textbf{is}\ S_{q_k}} = \begin{cases} < c_{q_k} \cdot T \to \mathsf{repBind}_{\Gamma,P_i,\beta,c_{q_k},c_k} > \ ,\ \mathsf{selfBind}_{\beta,c_{q_k},c_k} & \text{if } \Gamma(S_{q_k}).rep = \perp \\ \mathsf{selfBind}_{\beta,c_{q_k},c_k} & \text{otherwise} \end{cases}$$

$$\text{with } \beta = \begin{cases} c_j\ \textbf{is}\ S_j & \text{if } \exists\ c_j\ \textbf{is}\ S_j \in P_i\ s.t.\ c_j = c_k \\ \perp & \text{otherwise} \end{cases}$$

$$\mathsf{selfBind}_{\beta,c_{q_k},c_k} = \begin{cases} < c_{q_k} \cdot Self \to c_k :: Self,\ c_{q_k} \to c_k > & \text{if } \beta = \perp \\ < c_{q_k} \cdot Self \to c_j \cdot Self,\ c_{q_k} \to c_j > & \text{if } \beta = c_j\ \textbf{is}\ S_j \end{cases}$$

$$\mathsf{repBind}_{\Gamma,\beta,P_i,c_{q_k},c_k} = \begin{cases} c_j \cdot T & \text{if } \beta = c_j\ \textbf{is}\ S_j\ \wedge\ \Gamma(S_j).rep = \perp \\ [\![\tau_j]\!]^{\text{type}}_{\Gamma,P_i,\perp} & \text{if } \beta = c_j\ \textbf{is}\ S_j\ \wedge\ \Gamma(S_j).rep = \tau_j \\ [\![\Gamma(c_k).rep]\!]^{\text{type}}_{\Gamma,\emptyset,c_k} & \text{otherwise} \end{cases}$$

Factory Methods for the Representation:

$$[\![rep]\!]^{\text{OP}}_{\Gamma,\mathcal{P},\Gamma_h,\alpha} = \begin{cases} \textbf{protected abstract method } makeSelf\ (\textbf{in } x : \mathsf{repType}_{\Gamma,\mathcal{P},rep}\ ,\ \textbf{return } y : \mathsf{selfBind}_\alpha) \\ \textbf{protected abstract method } getRep\ (\textbf{in } x : \mathsf{selfBind}_\alpha\ ,\ \textbf{return } y : \mathsf{repType}_{\Gamma,\mathcal{P},rep}) \end{cases}$$

when $\Gamma_h = \emptyset$

$$\mathsf{repType}_{\Gamma,\mathcal{P},rep} = \begin{cases} T & \text{if } rep = \perp \\ [\![\tau]\!]^{\text{type}}_{\Gamma,\mathcal{P},\perp} & \text{if } rep = \tau \end{cases} \qquad \mathsf{selfBind}_\alpha = \begin{cases} TSelf & \text{if } \alpha = \perp \\ c :: Self & \text{if } \alpha = c \end{cases}$$

Types:

Given a list of parameter declarations P and α either referencing a collection or set to $\perp$:

$$[\![\tau]\!]^{\text{type}}_{\Gamma,P,\alpha} = [\![c]\!]^{\text{type}}_{\Gamma,P,\alpha} \mid [\![t]\!]^{\text{type}}_{\Gamma,P,\alpha} \mid [\![\tau_1 \to \tau_2]\!]^{\text{type}}_{\Gamma,P,\alpha} \mid [\![\tau_1 * \tau_2]\!]^{\text{type}}_{\Gamma,P,\alpha} \mid [\![\textbf{self}]\!]^{\text{type}}_{\Gamma,P,\alpha}$$

$$[\![c]\!]^{\text{type}}_{\Gamma,P,\alpha} = \begin{cases} c_j \cdot Self & \text{if } \exists\ c_j\ \textbf{is}\ S_j \in P\ s.t.\ c_j = c \\ c :: Self & \text{otherwise} \end{cases} \qquad [\![t]\!]^{\text{type}}_{\Gamma,P,\alpha} = t \qquad [\![\textbf{self}]\!]^{\text{type}}_{\Gamma,P,\alpha} = \begin{cases} TSelf & \text{if } \alpha = \perp \\ c :: Self & \text{if } \alpha = c \end{cases}$$

$$[\![\tau_1 \to \tau_2]\!]^{\text{type}}_{\Gamma,P,\alpha} = Fun < A \to [\![\tau_1]\!]^{\text{type}}_{\Gamma,P,\alpha},\ B \to [\![\tau_2]\!]^{\text{type}}_{\Gamma,P,\alpha} >_{\mathbf{f}} \qquad [\![\tau_1 * \tau_2]\!]^{\text{type}}_{\Gamma,P,\alpha} = Pair < A \to [\![\tau_1]\!]^{\text{type}}_{\Gamma,P,\alpha},\ B \to [\![\tau_2]\!]^{\text{type}}_{\Gamma,P,\alpha} >_{\mathbf{f}}$$

Fig. 2 Transformation Rules: Focal to UML (1)

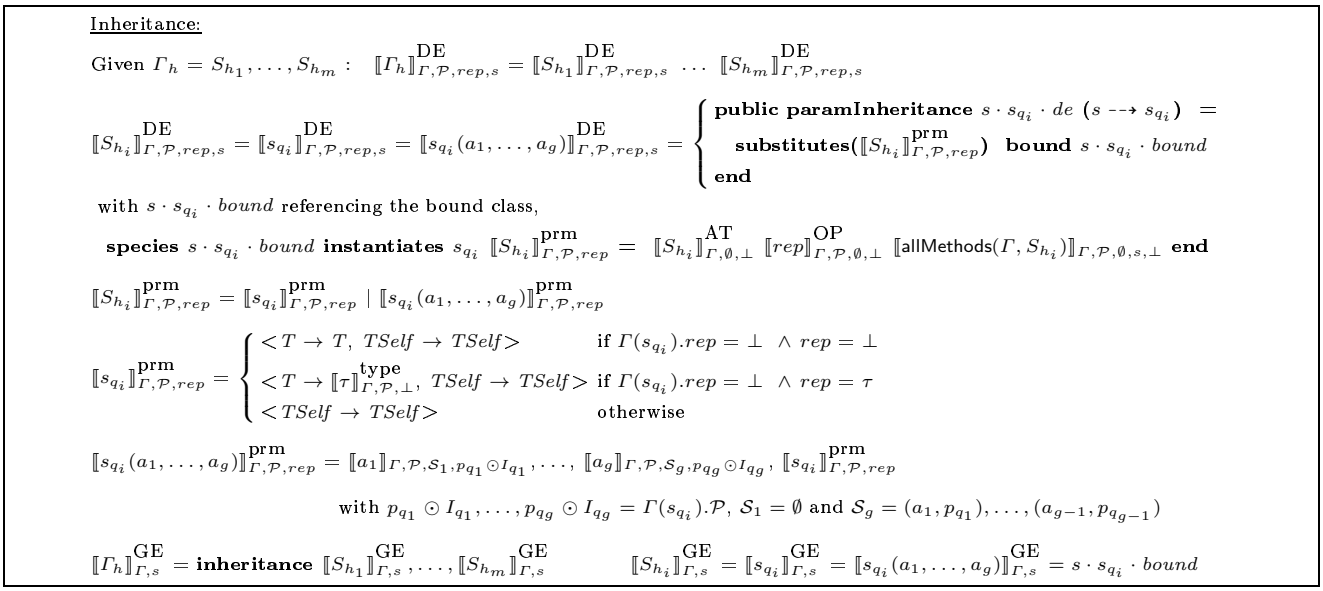

Fig. 3 Transformation Rules: Focal to UML (2)

formedness rules are said to characterize the semantics of UML). Another form of soundness, not considered in this paper, would be to establish that the semantics of Focal is also preserved by the transformation, which is equivalent to show that there exists a model of the UML metamodel (together with the profile), for which the well-formedness rules are correct and which is compatible with a model of Focal.

6 An Application Example

To illustrate our transformation process, we consider a relatively concise example extracted from the formalization realized within the EDEMOI project. This concerns the specification established for cabin persons. The corresponding Focal species is defined as follows:

```
species cabinPerson (cb is cabinBaggage) =
  rep;
  sig equal in self → self → bool;
  sig identityVerified in self → bool;
  sig cabinBaggage in self → cb;
  property equal_reflexive:
    all x in self, !equal (x, x); ...
end
```

It can be observed that cabinPerson is a parameterized species and its representation is left undefined. We also assume that the representation of species cabinBaggage is still abstract. To give an example of inheritance and show how the abstraction of a concrete representation is handled during the transformation process, we also introduce collection cabinPerson_col, which provides an implementation for species cabinPerson:

```
collection cabinPerson_col
           implements cabinPerson (bag) =

  rep = string * bag * bool;
  let name (s in self) in string = #first (s);
  let cabinBaggage (s in self) in bag =
    #first (#scnd (s));
  let identityVerified (s in self) in bool =
    #scnd (#scnd (s)); ...
end
```

In this collection, the representation is specified as a triple, with the functions name, cabinBaggage and identityVerified defined accordingly. In the "implements" clause, species cabinPerson is instantiated with bag, which is a collection derived from cabinBaggage.

Now, by applying the transformation rules described in Section 4, the UML classes shown in Figure 4 (using the corresponding graphical visualization) are obtained, where we write TSelf → Bool for the bound class Fun<TSelf, Bool>.

7 Conclusion

In this paper, we present a formal and sound framework for the transformation of Focal specifications into UML models, with the objective to provide a graphical documentation for developers. The transformation rules proposed attempt to provide an appropriate design pattern for the representation of algebraic structures and algorithms within an object-oriented paradigm. Hence, from the UML models produced, it may be possible to map a Focal specification to any appropriate object-oriented programming language, e.g. Java or C#.

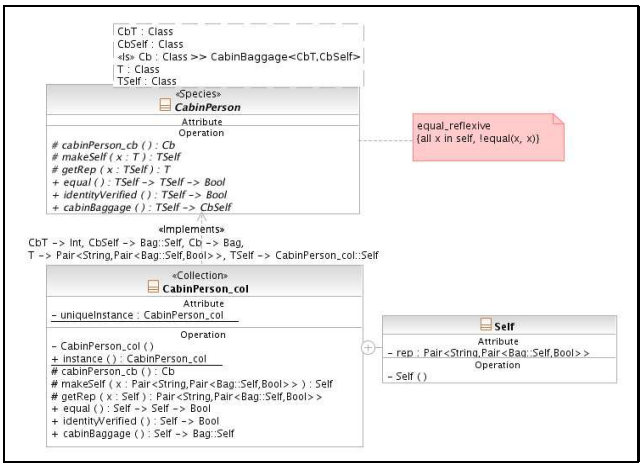

Fig. 4 CabinPerson Classes

Regarding future work, we expect to use the present transformation rules as a basis to generate higher-level views that would be more pertinent for certification authorities or end-users (not only for developers). Another perspective is to apply our transformation process to more concrete specifications (the models realized for the EDEMOI project are quite abstract), such as the standard library of Focal, which consists of a large formalization of Computer Algebra. In this way, it would be possible to see whether the generated UML models are fairly comprehensible and can be used for managing libraries. Finally, we aim to generate more dynamic views of the formal models (sequence and state-transition diagrams) through static analysis performed on Focal specifications.

References

1. R. Bonichon, D. Delahaye, and D. Doligez. Zenon: An Extensible Automated Theorem Prover Producing Checkable Proofs. In *Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, volume 4790 of *LNCS/LNAI*, pages 151–165. Springer, Oct. 2007.
2. O. Caron, B. Carré, A. Muller, and G. Vanwormhoudt. An OCL Formulation of UML2 Template Binding. In *UML Modeling Languages and Applications (UML)*, volume 3273 of *LNCS*, pages 27–40. Springer, Oct. 2004.
3. D. Delahaye, J.-F. Étienne, and V. Viguié Donzeau-Gouge. Certifying Airport Security Regulations using the Focal Environment. In *Formal Methods (FM)*, volume 4085 of *LNCS*, pages 48–63. Springer, Aug. 2006.
4. D. Delahaye, J.-F. Étienne, and V. 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)*, pages 45–52. IEEE CS Press, Nov. 2006.
5. D. Delahaye, J.-F. Étienne, and V. Viguié Donzeau-Gouge. Producing UML Models from Focal Specifications: An Application to Airport Security Regulations. In *Theoretical Aspects of Software Engineering (TASE)*. IEEE CS Press, June 2008.
6. C. Dubois, T. Hardin, and V. Viguié Donzeau-Gouge. Building Certified Components within Focal. In *Trends in Functional Programming (TFP)*, volume 5, pages 33–48. Intellect, Nov. 2004.
7. S. Dupuy, Y. Ledru, and M. Chabre-Peccoud. An Overview of RoZ: A Tool for Integrating UML and Z specifications. In *Conference on Advanced Information Systems Engineering (CAiSE)*, volume 1789 of *LNCS*, pages 417–430. Springer, June 2000.
8. T. Goldschmidt, S. Becker, and A. Uhl. Classification of Concrete Textual Syntax Mapping Approaches. In *European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA)*, volume 5095 of *LNCS*, pages 169–184. Springer, June 2008.
9. A. Idani and Y. Ledru. Dynamic Graphical UML Views from Formal B Specifications. *International Journal of Information and Software Technology*, 48(3):154–169, Mar. 2006.
10. F. Jouault and I. Kurtev. On the Interoperability of Model-to-Model Transformation Languages. *Science of Computer Programming*, 68(3):114–137, Oct. 2007.
11. S.-K. Kim and D. A. Carrington. A Formal Mapping between UML Models and Object-Z Specifications. In *International Z and B Conference (ZB)*, volume 1878 of *LNCS*, pages 2–21. Springer, Sept. 2000.
12. R. Laleau and F. Polack. Coming and Going from UML to B: A Proposal to Support Traceability in Rigorous IS Development. In *International Z and B Conference (ZB)*, volume 2272 of *LNCS*, pages 517–534. Springer, Jan. 2002.
13. M. Maarek and V. Prevosto. FocDoc: The Documentation System of Foc. In *Calculemus*. LIP6, Sept. 2003.
14. The EDEMOI Project, 2003. `http://www-lsr.imag.fr/EDEMOI/`.
15. The Focal Development Team. *Focal, version 0.3.1*. CNAM/INRIA/LIP6, May 2005. Available at: `http://focal.inria.fr/`.
16. The Object Management Group. *Unified Modeling Language: Superstructure, version 2.1.1*, Feb. 2007. Available at: `http://www.omg.org/`.

BIBLIOGRAPHY

[1] Jean-Raymond Abrial. *The B Book, Assigning Programs to Meanings*. Cambridge University Press, Cambridge (UK), 1996. ISBN 0521496195.

[2] Annie I. Antón, Travis D. Breaux, Dimitris Karagiannis, and John Mylopoulos, editors. *International Workshop on Requirements Engineering and Law (RELAW), in conjunction with the International Requirements Engineering Conference (RE)*. IEEE CS Press, September 2008.

[3] María Virginia Aponte and Roberto Di Cosmo. Type Isomorphisms for Module Signatures. In *Programming Languages: Implementations, Logics, and Programs (PLILP)*, volume 1140 of *LNCS*, pages 334–346, Aachen (Germany), September 1996. Springer.

[4] María Virginia Aponte, Roberto Di Cosmo, and Catherine Dubois. Signature subtyping modulo type isomorphisms, 1997. Draft.

[5] Dennis S. Arnon, George E. Collins, and Scott McCallum. Cylindrical Algebraic Decomposition I: The Basic Algorithm. *SIAM Journal on Computing*, 13(4):865–877, November 1984.

[6] Dennis S. Arnon, George E. Collins, and Scott McCallum. Cylindrical Algebraic Decomposition II: An Adjacency Algorithm for the Plane. *SIAM Journal on Computing*, 13(4):878–889, November 1984.

[7] Agnès Arnould, Laurent Fuchs, Marc Aiguier, and Thibaud Brunet. Automatic Generation of Functional Programs from CASL Specifications. In *International Conference on Software Engineering Advances (ICSEA)*, page 34, Papeete (Tahiti, French Polynesia), October 2006. IEEE CS Press.

[8] Egidio Astesiano, Michel Bidoit, Hélène Kirchner, Bernd Krieg-Brückner, Peter D. Mosses, Donald Sannella, and Andrzej Tarlecki. CASL: The Common Algebraic Specification Language. *Theoretical Computer Science (TCS)*, 286(2):153–196, September 2002.

[9] Isabelle Attali and Didier Parigot. Integrating Natural Semantics and Attribute Grammars: The Minotaur System. Technical Report 2339, INRIA, September 1994.

[10] Lennart Augustsson. A Compiler for Lazy ML. In *LISP and Functional Programming (LFP)*, pages 218–227, Austin (TX, USA), August 1984. ACM Press.

[11] Serge Autexier, Jacques Calmet, David Delahaye, Patrick D. F. Ion, Laurence Rideau, Renaud Rioboo, and Alan P. Sexton, editors. *Intelligent Computer Mathematics, 10th International Conference, AISC 2010, 17th Symposium, Calculemus 2010, and 9th International Conference, MKM 2010, Paris, France, July 5-10, 2010.*

Proceedings, volume 6167 of *LNCS*, Paris (France), July 2010. Springer. ISBN 978-3-642-14127-0.

[12] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized Metatheory for the Masses: The PoplMark Challenge. In *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 3603 of *LNCS*, pages 50–65, Oxford (UK), August 2005. Springer.

[13] Philippe Ayrault, Matthieu Carlier, David Delahaye, Catherine Dubois, Damien Doligez, Lionel Habib, Thérèse Hardin, Mathieu Jaume, Charles Morisset, François Pessaux, Renaud Rioboo, and Pierre Weis. Trusted Software within Focal. In *Computer & Electronics Security Applications Rendez-Vous (C&ESAR)*, Rennes (France), December 2008.

[14] Henk Barendregt and Arjeh M. Cohen. Electronic Communication of Mathematics and the Interaction of Computer Algebra Systems and Proof Assistants. *Journal of Symbolic Computation (JSC)*, 32(1/2):3–22, July/August 2001.

[15] Saugata Basu, Richard Pollack, and Marie-Françoise Roy. *Algorithms in Real Algebraic Geometry*, volume 10 of *Algorithms and Computation in Mathematics*. Springer, Secausus (NJ, USA), 2nd edition, August 2006. ISBN 3540330984.

[16] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. The Even More Liberalized δ-Rule in Free Variable Semantic Tableaux. In *Computational Logic and Proof Theory: Kurt Gödel Colloquium (KGC)*, volume 713 of *LNCS*, pages 108–119, Brno (Czech Republic), August 1993. Springer.

[17] Stefan Berghofer. Program Extraction in Simply-Typed Higher Order Logic. In *Types for Proofs and Programs (TYPES)*, volume 2646 of *LNCS*, pages 21–38, Berg en Dal (Netherlands), April 2002. Springer.

[18] Stefan Berghofer, Lukas Bulwahn, and Florian Haftmann. Turning Inductive into Equational Specifications. In *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5674 of *LNCS*, pages 131–146, Munich (Germany), August 2009. Springer.

[19] Stefan Berghofer and Tobias Nipkow. Executing Higher Order Logic. In *Types for Proofs and Programs (TYPES)*, volume 2277 of *LNCS*, pages 24–40. Springer, December 2000.

[20] Karim Berkani, Catherine Dubois, Alain Faivre, and Jérôme Falampin. Validation des règles de base de l'Atelier B. *Technique et Science Informatiques (TSI)*, 23(7):855–878, 2004.

[21] Nicolas Bertaux and David Delahaye. Developing Structured Libraries using the Focal Environment. In *Modules and Libraries for Proof Assistants (MLPA)*, volume 429, pages 2–10, Montréal (Canada), August 2009. ACM Press.

[22] 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.

[23] Frédéric Blanqui, Thérèse Hardin, and Pierre Weis. On the Implementation of Construction Functions for Non-free Concrete Data Types. In *European Symposium on Programming (ESOP)*, volume 4421 of *LNCS*, pages 95–109, Braga (Portugal), March 2007. Springer.

[24] Richard Bonichon, David Delahaye, and Damien Doligez. Zenon: An Extensible Automated Theorem Prover Producing Checkable Proofs. In *Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, volume 4790 of *LNCS/LNAI*, pages 151–165, Yerevan (Armenia), October 2007. Springer.

[25] Patrick Borras, Dominique Clément, Thierry Despeyroux, Janet Incerpi, Gilles Kahn, Bernard Lang, and Valérie Pascual. Centaur: The System. In *Practical Software Development Environments (PSDE)*, volume 24(2) of *SIGPLAN Notices*, pages 14–24, Boston (MA, USA), November 1988. ACM Press.

[26] Sylvain Boulmé. *Spécification d'un environnement dédié à la programmation certifiée de bibliothèques de calcul formel*. PhD thesis, Université Pierre et Marie Curie (Paris 6), December 2000.

[27] Samuel Boutin. Using Reflection to Build Efficient and Certified Decision Procedures. In *Theoretical Aspects of Computer Software (TACS)*, volume 1281 of *LNCS*, pages 515–529, Sendai (Japan), September 1997. Springer.

[28] William S. Brown. The Subresultant PRS Algorithm. *ACM Transactions on Mathematical Software (TOMS)*, 4(3):237–249, September 1978.

[29] Kim Bruce, Roberto Di Cosmo, and Giuseppe Longo. Provable Isomorphisms of Type. *Mathematical Structures in Computer Science*, 2(2):231–247, June 1992.

[30] Olivier Caron, Bernard Carré, Alexis Muller, and Gilles Vanwormhoudt. An OCL Formulation of UML2 Template Binding. In *UML Modeling Languages and Applications (UML)*, volume 3273 of *LNCS*, pages 27–40. Springer, October 2004.

[31] John Cartmell. Generalized Algebraic Theories and Contextual Categories. *Annals of Pure and Applied Logic (APAL)*, 32:209–243, 1986.

[32] Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. A TLA+ Proof System. In *Knowledge Exchange: Automated Provers and Proof Assistants (KEAPPA)*, volume 418, pages 17–37, Doha (Qatar), November 2008. CEUR-WS.

[33] Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. Verifying Safety Properties with the TLA+ Proof System. In *International Joint Conference on Automated Reasoning (IJCAR)*, volume 6173 of *LNCS*, pages 142–148, Edinburgh (UK), July 2010. Springer.

[34] Adam Chlipala. Certified Programming with Dependent Types, 2010. http://adam.chlipala.net/cpdt/.

[35] Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Effective Interactive Proofs for Higher-Order Imperative Programs. In

International Conference on Functional Programming (ICFP), pages 79–90, Edinburgh (Scotland, UK), August 2009. ACM Press.

[36] Jacek Chrzaszcz. Implementing Modules in the Coq System. In *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 2758 of *LNCS*, pages 270–286, Rome (Italy), September 2003. Springer.

[37] George E. Collins. Subresultants and Reduced Polynomial Remainder Sequences. *Journal of the ACM*, 14(1):128–142, January 1967.

[38] George E. Collins. Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition. In *Automata Theory and Formal Languages*, volume 33 of *LNCS*, pages 134–183, Kaiserslautern (Germany), May 1975. Springer.

[39] Pierre Corbineau. A Declarative Language for the Coq Proof Assistant. In *Types for Proofs and Programs (TYPES)*, volume 4941 of *LNCS*, pages 69–84, Cividale des Friuli (Italy), May 2007. Springer.

[40] Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas. Goal-Directed Requirements Acquisition. *Science of Computer Programming*, 20:3–50, 1993.

[41] Daniel De Rauglaudre. *Camlp5, version 5.14*. INRIA, April 2010. http://pauillac.inria.fr/~ddr/camlp5/.

[42] David Delahaye. Search2: un outil de recherche dans une bibliothèque de preuves Coq modulo isomorphismes. Master's thesis, Université Pierre et Marie Curie (Paris 6), September 1997.

[43] David Delahaye. Information Retrieval in a Coq Proof Library using Type Isomorphisms. In *Types for Proofs and Programs (TYPES)*, volume 1956 of *LNCS*, pages 131–147, Lökeberg (Sweden), June 1999. Springer.

[44] David Delahaye. A Tactic Language for the System Coq. In *Logic for Programming and Automated Reasoning (LPAR)*, volume 1955 of *LNCS/LNAI*, pages 85–95, Reunion Island (France), November 2000. Springer.

[45] David Delahaye. *Conception de langages pour décrire les preuves et les automatisations dans les outils d'aide à la preuve: une étude dans le cadre du système Coq*. PhD thesis, Université Pierre et Marie Curie (Paris 6), December 2001.

[46] David Delahaye. A Proof Dedicated Meta-Language. In *Logical Frameworks and Meta-Languages (LFM)*, volume 70(2) of *ENTCS*, Copenhagen (Denmark), July 2002. Elsevier.

[47] David Delahaye. Free-Style Theorem Proving. In *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 2410 of *LNCS*, pages 164–181, Hampton (VA, USA), August 2002. Springer.

[48] David Delahaye, Roberto Di Cosmo, and Benjamin Werner. Recherche dans une bibliothèque de preuves Coq en utilisant le type et modulo isomorphismes. In *PRC/GDR de programmation, Pôle Preuves et Spécifications Algébriques*, Rennes (France), November 1997.

[49] David Delahaye, Catherine Dubois, and Jean-Frédéric Étienne. Extracting Purely Functional Contents from Logical Inductive Types. In *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 4732 of *LNCS*, pages 70–85, Kaiserslautern (Germany), September 2007. Springer.

[50] David Delahaye, Catherine Dubois, and Pierre-Nicolas Tollitte. Génération de code fonctionnel certifié à partir de spécifications inductives dans l'environnement Focalize. In *Journées Francophones des Langages Applicatifs (JFLA)*, Vieux-Port La Ciotat (France), January 2010. INRIA.

[51] David Delahaye and Micaela Mayero. `Field`: une procédure de décision pour les nombres réels en Coq. In *Journées Francophones des Langages Applicatifs (JFLA)*, Pontarlier (France), January 2001. INRIA.

[52] David Delahaye and Micaela Mayero. Dealing with Algebraic Expressions over a Field in Coq using Maple. *Journal of Symbolic Computation (JSC)*, 39(5):569–592, May 2005.

[53] David Delahaye and Micaela Mayero. Quantifier Elimination over Algebraically Closed Fields in a Proof Assistant using a Computer Algebra System. In *Symposium on the Integration of Symbolic Computation and Mechanized Reasoning (Calculemus)*, volume 151(1) of *ENTCS*, pages 57–73, University of Newcastle upon Tyn (UK), July 2005. Elsevier.

[54] 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 *LNCS*, pages 48–63, Hamilton, Ontario (Canada), August 2006. Springer.

[55] David Delahaye, Jean-Frédéric Étienne, and Véronique Viguié Donzeau-Gouge. Modeling Airport Security Regulations in Focal. In *Regulations Modelling and their Validation & Verification (REMO2V)*, pages 806–812, Luxembourg (Grand-Duchy of Luxembourg), June 2006. Presses Universitaires de Namur.

[56] 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)*, pages 45–52, Paphos (Cyprus), November 2006. IEEE CS Press.

[57] David Delahaye, Jean-Frédéric Étienne, and Véronique Viguié Donzeau-Gouge. A Formal and Sound Transformation from Focal to UML: An Application to Airport Security Regulations. *Innovations in Systems and Software Engineering (ISSE) NASA Journal*, 4(3):267–274, September 2008.

[58] David Delahaye, Jean-Frédéric Étienne, and Véronique Viguié Donzeau-Gouge. Formal Modeling of Airport Security Regulations using the Focal Environment. In *Requirements Engineering and Law (RELAW)*, Barcelona (Spain), September 2008. IEEE CS Press.

[59] David Delahaye, Jean-Frédéric Étienne, and Véronique Viguié Donzeau-Gouge. Producing UML Models from Focal Specifications: An Application to Airport Security Regulations. In *Theoretical Aspects of Software Engineering (TASE)*, pages 121–124, Nanjing (China), June 2008. IEEE CS Press.

[60] David Delahaye, Jean-Frédéric Étienne, and Véronique Viguié Donzeau-Gouge. *Modeling and Certifying Airport Security Regulations*. Defense, Security and Strategies. Nova Science Publishers, Inc., apr 2010. ISBN 9781608768936.

[61] Roberto Di Cosmo. *Isomorphisms of Types*. PhD thesis, Universitá di Pisa, January 1993.

[62] Roberto Di Cosmo. *Isomorphisms of Types: from λ-calculus to information retrieval and language design*. Progress in Theoretical Computer Science. Birkhauser, 1995. ISBN-0-8176-3763-X.

[63] Catherine Dubois and Richard Gayraud. Compilation de la sémantique naturelle vers ML. In *Journées Francophones des Langages Applicatifs (JFLA)*, Morzine-Avoriaz (France), February 1999. INRIA.

[64] Sophie Dupuy, Yves Ledru, and Monique Chabre-Peccoud. An Overview of RoZ: A Tool for Integrating UML and Z Specifications. In *Conference on Advanced Information Systems Engineering (CAiSE)*, volume 1789 of *LNCS*, pages 417–430. Springer, June 2000.

[65] Roy Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *Journal of Symbolic Logic (JSL)*, 57(3), September 1992.

[66] Jean-Frédéric Étienne. *Certifying Airport Security Regulations using the Focal Environment*. PhD thesis, Conservatoire National des Arts et Métiers (CNAM), July 2008.

[67] Stéphane Fechter. *Sémantique des traits orientés objet de Focal*. PhD thesis, Université Pierre et Marie Curie (Paris 6), July 2005.

[68] John Fitzgerald, Peter Gorm Larsen, Paul Mukherjee, Nico Plat, and Marcel Verhoef. *Validated Designs for Object-Oriented Systems*. Springer, 1st edition, 2005. ISBN 978-1-85233-881-7.

[69] Murdoch Gabbay and Andrew M. Pitts. A New Approach to Abstract Syntax Involving Binders. In *Logic in Computer Science (LICS)*, pages 214–224, Trento (Italy), July 1999. IEEE CS Press.

[70] Mariusz Giero and Freek Wiedijk. MMode, A Mizar Mode for the Proof Assistant Coq. Technical Report NIII-R0333, University of Nijmegen, Nijmegen (The Netherlands), 2003.

[71] Martin Giese and Wolfgang Ahrendt. Hilbert's ϵ-terms in Automated Theorem Proving. In *Analytic Tableaux and Related Methods (TABLEAUX)*, volume 1617 of *LNAI*, pages 171–185, Saratoga Springs (NY, USA), June 1999. Springer.

[72] Thomas Goldschmidt, Steffen Becker, and Axel Uhl. Classification of Concrete Textual Syntax Mapping Approaches. In *European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA)*, volume 5095 of *LNCS*, pages 169–184. Springer, June 2008.

[73] Andrew D. Gordon and Thomas F. Melham. Five Axioms of Alpha-Conversion. In *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 1125 of *LNCS*, pages 173–190, Turku (Finland), August 1996. Springer.

[74] Michael J. C. Gordon, Robin Milner, Lockwood Morris, Malcolm C. Newey, and Christopher P. Wadsworth. A Metalanguage for Interactive Proof in LCF. In *Principles of Programming Languages (POPL)*, pages 119–130, Tucson (AZ, USA), January 1978. ACM Press.

[75] Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *LNCS*. Springer, 1979.

[76] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, Norwell (MA, USA), 1993. ISBN 0-7923-9311-2.

[77] Thérèse Hardin and Renaud Rioboo. Les objets des mathématiques. *RSTI - L'objet*, 10(4):83–118, October 2004.

[78] John Harrison. Metatheory and Reflection in Theorem Proving: A Survey and Critique. Technical Report CRC-053, SRI Cambridge, Cambridge (UK), February 1995.

[79] John Harrison. A Mizar Mode for HOL. In *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 1125 of *LNCS*, pages 203–220, Turku (Finland), August 1996. Springer.

[80] John Harrison. Proof Style. In *Types for Proofs and Programs (TYPES)*, volume 1512 of *LNCS*, pages 154–172, Aussois (France), September 1996. Springer.

[81] David Hilbert and Paul Bernays. *Grundlagen der Mathematik*, volume 2. Springer, 1939.

[82] Joe Hurd. Integrating Gandalf and HOL. In *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 1690 of *LNCS*, pages 311–322, Nice (France), September 1999. Springer.

[83] Akram Idani and Yves Ledru. Dynamic Graphical UML Views from Formal B Specifications. *International Journal of Information and Software Technology*, 48(3):154–169, March 2006.

[84] Akram Idani and Yves Ledru. Object Oriented Concepts Identification from Formal B Specifications. *Formal Methods in System Design*, 30(3):217–232, June 2007.

[85] Éric Jaeger and Catherine Dubois. Why Would You Trust *B*? In *Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, volume 4790 of *LNCS*, pages 288–302, Yerevan (Armenia), October 2007. Springer.

[86] Mathieu Jaume and Charles Morisset. A Formal Approach to Implement Access Control. *Journal of Information Assurance and Security (JIAS)*, 1(2):137–148, June 2006.

[87] Fairouz Kamareddine and Joe B. Wells. Computerizing Mathematical Text with MathLang. In *Logical and Semantic Frameworks, with Applications (LSFA)*, volume 205 of *ENTCS*, pages 5–30. Elsevier, April 2008.

[88] Eunsuk Kang and Mark Aagaard. Improving the Usability of HOL Through Controlled Automation Tactics. In *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 4732 of *LNCS*, pages 157–172, Kaiserslautern (Germany), September 2007. Springer.

[89] Soon-Kyeong Kim and David A. Carrington. A Formal Mapping between UML Models and Object-Z Specifications. In *International Z and B Conference (ZB)*, volume 1878 of *LNCS*, pages 2–21. Springer, September 2000.

[90] Régine Laleau and Michel Lemoine, editors. *International Workshop on Regulations Modelling and their Validation and Verification (REMO2V), in conjunction with the Conference on Advanced Information Systems Engineering (CAiSE)*. Presses Universitaires de Namur, June 2006.

[91] Régine Laleau and Amel Mammar. An Overview of a Method and its Support Tool for Generating B Specifications from UML Notations. In *Automated Software Engineering (ASE)*, pages 269–272, Grenoble (France), September 2000. IEEE CS Press.

[92] Régine Laleau and Fiona Polack. Coming and Going from UML to B: A Proposal to Support Traceability in Rigorous IS Development. In *International Z and B Conference (ZB)*, volume 2272 of *LNCS*, pages 517–534. Springer, January 2002.

[93] Régine Laleau, Sylvie Vignes, Yves Ledru, Michel Lemoine, Didier Bert, Véronique Donzeau-Gouge, Catherine Dubois, and Fabien Peureux. Application of Requirements Engineering Techniques to the Analysis of Civil Aviation Security Standards. In *Situational Requirements Engineering Processes (SREP)*, pages 91–106, Paris (France), August 2005. University of Limerick (Ireland).

[94] Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):872–923, May 1994.

[95] Leslie Lamport. How to Write a Proof. *American Mathematical Monthly*, 102(7):600–608, August 1995.

[96] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, Boston (MA, USA), 1st edition, 2003. ISBN 978-0-321-14306-8.

[97] Albert C. Leisenring. *Mathematical Logic and Hilbert's ϵ-Symbol*. MacDonald Technical and Scientific, London, 1969. ISBN 0356026795.

[98] Pierre Letouzey. A New Extraction for Coq. In *Types for Proofs and Programs (TYPES)*, volume 2646 of *LNCS*, pages 200–219, Berg en Dal (The Netherlands), April 2002. Springer.

[99] Manuel Maarek and Virgile Prevosto. FocDoc: The Documentation System of Foc. In Thérèse Hardin and Renaud Rioboo, editors, *Symposium on the Integration of Symbolic Computation and Mechanized Reasoning (Calculemus)*, Roma (Italy), September 2003. LIP6.

[100] Lena Magnusson. *The Implementation of ALF - A Proof Editor Based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution*. PhD thesis, Chalmers University of Technology and Göteborg University, January 1995.

[101] Assia Mahboubi. Implementing the Cylindrical Algebraic Decomposition within the Coq System. *Mathematical Structures in Computer Science (MSCS)*, 17(1):99–127, March 2007.

[102] Louis Mandel and Marc Pouzet. *Reactive-ML, version 1.07.06*. LRI, January 2010. http://rml.lri.fr/.

[103] William McCune and Olga Shumsky. System Description: IVY. In *Conference on Automated Deduction (CADE)*, volume 1831 of *LNCS*, pages 401–405, Pittsburgh (PA, USA), June 2000. Springer.

[104] Nicholas A. Merriam and Michael D. Harrison. What is Wrong with GUIs for Theorem Provers? In *User Interfaces for Theorem Provers (UITP)*, Sophia Antipolis (France), September 1997.

[105] David Overton, Zoltan Somogyi, and Peter J. Stuckey. Constraint-Based Mode Analysis of Mercury. In *Principles and Practice of Declarative Programming (PPDP)*, pages 109–120, Pittsburgh (PA, USA), October 2002. ACM Press.

[106] Christine Paulin-Mohring and Benjamin Werner. Synthesis of ML Programs in the System Coq. *Journal of Symbolic Computation (JSC)*, 15(5/6):607–640, May 1993.

[107] Laurence C. Paulson and Kong Woei Susanto. Source-Level Proof Reconstruction for Interactive Theorem Proving. In *Theorem Proving in Higher Order Logics (TPHOLs)*, LNCS. Springer, September 2007.

[108] Mikael Pettersson. A Compiler for Natural Semantics. In *Compiler Construction (CC)*, volume 1060 of *LNCS*, pages 177–191, Linköping (Sweden), April 1996. Springer.

[109] Frank Pfenning and Conal Elliott. Higher-Order Abstract Syntax. In *Programming Language Design and Implementation (PLDI)*, volume 23(7) of *SIGPLAN Notices*, pages 199–208, Atlanta (GA, USA), June 1988. ACM Press.

[110] Marc Pouzet. *Lucid Synchrone, version 3.0*. LRI, April 2006. http://www.lri.fr/~pouzet/lucid-synchrone/.

[111] Virgile Prevosto. *Conception et implantation du langage Foc pour le développement de logiciels certifiés*. PhD thesis, Université Pierre et Marie Curie (Paris 6), September 2003.

[112] Virgile Prevosto, Damien Doligez, and Thérèse Hardin. Algebraic Structures and Dependent Records. In *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 2410 of *LNCS*, pages 298–313, Hampton (VA, USA), August 2002. Springer.

[113] Renaud Rioboo. Invariants for the Focal Language. *Annals of Mathematics and Artificial Intelligence (AMAI)*, 56(3–4):273–296, August 2009.

[114] Mikael Rittri. Using Types as Search Keys in Function Libraries. *Journal of Functional Programming (JFP)*, 1(1):171–89, January 1991.

[115] Sergei Soloviev. The Category of Finite Sets and Cartesian Closed Categories. *Journal of Soviet Mathematics*, 22(3):154–172, June 1983.

[116] Elie Soubiran. A Unified Framework and a Transparent Name-Space for the Coq Module System. In *Modules and Libraries for Proof Assistants (MLPA)*, volume 429, pages 38–45, Montréal (Canada), August 2009. ACM Press.

[117] Antonis Stampoulis and Zhong Shao. VeriML: Typed Computation of Logical Terms inside a Language with Effects. In *International Conference on Functional Programming (ICFP)*, SIGPLAN Notices, Baltimore (MD, USA), September 2010. ACM Press. To appear.

[118] Robert F. Stärk. Input/Output Dependencies of Normal Logic Programs. *Journal of Logic and Computation*, 4(3):249–262, June 1994.

[119] Adam W. Strzeboński. Cylindrical Algebraic Decomposition using Validated Numerics. *Journal of Symbolic Computation (JSC)*, 41(9):1021–1038, September 2006.

[120] Don Syme. Three Tactic Theorem Proving. In *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 1690 of *LNCS*, pages 203–220, Nice (France), September 1999. Springer.

[121] Tanel Tammet. *Gandalf, version c-2.6.r1*. Chalmers University of Technology, June 2003. http://deepthought.ttu.ee/it/gandalf/.

[122] Alfred Tarski. *A Decision Method for Elementary Algebra and Geometry*. University of California Press, Berkeley (CA, USA), 2nd edition, May 1951.

[123] The ACL2 Development Team. *ACL2, version 3.4*. University of Texas, August 2008. http://www.cs.utexas.edu/users/moore/acl2/.

[124] The Alfa Development Team. *Alfa*. Chalmers University of Technology, August 2003. http://www.cs.chalmers.se/~hallgren/Alfa/.

[125] The Axiom Developers. *Axiom: The Scientific Computation System*. Math Action, 2007. http://axiom-developer.org/.

[126] The Calculemus Interest Group, 2010. http://www.calculemus.net/.

[127] The Caml Development Team. *Caml Light, version 0.75*. INRIA, January 2002. http://caml.inria.fr/.

[128] The Caml Development Team. *Objective Caml, version 3.11.2*. INRIA, January 2010. http://caml.inria.fr/.

[129] The Coq Development Team. *Coq, version 8.2*. INRIA, February 2009. http://coq.inria.fr/.

[130] The E Development Team. *E 1.2*. Technische Universität München, July 2010. http://www4.informatik.tu-muenchen.de/~schulz/WORK/eprover.html.

[131] The EDEMOI Project, 2003. http://www-lsr.imag.fr/EDEMOI/.

[132] The Focal Development Team. *Focal, version 0.3.1*. CNAM, INRIA, and LIP6, May 2005. http://focal.inria.fr/.

[133] The Focalize Development Team. *Focalize, version 0.6.0*. CNAM, INRIA, and LIP6, May 2010. http://focalize.inria.fr/.

[134] The FTA Project, 2000. http://www.cs.ru.nl/~freek/fta/.

[135] The HOL Development Team. *HOL 4, Kananaskis-6 release*. Prosper Project and SourceForge.net, September 2010. http://hol.sourceforge.net/.

[136] The Isabelle Development Team. *Isabelle 2008*. University of Cambridge and Technische Universität München, June 2008. http://isabelle.in.tum.de/.

[137] The LEGO Development Team. *LEGO, version 1.3.1*. University of Edinburgh, November 1998. http://www.dcs.ed.ac.uk/home/lego/.

[138] The Maple Development Team. *Maple 12*. Waterloo Maple Inc., May 2008. http://www.maplesoft.com/.

[139] The Mathematica Development Team. *Mathematica 7*. Wolfram Research, Inc., November 2008. http://www.wolfram.com/products/mathematica/.

[140] The Mizar Development Team. *Mizar, version 7.10.01*. University of Bialystok, October 2008. http://www.mizar.org/.

[141] The NuPRL Development Team. *NuPRL, version 5*. Cornell University, March 2003. http://www.cs.cornell.edu/Info/Projects/NuPRL/.

[142] The Object Management Group. *Object Constraint Language, version 2.0*, May 2006. http://www.omg.org/.

[143] The Object Management Group. *UML/XMI Schema, version 2.1*, October 2007. http://schema.omg.org/spec/UML/2.1.

[144] The Object Management Group. *Unified Modeling Language: Superstructure, version 2.1.1*, February 2007. http://www.omg.org/.

[145] The OpenMath Society. *OpenMath Version 2.0*, June 2007. http://www.openmath.org/.

[146] The PVS Development Team. *PVS, version 4.2*. SRI and NASA, July 2008. http://pvs.csl.sri.com/.

[147] The QEPCAD Development Team. *QEPCAD, version B 1.54*, April 2010. http://www.usna.edu/Users/cs/qepcad/.

[148] The Quotient Project, 2007. http://quotient.loria.fr/.

[149] The REVE Project, 2006. http://reve.futurs.inria.fr/.

[150] The SPASS Development Team. *SPASS 3.7*. Max-Planck-Institut Informatik, May 2010. http://www.spass-prover.org/.

[151] The Twelf Development Team. *Twelf, version 1.5R1*. University of Carnegie Mellon and IT University of Copenhagen, March 2005. http://twelf.plparty.org/.

[152] Laurent Théry. Colouring Proofs: A Lightweight Approach to Adding Formal Structure to Proofs. In *User Interfaces for Theorem Provers (UITP)*, volume 103 of *ENTCS*, pages 121–138, Amsterdam (The Netherlands), September 2003. Elsevier.

[153] Alberto Verdejo and Narciso Martí-Oliet. Executable Structural Operational Semantics in Maude. *Journal of Logic and Algebraic Programming*, 67(1-2):226–293, April 2006.

[154] Andrei Voronkov. *Vampire*. University of Manchester, August 2008. http://www.voronkov.com/vampire.cgi.

[155] Markus Wenzel. Isar - A Generic Interpretative Approach to Readable Formal Proof Documents. In *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 1690 of *LNCS*, pages 167–184, Nice (France), September 1999. Springer.

[156] Freek Wiedijk. Mizar Light for HOL Light. In *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 2152 of *LNCS*, pages 378–394, Edinburgh (Scotland, UK), September 2001. Springer.

[157] Freek Wiedijk. Formal Proof Sketches. In *Types for Proofs and Programs (TYPES)*, volume 3085 of *LNCS*, pages 378–393, Torino (Italy), April 2003. Springer.

[158] Vincent Zammit. On the Implementation of an Extensible Declarative Proof Language. In *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 1690 of *LNCS*, pages 185–202, Nice (France), September 1999. Springer.