IN PARTNERSHIP WITH:
IMT Atlantique Bretagne-Pays de la Loire

Université Nantes

Activity Report 2019

Project-Team GALLINETTE

Gallinette: developing a new generation of proof assistants

IN COLLABORATION WITH: Laboratoire des Sciences du numerique de Nantes

RESEARCH CENTER
Rennes - Bretagne-Atlantique

THEME
Proofs and Verification

Table of contents

Project-Team GALLINETTE

Creation of the Team: 2017 May 01, updated into Project-Team: 2018 June 01

Keywords:

Computer Science and Digital Science:

A2.1.1. - Semantics of programming languages
A2.1.2. - Imperative programming
A2.1.3. - Object-oriented programming
A2.1.4. - Functional programming
A2.1.11. - Proof languages
A2.2.3. - Memory management
A2.4.3. - Proofs
A7.2.3. - Interactive Theorem Proving
A7.2.4. - Mechanized Formalization of Mathematics
A8.4. - Computer Algebra

Other Research Topics and Application Domains:

B6.1. - Software industry

1. Team, Visitors, External Collaborators

Research Scientists

Nicolas Tabareau [Team leader, Inria, Researcher, HDR]
Assia Mahboubi [Inria, Researcher]
Guillaume Munch-Maccagnoni [Inria, Researcher]
Pierre-Marie Pédrot [Inria, Researcher]

Faculty Members

Julien Cohen [Univ de Nantes, Associate Professor]
Rémi Douence [IMT Atlantique, Associate Professor, HDR]
Hervé Grall [IMT Atlantique, Associate Professor]
Guilhem Jaber [Univ de Nantes, Associate Professor]

Post-Doctoral Fellows

Eric Finster [Inria, until Jan 2019]
Marie Kerjean [Inria]
Maxime Lucas [Inria]
Kenji Maillard [Inria, from Dec 2019]
Étienne Miquey [Inria, until Nov 2019]
Pierre Vial [Inria]

PhD Students

Antoine Allioux [Inria]
Meven Bertrand [Univ de Nantes, from Sep 2019]
Gaetan Gilbert [IMT Atlantique]
Ambroise Lafont [IMT Atlantique]
Xavier Montillet [Univ de Nantes]
Loic Pujet [Univ de Nantes, from Sep 2019]
Theo Winterhalter [Univ de Nantes]

Igor Zhirkov [Armines, until Sep 2019]

Technical staff

Simon Boulier [Inria, Engineer]

Interns and Apprentices

Esaie Bauer [Ecole Normale Supérieure Lyon, from Jun 2019 until Jul 2019]
Adnan Ben Mansour [Inria, from Jun 2019 until Aug 2019]
Meven Bertrand [Ecole Normale Supérieure Lyon, until Jul 2019]
Guillaume Combette [Ecole Normale Supérieure Lyon, from Oct 2019]
Lucas Escot [Ecole Normale Supérieure Lyon, from Oct 2019]
Paul Geneau de Lamarliere [Ecole Normale Supérieure Lyon, from Jun 2019 until Jul 2019]
Loic Pujet [Ecole Normale Supérieure Paris, from Apr 2019 until Aug 2019]

2. Overall Objectives

2.1. Overall Objectives

The EPI Gallinette aims at developing a new generation of proof assistants, with the belief that practical experiments must go in pair with foundational investigations:

- The goal is to advance proof assistants both as certified programming languages and mechanised logical systems. Advanced programming and mathematical paradigms must be integrated, notably dependent types and effects. The distinctive approach is to implement new programming and logical paradigms on top of Coq by considering the latter as a target language for compilation.
- The aim of foundational investigations is to extend the boundaries of the Curry-Howard correspondence. It is seen both as providing foundations for programming languages and logic, and as a purveyor of techniques essential to the development of proof assistants. Under this perspective, the development of proof assistants is seen as a total experiment using the correspondence in every aspect: programming languages, type theory, proof theory, rewriting and algebra.

3. Research Program

3.1. Scientific Context

Software quality is a requirement that is becoming more and more prevalent, by now far exceeding the traditional scope of embedded systems. The development of tools to construct software that respects a given specification is a major challenge facing computer science. *Proof assistants* such as Coq [49] provide a formal method whose central innovation is to produce *certified programs* by transforming the very activity of programming. Programming and proving are merged into a single development activity, informed by an elegant but rigid mathematical theory inspired by the correspondence between programming, logic and algebra: the *Curry-Howard correspondence*. For the certification of programs, this approach has shown its efficiency in the development of important pieces of certified software such as the C compiler of the CompCert project [78]. The extracted CompCert compiler is reliable and efficient, running only 15% slower than GCC 4 at optimisation level 2 (`gcc -O2`), a level of optimisation that was considered before to be highly unreliable.

Proof assistants can also be used to *formalise mathematical theories*: they not only provide a means of representing mathematical theories in a form amenable to computer processing, but their internal logic provides a language for reasoning about such theories. In the last decade, proof assistants have been used to verify extremely large and complicated proofs of recent mathematical results, sometimes requiring either intensive computations [60], [64] or intricate combinations of a multitude of mathematical theories [59]. But formalised mathematics is more than just proof checking and proof assistants can help with the organisation mathematical knowledge or even with the discovery of new constructions and proofs.

Unfortunately, the rigidity of the theory behind proof assistants impedes their expressiveness both as programming languages and as logical systems. For instance, a program extracted from Coq only uses a purely functional subset of OCaml, leaving behind important means of expression such as side-effects and objects. Limitations also appears in the formalisation of advanced mathematics: proof assistants do not cope well with classical axioms such as excluded middle and choice which are sometimes used crucially. The fact of the matter is that the development of proof assistants cannot be dissociated from a reflection on the nature of programs and proofs coming from the Curry-Howard correspondence. In the EPC Gallinette, we propose to address several drawbacks of proof assistants by pushing the boundaries of this correspondence.

In the 1970's, the Curry-Howard correspondence was seen as a perfect match between functional programs, intuitionistic logic, and Cartesian closed categories. It received several generalisations over the decades, and now it is more widely understood as a fertile correspondence between computation, logic, and algebra. Nowadays, the view of the Curry-Howard correspondence has evolved from a perfect match to a collection of theories meant to explain similar structures at work in logic and computation, underpinned by mathematical abstractions. By relaxing the requirement of a perfect match between programs and proofs, and instead emphasising the common foundations of both, the insights of the Curry-Howard correspondence may be extended to domains for which the requirements of programming and mathematics may in fact be quite different.

Consider the following two major theories of the past decades, which were until recently thought to be irreconcilable:

- **(Martin-Löf) Type theory:** introduced by Martin-Löf in 1971, this formalism [85] is both a programming language and a logical system. The central ingredient is the use of *dependent types* to allow fine-grained invariants to be expressed in program types. In 1985, Coquand and Huet developed a similar system called the *calculus of constructions*, which served as logical foundation of the first implementation of Coq. This kind of systems is still under active development, especially with the recent advent of homotopy type theory (HoTT) [107] which gives a new point of view on types and the notion of equality in type theory.
- **The theory of effects:** starting in the 1980's, Moggi [90] and Girard [57] put forward monads and co-monads as describing various compositional notions of computation. In this theory, programs can have side-effects (state, exceptions, input-output), logics can be non-intuitionistic (linear, classical), and different computational universes can interact (modal logics). Recently, the safe and automatic management of resources has also seen a coming of age (Rust, Modern C++) confirming the importance of linear logic for various programming concepts. It is now understood that the characteristic feature of the theory of effects is sensitivity to *evaluation order*, in contrast with type theory which is built around the assumption that evaluation order is irrelevant.

We now outline a series of scientific challenges aimed at understanding of type theory, effects, and their combination.

More precisely, three key axes of improvement have been identified:

1. Making the notion of equality closer to what is usually assumed when doing proofs on black board, with a balance between irrelevant equality for simple structures and equality up-to equivalences for more complex ones (Section 3.2). Such a notion of equality should allow one to implement traditional model transformations that enhance the logical power of the proof assistant using distinct compilation phases.
2. Advancing the foundations of effects within the Curry-Howard approach. The objective is to pave the way for the integration of effects in proof assistants and to prototype the corresponding implementation. This integration should allow for not only certified programming with effects, but also the expression of more powerful logics (Section 3.3).
3. Making more programming features (notably, object polymorphism) available in proof assistants, in order to scale to practical-sized developments. The objective is to enable programming styles closer to common practices. One of the key challenges here is to leverage gradual typing to dependent programming (Section 3.4).

To validate the new paradigms, we propose in Section 3.5 three particular application fields in which members of the team already have a strong expertise: code refactoring, constraint programming and symbolic computation.

3.2. Enhance the computational and logical power of proof assistants

The democratisation of proof assistants based on type theory has likely been impeded one central problem: the mismatch between the conception of equality in mathematics and its formalisation in type theory. Indeed, some basic principles that are used implicitly in mathematics—such as Church's principle of propositional extensionality, which says that two propositions are equal when they are logically equivalent—are not derivable in type theory. Even more problematically, from a computer science point of view, the basic concept of two functions being equal when they are equal at every "point" of their domain is also not derivable: rather, it must be added as an additional axiom. Of course, these principles are consistent with type theory so that working under the corresponding additional assumptions is safe. But the use of these assumptions in a definition potentially clutters its computational behaviour: since axioms are computational black boxes, computation gets stuck at the points of the code where they have been used.

We propose to investigate how expressive logical transformations such as forcing [70] and sheaf construction might be used to enhance the computational and logical power of proof assistants—with a particular emphasis on their implementation in the Coq proof assistant by the means of effective translations (or compilation phases). One of the main topics of this task, in connection to the ERC project CoqHoTT, is the integration in Coq of new concepts inspired by homotopy type theory [107] such as the univalence principle, and higher inductive types.

3.2.1. A definitional proof-irrelevant version of Coq.

In the Coq proof assistant, the sort $\mathbf{Prop}$ stands for the universe of types which are propositions. That is, when a term P has type $\mathbf{Prop}$, the only relevant fact is whether P is inhabited (that is true) or not (that is false). This property, known as *proof irrelevance,* can be expressed formally as: $\forall x\ y : P, x = y$. Originally, the *raison d'être* of the sort $\mathbf{Prop}$ was to characterise types with no computational meaning with the intention that terms of such types could be erased upon extraction. However, the assumption that every element of $\mathbf{Prop}$ should be proof irrelevant has never been integrated to the system. Indeed, in Coq, proof irrelevance for the sort $\mathbf{Prop}$ is not incorporated into the theory: it is only compatible with it, in the sense that its assumption does not give rise to an inconsistent theory. In fact, the exact status of the sort $\mathbf{Prop}$ in Coq has never been entirely clarified, which explains in part this lack of integration. Homotopy type theory brings fresh thinking on this issue and suggests turning $\mathbf{Prop}$ into the collection of terms that a certain static inference procedure tags as proof irrelevant. The goal of this task is to integrate this insight in the Coq system and to implement a definitional proof-irrelevant version of the sort $\mathbf{Prop}$.

3.2.2. Extend the Coq proof assistant with a computational version of univalence

The univalence principle is becoming widely accepted as a very promising avenue to provide new foundations for mathematics and type theory. However, this principle has not yet been incorporated into a proof assistant. Indeed, the very mathematical structures (known as ∞-groupoids) motivating the theory remain to this day an active area of research. Moreover, a correct and decidable type checking procedure for the whole theory raises both computational complexity and logical coherence issues. Observational type theory [32], as implemented in Epigram, provides a first-stage approximation to homotopy type theory, but only deals with functional extensionality and does not capture univalence. Coquand and his collaborators have obtained significant results on the computational meaning of univalence using cubical sets [39], [45]. Bickford has initiated a promising formalisation work [1] in the NuPRL system. However, a complete formalisation in intensional type theory remains an open problem.

[1] http://www.nuprl.org/wip/Mathematics/cubical!type!theory/index.html

Hence a major objective is to achieve a complete internalisation of univalence in intensional type theory, including an integration to a new version of Coq. We will strive to keep compatibility with previous versions, in particular from a performance point of view. Indeed, the additional complexity of homotopy type theory should not induce an overhead in the type checking procedure used by the software if we want our new framework to become rapidly adopted by the community. Concretely, we will make sure that the compilation time of Coq's Standard Library will be of the same order of magnitude.

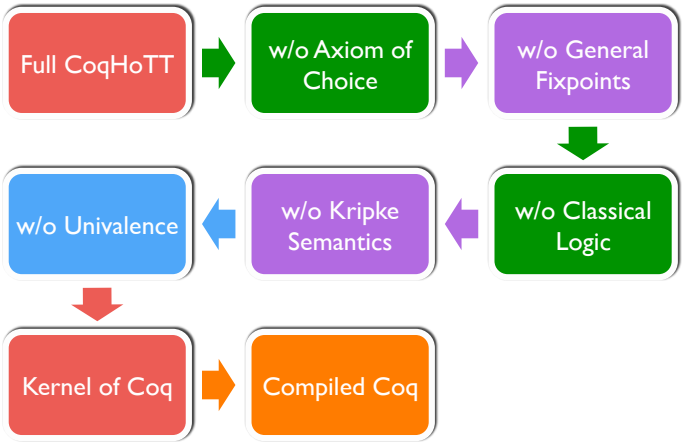

Figure 1. Multiple compilation phases to increase the logical and computational power of Coq.

3.2.3. Extend the logical power of type theory without axioms in a modular way

Extending the power of a logic using model transformations (*e.g.,* forcing transformation [71], [70] or the sheaf construction [100]) is a classic topic of mathematical logic [46], [76]. However, these ideas have not been much investigated in the setting of type theory, even though they may provide a useful framework for extending the logical power of proof assistant in a modular way. There is a good reason for this: with a syntactic notion of equality, the underlying structure of type theory does not conform to the structure of topos used in mathematical logic. A direct incorporation of the standard techniques is therefore not possible. However, a univalent notion of equality brings type theory closer to the required algebraic structure, as it corresponds to the notion of ∞-topos recently studied by Lurie [83]. The goal of this task is to revisit model transformations in the light of the univalence principle, and to obtain in this way new internal transformations in type theory which can in turn be seen as compilation phases. The general notion of an internal syntactical translation has already been investigated in the team [40].

3.2.4. Methodology: Extending type theory with different compilation phases

The Gallinette project advocates the use of distinct compilation phases as a methodology for the design of a new generation of proof assistants featuring modular extensions of a core logic. The essence of a compiler is the separation of the complexity of a translation process into modular stages, and the organization of their re-composition. This idea finds a natural application in the design of complex proof assistants (Figure 1). For instance, the definition of type classes in Coq follows this pattern, and is morally given by the means of a translation into a type-class free kernel. More recently, a similar approach by compilation stages, using the forcing transformation, was used to relax the strict positivity condition guarding inductive types [71], [70]. We believe that this flavour of compilation-based strategies offers a promising direction of investigation for the propose of defining a decidable type checking algorithm for HoTT.

3.3. Semantic and logical foundations for effects in proof assistants based on type theory

We propose the incorporation of effects in the theory of proof assistants at a foundational level. Not only would this allow for certified programming with effects, but it would moreover have implications for both semantics and logic.

We mean *effects* in a broad sense that encompasses both Moggi's monads [90] and Girard's linear logic [57]. These two seminal works have given rise to respective theories of effects (monads) and resources (co-monads). Recent advances, however have unified these two lines of thought: it is now clear that the defining feature of effects, in the broad sense, is sensitivity to evaluation order [79], [50].

In contrast, the type theory that forms the foundations of proof assistants is based on pure λ calculus and is built on the assumption that evaluation order is irrelevant. Evaluation order is therefore the blind spot of type theory. In Moggi [91], integrating the dependent types of type theory with monads is *"the next difficult step [...] currently under investigation"*.

Any realistic program contains effects: state, exceptions, input-output. More generally, evaluation order may simply be important for complexity reasons. With this in mind, many works have focused on certified programming with effects: notably Ynot [95], and more recently $\mathrm{F}^{\star}$ [105] and Idris [41], which propose various ways for encapsulating effects and restricting the dependency of types on effectful terms. Effects are either specialised, such as the monads with Hoare-style pre- and post-conditions found in Ynot or $\mathrm{F}^{\star}$, or more general, such as the algebraic effects implemented in Idris. But whereas there are several experiments and projects pursuing the certification of programs with effects, each making its own choices on how effects and dependency should be merged, there is on the other hand a deficit of logical and semantic investigations.

We propose to develop the foundations of a type theory with effects taking into account the logical and semantic aspects, and to study their practical and theoretical consequences. A type theory that integrates effects would have logical, algebraic and computational implications when viewed through the Curry-Howard correspondence. For instance, effects such as control operators establish a link with classical proof theory [62]. Indeed, control operators provide computational interpretations of type isomorphisms such as $A \cong \neg\neg A$ and $\neg\forall x A \cong \exists x \neg A$ (e.g. [92]), whereas the conventional wisdom of type theory holds that such axioms are non-constructive (this is for instance the point of view that has been advocated so far in homotopy type theory [107]). Another example of an effect with logical content is state (more precisely memoization) which is used to provide constructive content to the classical dependent axiom of choice [38], [74], [66]. In the long term, a whole body of literature on the constructive content of classical proofs is to be explored and integrated, providing rich sources of inspiration: Kohlenbach's proof mining [73] and Simpson's reverse mathematics [103], for instance, are certainly interesting to investigate from the Curry-Howard perspective.

The goal is to develop a type theory with effects that accounts both for practical experiments in certified programming, and for clues from denotational semantics and logical phenomena, in a unified setting.

3.3.1. Models for integrating effects with dependent types

A crucial step is the integration of dependent types with effects, a topic which has remained *"currently under investigation"* [91] ever since the beginning. The difficulty resides in expressing the dependency of types on terms that can perform side-effects during the computation. On the side of denotational semantics, several extensions of categorical models for effects with dependent types have been proposed [29], [108] using axioms that should correspond to restrictions in terms of expressivity but whose practical implications, however, are not immediately transparent. On the side of logical approaches [66], [67], [77], [89], one first considers a drastic restriction to terms that do not compute, which is then relaxed by semantic means. On the side of systems for certified programming such as $\mathrm{F}^{\star}$, the type system ensures that types only depend on pure and terminating terms.

Thus, the recurring idea is to introduce restrictions on the dependency in order to establish an encapsulation of effects. In our approach, we seek a principled description of this idea by developing the concept of *semantic*

value (thunkables, linears) which arose from foundational considerations [56], [102], [93] and whose relevance was highlighted in recent works [80], [99]. The novel aspect of our approach is to seek a proper extension of type theory which would provide foundations for a classical type theory with axiom of choice in the style of Herbelin [66], but which moreover could be generalised to effects other than just control by exploiting an abstract and adaptable notion of semantic value.

3.3.2. Intuitionistic depolarisation

In our view, the common idea that evaluation order does not matter for pure and termination computations should serve as a bridge between our proposals for dependent types in the presence of effects and traditional type theory. Building on the previous goal, we aim to study the relationship between semantic values, purity, and parametricity theorems [101], [58]. Our goal is to characterise parametricity as a form of intuitionistic *depolarisation* following the method by which the first game model of full linear logic was given (Melliès [86], [87]). We have two expected outcomes in mind: enriching type theory with intensional content without losing its properties, and giving an explanation of the dependent types in the style of Idris and $F^{\star}$ where purity- and termination-checking play a role.

3.3.3. Developing the rewriting theory of calculi with effects

An integrated type theory with effects requires an understanding of evaluation order from the point of view of rewriting. For instance, rewriting properties can entail the decidability of some conversions, allowing the automation of equational reasoning in types [27]. They can also provide proofs of computational consistency (that terms are not all equivalent) by showing that extending calculi with new constructs is conservative [104]. In our approach, the λ-calculus is replaced by a calculus modelling the evaluation in an abstract machine [51]. We have shown how this approach generalises the previous semantic and proof-theoretic approaches [33], [79], [81], and overcomes their shortcomings [94].

One goal is to prove computational consistency or decidability of conversions purely using advanced rewriting techniques following a technique introduced in [104]. Another goal is the characterisation of weak reductions: extensions of the operational semantics to terms with free variables that preserve termination, whose iteration is equivalent to strong reduction [28], [54]. We aim to show that such properties derive from generic theorems of higher-order rewriting [110], so that weak reduction can easily be generalised to richer systems with effects.

3.3.4. Direct models and categorical coherence

Proof theory and rewriting are a source of *coherence theorems* in category theory, which show how calculations in a category can be simplified with an embedding into a structure with stronger properties [84], [75]. We aim to explore such results for categorical models of effects [79], [50]. Our key insight is to consider the reflection between *indirect and direct models* [56], [93] as a coherence theorem: it allows us to embed the traditional models of effects into structures for which the rewriting and proof-theoretic techniques from the previous section are effective.

Building on this, we are further interested in connecting operational semantics to 2-category theory, in which a second dimension is traditionally considered for modelling conversions of programs rather than equivalences. This idea has been successfully applied for the λ-calculus [72], [68] but does not scale yet to more realistic models of computation. In our approach, it has already been noticed that the expected symmetries coming from categorical dualities are better represented, motivating a new investigation into this long-standing question.

3.3.5. Models of effects and resources

The unified theory of effects and resources [50] prompts an investigation into the semantics of safe and automatic resource management, in the style of Modern C++ and Rust. Our goal is to show how advanced semantics of effects, resources, and their combination arise by assembling elementary blocks, pursuing the methodology applied by Melliès and Tabareau in the context of continuations [88]. For instance, by combining control flow (exceptions, return) with linearity allows us to describe in a precise way the "Resource Acquisition Is Initialisation" idiom in which the resource safety is ensured with scope-based destructors. A further step would be to reconstruct uniqueness types and borrowing using similar ideas.

3.4. Language extensions for the scaling of proof assistants

The development of tools to construct software systems that respect a given specification is a major challenge of current and future research in computer science. Certified programming with dependent types has recently attracted a lot of interest, and Coq is the *de facto* standard for such endeavours, with an increasing number of users, pedagogical resources, and large-scale projects. Nevertheless, significant work remains to be done to make Coq more usable from a software engineering point of view. The Gallinette team proposes to make progress on three lines of work: (i) the development of gradual certified programming, (ii) the integration of imperative features and object polymorphism in Coq, and (iii) the development of robust tactics for proof engineering for the scaling of formalised libraries.

3.4.1. Gradual Certified Programming

One of the main issues faced by a programmer starting to internalise in a proof assistant code written in a more permissive world is that type theory is constrained by a strict type discipline which lacks flexibility. Concretely, as soon that you start giving more a precise type/specification to a function, the rest of the code interacting with this functions needs to be more precise too. To address this issue, the Gallinette team will put strong efforts into the development of gradual typing in type theory to allow progressive integration of code that comes from a more permissive world.

Indeed, on the way to full verification, programmers can take advantage of a gradual approach in which some properties are simply asserted instead of proven, subject to dynamic verification. Tabareau and Tanter have made preliminary progress in this direction [106]. This work, however, suffers from a number of limitations, the most important being the lack of a mechanism for handling the possibility of runtime errors within Coq. Instead of relying on axioms, this project will explore the application of Section 3.3 to embed effects in Coq. This way, instead of postulating axioms for parts of the development that are too hard/marginal to be dealt with, the system adds dynamic checks. Then, after extraction, we get a program that corresponds to the initial program but with dynamic check for parts that have not been proven, ensuring that the program will raise an error instead of going outside its specification.

This will yield new foundations of gradual certified programming, both more expressive and practical. We will also study how to integrate previous techniques with the extraction mechanism of Coq programs to OCaml, in order to exploit the exception mechanism of OCaml.

3.4.2. Imperative features and object polymorphism in the Coq proof assistant

3.4.2.1. Imperative features.

Abstract data types (ADTs) become useful as the size of programs grows since they provide for a modular approach, allowing abstractions about data to be expressed and then instantiated. Moreover, ADTs are natural concepts in the calculus of inductive constructions. But while it is easy to declare an ADT, it is often difficult to implement an efficient one. Compare this situation with, for example, Okasaki's purely functional data structures [96] which implement ADTs like queues in languages with imperative features. Of course, Okasaki's queues enforce some additional properties for free, such as persistence, but the programmer may prefer to use and to study a simpler implementation without those additional properties. Also in certified symbolic computation (see 3.5.3), an efficient functional implementation of ADTs is often not available, and efficiency is a major challenge in this area. Relying on the theoretical work done in 3.3, we will equip Coq with imperative features and we will demonstrate how they can be used to provide efficient implementations of ADTs. However, it is also often the case that imperative implementation are hard-to-reason-on, requiring for instance the use of separation logic. But in that case, we could take benefice of recent works on integration of separation logic in the Coq proof assistant and in particular the Iris project http://iris-project.org/.

3.4.2.2. Object polymorphism.

Object-oriented programming has evolved since its foundation based on the representation of computations as an exchange of messages between objects. In modern programming languages like Scala, which aims at a synthesis between object-oriented and functional programming, object-orientation concretely results in the use of hierarchies of interfaces ordered by the subtyping relation and the definition of interface implementations

that can interoperate. As observed by Cook and Aldrich [48], [31], interoperability can be considered as the essential feature of objects and is a requirement for many modern frameworks and ecosystems: it means that two different implementations of the same interface can interoperate.

Our objective is to provide a representation of object-oriented programs, by focusing on subtyping and interoperability.

For subtyping, the natural solution in type theory is coercive subtyping [82], as implemented in Coq, with an explicit operator for coercions. This should lead to a shallow embedding, but has limitations: indeed, while it allows subtyping to be faithfully represented, it does not provide a direct means to represent union and intersection types, which are often associated with subtyping (for instance intersection types are present in Scala). A more ambitious solution would be to resort to subsumptive subtyping (or semantic subtyping [55]): in its more general form, a type algebra is extended with boolean operations (union, intersection, complementing) to get a boolean algebra with operators (the original type constructors). Subtying is then interpreted as the natural partial order of the boolean algebra.

We propose to use the type class machinery of Coq to implement semantic subtyping for dependent type theory. Using type class resolution, we can emulate inference rules of subsumptive subtyping without modifying Coq internally. This has also another advantage. As subsumptive subtyping for dependent types should be undecidable in general, using type class resolution allows for an incomplete yet extensible decision procedure.

3.4.3. Robust tactics for proof engineering for the scaling of formalised libraries

When developing certified software, a major part of the effort is spent not only on writing proof scripts, but on *rewriting* them, either for the purpose of code maintenance or because of more significant changes in the base definitions. Regrettably, proof scripts suffer more often than not from a bad programming style, and too many proof developers casually neglect the most elementary principles of well-behaved programmers. As a result, many proof scripts are very brittle, user-defined tactics are often difficult to extend, and sometimes even lack a clear specification. Formal libraries are thus generally very fragile pieces of software. One reason for this unfortunate situation is that proof engineering is very badly served by the tools currently available to the users of the Coq proof assistant, starting with its tactic language. One objective of the Gallinette team is to develop better tools to write proof scripts.

Completing and maintaining a large corpus of formalised mathematics requires a well-designed tactic language. This language should both accommodate the possible specific needs of the theories at stake, and help with diagnostics at refactoring time. Coq's tactic language is in fact two-leveled. First, it includes a basic tactic language, to organise the deductive steps in a proof script and to perform the elementary bureaucracy. Its second layer is a meta-programming language, which allows user to defined their own new tactics at toplevel. Our first direction of work consists in the investigation of the appropriate features of the *basic tactic language*. For instance, the design of the Ssreflect tactic language, and its support for the small scale reflection methodology [61], has been a key ingredient in at least two large scale formalisation endeavours: the Four Colour Theorem [60] and of the Odd Order Theorem [59]. Building on our experience with the Ssreflect tactic language, we will contribute to the ongoing work on the basic tactic language for Coq. The second objective of this task is to contribute to the design of a *typed tactic language*. In particular, we will build on the work of Ziliani and his collaborators [109], extending it with reasoning about the effects that tactics have on the "state of a proof" (e.g. number of sub-goals, metavariables in context). We will also develop a novel approach for incremental type checking of proof scripts, so that programmers gain access to a richer discovery- engineering interaction with the proof assistant.

3.5. Practical experiments

The first three axes of the EPC Gallinette aim at developing a new generation of proof assistants. But we strongly believe that foundational investigations must go hand in hand with practical experiments. Therefore, we expect to benefit from existing expertise and collaborations in the team to experiment our extensions of Coq on real world developments. It should be noticed that those practical experiments are strongly guided by the deep history of research on software engineering of team members.

3.5.1. Certified Code Refactoring

In the context of refactoring of C programs, we intend to formalise program transformations that are written in an imperative style to test the usability of our addition of effects in the proof assistant. This subject has been chosen based on the competence of members of the team.

We are currently working on the formalisation of refactoring tools in Coq [44]. Automatic refactoring of programs in industrial languages is difficult because of the large number of potential interactions between language features that are difficult to predict and to test. Indeed, all available refactoring tools suffer from bugs : they fail to ensure that the generated program has the same behaviour as the input program. To cope with that difficulty, we have chosen to build a refactoring tool with Coq : a program transformation is written in the Coq programming language, then proven correct on all possible inputs, and then an OCaml executable program is generated by the platform. We rely on the CompCert C formalisation of the C language. CompCert is currently the most complete formalisation of an industrial language, which justifies that choice. We have three goals in that project :

- Build a refactoring tool that programmers can rely on and make it available in a popular platform (such as Eclipse, IntelliJ or Frama-C).
- Explore large, drastic program transformations such as replacing a design architecture for an other one, by applying a sequence of small refactoring operations (as we have done for Java and Haskell programs before [47], [43], [30]), while ensuring behaviour preservation.
- Explore the use of enhancements of proof systems on large developments. For instance, refactoring tools are usually developed in the imperative/object paradigm, so the extension of Coq with side effects or with object features proposed in the team can find a direct use-case here.

3.5.2. Certified Constraint Programming

We plan to make use of the internalisation of the object-oriented paradigm in the context of constraint programming. Indeed, this domain is made of very complex algorithms that are often developed using object-oriented programming (as it is the case for instance for CHOCO, which is developed in the Tasc Group at IMT Atlantique, Nantes). We will in particular focus on filtering algorithms in constraint solvers, for which research publications currently propose new algorithms with manual proofs. Their formalisation in Coq is challenging. Another interesting part of constraint solving to formalise is the part that deals with program generation (as opposed to extraction). However, when there are numerous generated pieces of code, it is not realistic to prove their correctness manually, and it can be too difficult to prove the correctness of a generator. So we intend to explore a middle path that consists in generating a piece of code along with its corresponding proof (script or proof term). A target application could be interval constraints (for instance Allen interval algebra or region connection calculus) that can generate thousands of specialised filtering algorithms for a small number of variables [36].

Finally, Rémi Douence has already worked (articles publishing [63], [97], [53], PhD Thesis advising [98]) with different members of the Tasc team. Currently, he supervises with Nicolas Beldiceanu the PhD Thesis of Ekaterina Arafailova in the Tasc team. She studies finite transducers to model time-series constraints [37], [35], [34]. This work requires proofs, manually done for now, we would like to explore when these proofs could be mechanised.

3.5.3. Certified Symbolic Computation

We will investigate how the addition of effects in the Coq proof assistant can facilitate the marriage of computer algebra with formal proofs. Computer algebra systems on one hand, and proof assistants on the other hand, are both designed for doing mathematics with the help of a computer, by the means of symbolic computations. These two families of systems are however very different in nature: computer algebra systems allow for implementations faithful to the theoretical complexity of the algorithms, whereas proof assistants have the expressiveness to specify exactly the semantic of the data-structures and computations.

Experiments have been run that link computer algebra systems with Coq [52], [42]. These bridges rely on the implementation of formal proof-producing core algorithms like normalisation procedures. Incidentally, they require non trivial maintenance work to survive the evolution of both systems. Other proof assistants like the Isabelle/HOL system make use of so-called reflection schemes: the proof assistant can produce code in an external programming language like SML, but also allows to import the values output by these extracted programs back inside the formal proofs. This feature extends the trusted base of code quite significantly but it has been used for major achievements like a certified symbolic/numeric ODE solver [69].

We would like to bring Coq closer to the efficiency and user-friendliness of computer algebra systems: for now it is difficult to use the Coq programming language so that certified implementations of computer algebra algorithms have the right, observable, complexity when they are executed inside Coq. We see the addition of effects to the proof assistant as an opportunity to ease these implementations, for instance by making use of caching mechanisms or of profiling facilities. Such enhancements should enable the verification of computation-intensive mathematical proofs that are currently beyond reach, like the validation of Helfgott's proof of the weak Goldbach conjecture [65].

4. Highlights of the Year

4.1. Highlights of the Year

4.1.1. Permanents members

Gaëtan Gilbert, currently PhD student in the Gallinette team, will be promoted expert engineer for the Coq consortium, staying in the Gallinette team.

Matthieu Sozeau, Inria Junior Researcher and leader of the Coq development team, is joining the Gallinette team end of 2019-beginning of 2020.

Nicolas Tabareau is now director of research (DR2) at Inria since October 2019.

4.1.2. Awards

Marie Kerjean has been awarded a L'Oréal - Unesco Foundation grant.

L'Oréal - Unesco Grants for Women in Science are awarded to talented young female researchers.

5. New Software and Platforms

5.1. Coq

The Coq Proof Assistant

KEYWORDS: Proof - Certification - Formalisation

SCIENTIFIC DESCRIPTION: Coq is an interactive proof assistant based on the Calculus of (Co-)Inductive Constructions, extended with universe polymorphism. This type theory features inductive and co-inductive families, an impredicative sort and a hierarchy of predicative universes, making it a very expressive logic. The calculus allows to formalize both general mathematics and computer programs, ranging from theories of finite structures to abstract algebra and categories to programming language metatheory and compiler verification. Coq is organised as a (relatively small) kernel including efficient conversion tests on which are built a set of higher-level layers: a powerful proof engine and unification algorithm, various tactics/decision procedures, a transactional document model and, at the very top an IDE.

FUNCTIONAL DESCRIPTION: Coq provides both a dependently-typed functional programming language and a logical formalism, which, altogether, support the formalisation of mathematical theories and the specification and certification of properties of programs. Coq also provides a large and extensible set of automatic or semi-automatic proof methods. Coq's programs are extractible to OCaml, Haskell, Scheme, ...

RELEASE FUNCTIONAL DESCRIPTION: Coq version 8.10 contains two major new features: support for a native fixed-precision integer type and a new sort SProp of strict propositions. It is also the result of refinements and stabilization of previous features, deprecations or removals of deprecated features, cleanups of the internals of the system and API, and many documentation improvements. This release includes many user-visible changes, including deprecations that are documented in the next subsection, and new features that are documented in the reference manual.

Version 8.10 is the fifth release of Coq developed on a time-based development cycle. Its development spanned 6 months from the release of Coq 8.9. Vincent Laporte is the release manager and maintainer of this release. This release is the result of 2500 commits and 650 PRs merged, closing 150+ issues.

See the Zenodo citation for more information on this release: https://zenodo.org/record/3476303#.Xe54f5NKjOQ

NEWS OF THE YEAR: Coq 8.10.0 contains:

- some quality-of-life bug fixes, - a critical bug fix related to template polymorphism, - native 63-bit machine integers, - a new sort of definitionally proof-irrelevant propositions: SProp, - private universes for opaque polymorphic constants, - string notations and numeral notations, - a new simplex-based proof engine for the tactics lia, nia, lra and nra, - new introduction patterns for SSReflect, - a tactic to rewrite under binders: under, - easy input of non-ASCII symbols in CoqIDE, which now uses GTK3.

All details can be found in the user manual.

- Participants: Yves Bertot, Frédéric Besson, Maxime Denes, Emilio Jesús Gallego Arias, Gaëtan Gilbert, Jason Gross, Hugo Herbelin, Assia Mahboubi, Érik Martin-Dorel, Guillaume Melquiond, Pierre-Marie Pédrot, Michael Soegtrop, Matthieu Sozeau, Enrico Tassi, Laurent Théry, Théo Zimmermann, Theo Winterhalter, Vincent Laporte, Arthur Charguéraud, Cyril Cohen, Christian Doczkal and Chantal Keller
- Partners: CNRS - Université Paris-Sud - ENS Lyon - Université Paris-Diderot
- Contact: Matthieu Sozeau
- URL: http://coq.inria.fr/

5.2. Math-Components

Mathematical Components library

KEYWORD: Proof assistant

FUNCTIONAL DESCRIPTION: The Mathematical Components library is a set of Coq libraries that cover the prerequiste for the mechanization of the proof of the Odd Order Theorem.

RELEASE FUNCTIONAL DESCRIPTION: The library includes 16 more theory files, covering in particular field and Galois theory, advanced character theory, and a construction of algebraic numbers.

- Participants: Alexey Solovyev, Andrea Asperti, Assia Mahboubi, Cyril Cohen, Enrico Tassi, François Garillot, Georges Gonthier, Ioana Pasca, Jeremy Avigad, Laurence Rideau, Laurent Théry, Russell O'Connor, Sidi Ould Biha, Stéphane Le Roux and Yves Bertot
- Contact: Assia Mahboubi
- URL: http://math-comp.github.io/math-comp/

5.3. Ssreflect

FUNCTIONAL DESCRIPTION: Ssreflect is a tactic language extension to the Coq system, developed by the Mathematical Components team.

- Participants: Assia Mahboubi, Cyril Cohen, Enrico Tassi, Georges Gonthier, Laurence Rideau, Laurent Théry and Yves Bertot
- Contact: Yves Bertot
- URL: http://math-comp.github.io/math-comp/

5.4. Ltac2

KEYWORDS: Coq - Proof assistant

FUNCTIONAL DESCRIPTION: A replacement for Ltac, the tactic language of Coq.

- Contact: Pierre-Marie Pédrot

6. New Results

6.1. Logical Foundations of Programming Languages

Participants: Esaie Bauer, Rémi Douence, Marie Kerjean, Ambroise Lafont, Maxime Lucas, Étienne Miquey, Guillaume Munch-Maccagnoni, Nicolas Tabareau.

6.1.1. Classical Logic

6.1.1.1. Continuation-and-environment-passing style translations: a focus on call-by-need

The call-by-need evaluation strategy for the λ-calculus is an evaluation strategy that lazily evaluates arguments only if needed, and if so, shares computations across all places where it is needed. To implement this evaluation strategy, abstract machines require some form of global environment. While abstract machines usually lead to a better understanding of the flow of control during the execution, easing in particular the definition of continuation-passing style translations, the case of machines with global environments turns out to be much more subtle. The main purpose of [21] is to understand how to type a continuation-and-environment-passing style translations, that it to say how to soundly translate a classical calculus with environment into a calculus that does not have these features. To this end, we focus on a sequent calculus presentation of a call-by-need λ-calculus with classical control for which Ariola et. al already defined an untyped translation and which we equipped with a system of simple types in a previous paper. We present here a type system for the target language of their translation, which highlights a variant of Kripke forcing related to the environment-passing part of the translation. Finally, we show that our construction naturally handles the cases of call-by-name and call-by-value calculi with environment, encompassing in particular the Milner Abstract Machine, a machine with global environments for the call-by-name λ-calculus.

6.1.1.2. Revisiting the duality of computation: an algebraic analysis of classical realizability models

In an impressive series of papers, Krivine showed at the edge of the last decade how classical realizability provides a surprising technique to build models for classical theories. In particular, he proved that classical realizability subsumes Cohen's forcing, and even more, gives rise to unexpected models of set theories. Pursuing the algebraic analysis of these models that was first undertaken by Streicher, Miquel recently proposed to lay the algebraic foundation of classical realizability and forcing within new structures which he called implicative algebras. These structures are a generalization of Boolean algebras based on an internal law representing the implication. Notably, implicative algebras allow for the adequate interpretation of both programs (i.e. proofs) and their types (i.e. formulas) in the same structure. The very definition of implicative algebras takes position on a presentation of logic through universal quantification and the implication and, computationally, relies on the call-by-name λ-calculus. In [13], we investigate the relevance of this choice, by introducing two similar structures. On the one hand, we define disjunctive algebras, which rely on internal laws for the negation and the disjunction and which we show to be particular cases of implicative algebras. On the other hand, we introduce conjunctive algebras, which rather put the focus on conjunctions and on the call-by-value evaluation strategy. We finally show how disjunctive and conjunctive algebras algebraically reflect the well-known duality of computation between call-by-name and call-by-value.

6.1.2. Models of programming languages mixing effects and resources

6.1.2.1. Efficient deconstruction with typed pointer reversal

Building on the connection between resource management in systems programming and ordered logic we established previously, we investigate a pervasive issue in the languages C++ and Rust whereby compiler-generated clean-up functions cause a stack overflow on deep structures. In [17], we show how to generate clean-up algorithms that run in constant time and space for a broad class of ordered algebraic datatypes such as ones that can be found in C++ and Rust or in future extensions of functional programming languages with first-class resources.

6.1.2.2. Resource safety in OCaml

Building on our investigations for a resource-management model for OCaml, we have proposed several preliminary improvements to the OCaml language. We contributed to the design and implementation of new resource management primitives (PRs #2118, #8962), resource-safe C APIs (PRs #8993, #8997, #9037), and core runtime capabilities (PR #8961). (#2118 has been merged into OCaml 4.08 and #8993 and #9037 have been merged into OCaml 4.10.)

We continued to interact with L. White and S. Dolan (Jane Street), on the design of resource management and exception safety in multicore OCaml.

6.1.3. Syntax and Rewriting Systems

6.1.3.1. Reduction Monads and Their Signatures

In [1], we study reduction monads, which are essentially the same as monads relative to the free functor from sets into multigraphs. Reduction monads account for two aspects of the lambda calculus: on the one hand, in the monadic viewpoint, the lambda calculus is an object equipped with a well-behaved substitution; on the other hand, in the graphical viewpoint, it is an oriented multigraph whose vertices are terms and whose edges witness the reductions between two terms. We study presentations of reduction monads. To this end, we propose a notion of reduction signature. As usual, such a signature plays the role of a virtual presentation, and specifies arities for generating operations-possibly subject to equations-together with arities for generating reduction rules. For each such signature, we define a category of models; any model is, in particular, a reduction monad. If the initial object of this category of models exists, we call it the reduction monad presented (or specified) by the given reduction signature. Our main result identifies a class of reduction signatures which specify a reduction monad in the above sense. We show in the examples that our approach covers several standard variants of the lambda calculus.

6.1.3.2. Modules over monads and operational semantics

[22] is a contribution to the search for efficient and high-level mathematical tools to specify and reason about (abstract) programming languages or calculi. Generalising the reduction monads of Ahrens et al., we introduce operational monads, thus covering new applications such as the-calculus, Positive GSOS specifications, and the big-step, simply-typed, call-by-value-calculus. Finally, we design a notion of signature for operational monads that covers all our examples.

6.1.3.3. Modular specification of monads through higher-order presentations

In their work on second-order equational logic, Fiore and Hur have studied presentations of simply typed languages by generating binding constructions and equations among them. To each pair consisting of a binding signature and a set of equations, they associate a category of 'models', and they give a monadicity result which implies that this category has an initial object, which is the language presented by the pair. In [10], we propose, for the untyped setting, a variant of their approach where monads and modules over them are the central notions. More precisely, we study, for monads over sets, presentations by generating ('higher-order') operations and equations among them. We consider a notion of 2-signature which allows to specify a monad with a family of binding operations subject to a family of equations, as is the case for the paradigmatic example of the lambda calculus, specified by its two standard constructions (application and abstraction) subject to β- and η-equalities. Such a 2-signature is hence a pair (Σ,E) of a binding signature Σ and a family

E of equations for Σ. This notion of 2-signature has been introduced earlier by Ahrens in a slightly different context. We associate, to each 2-signature (Σ,E), a category of 'models of (Σ,E); and we say that a 2-signature is 'effective' if this category has an initial object; the monad underlying this (essentially unique) object is the 'monad specified by the 2-signature'. Not every 2-signature is effective; we identify a class of 2-signatures, which we call 'algebraic', that are effective. Importantly, our 2-signatures together with their models enjoy 'modularity': when we glue (algebraic) 2-signatures together, their initial models are glued accordingly. We provide a computer formalization for our main results.

6.1.3.4. The Diamond Lemma for non-terminating rewriting systems

In [16], we study the confluence property for rewriting systems whose underlying set of terms admits a vector space structure. For that, we use deterministic reduction strategies. These strategies are based on the choice of standard reductions applied to basis elements. We provide a sufficient condition of confluence in terms of the kernel of the operator which computes standard normal forms. We present a local criterion which enables us to check the confluence property in this framework. We show how this criterion is related to the Diamond Lemma for terminating rewriting systems

6.1.4. Differential Linear Logic

6.1.4.1. Higher-order distributions for differential linear logic

Linear Logic was introduced as the computational counterpart of the algebraic notion of linearity. Differential Linear Logic refines Linear Logic with a proof-theoretical interpretation of the geometrical process of differentiation. In [24], we construct a polarized model of Differential Linear Logic satisfying computational constraints such as an interpretation for higher-order functions, as well as constraints inherited from physics such as a continuous interpretation for spaces. This extends what was done previously by Kerjean for first order Differential Linear Logic without promotion. Concretely, we follow the previous idea of interpreting the exponential of Differential Linear Logic as a space of higher-order distributions with compact-support, and is constructed as an inductive limit of spaces of distributions on Euclidean spaces. We prove that this exponential is endowed with a co-monadic like structure, with the notable exception that it is functorial only on isomorphisms. Interestingly, as previously argued by Ehrhard, this still allows one to interpret differential linear logic without promotion.

6.1.4.2. Chiralities in topological vector spaces

Chiralities are categories introduced by Mellies to account for a game semantics point of view on negation. In [23], [20], we uncover instances of this structure in the theory of topological vector spaces, thus constructing several new polarized models of Multiplicative Linear Logic. These models improve previously known smooth models of Differential Linear Logic, showing the relevance of chiralities to express topological properties of vector spaces. They are the first denotational polarized models of Multiplicative Linear Logic, based on the pre-existing theory of topological vector spaces, in which two distinct sets of formulas, two distinct negations, and two shifts appear naturally.

6.1.5. Distributed Programming

6.1.5.1. Chemical foundations of distributed aspects.

Distributed applications are challenging to program because they have to deal with a plethora of concerns, including synchronisation, locality, replication, security and fault tolerance. Aspect-oriented programming (AOP) is a paradigm that promotes better modularity by providing means to encapsulate cross-cutting concerns in entities called aspects. Over the last years, a number of distributed aspect-oriented programming languages and systems have been proposed, illustrating the benefits of AOP in a distributed setting. Chemical calculi are particularly well-suited to formally specify the behaviour of concurrent and distributed systems. The join calculus is a functional name-passing calculus, with both distributed and object-oriented extensions. It is used as the basis of concurrency and distribution features in several mainstream languages like C# (Polyphonic C#, now Cω), OCaml (JoCaml), and Scala Joins. Unsurprisingly, practical programming in the join calculus also suffers from modularity issues when dealing with crosscutting concerns. We propose the Aspect Join Calculus [9], an aspect-oriented and distributed variant of the join calculus that addresses crosscutting and

provides a formal foundation for distributed AOP. We develop a minimal aspect join calculus that allows aspects to advise chemical reactions. We show how to deal with causal relations in pointcuts and how to support advanced customisable aspect weaving semantics.

6.2. Type Theory and Proof Assistants

Participants: Simon Boulier, Gaëtan Gilbert, Maxime Lucas, Pierre-Marie Pédrot, Loïc Pujet, Nicolas Tabareau, Théo Winterhalter.

6.2.1. Type Theory

6.2.1.1. Effects in Type Theory.

There is a critical tension between substitution, dependent elimination and effects in type theory. In this paper, we crystallize this tension in the form of a no-go theorem that constitutes the fire triangle of type theory. To release this tension, we propose in [7] DCBPV, an extension of call-by-push-value (CBPV)-a general calculus of effects-to dependent types. Then, by extending to CBPV the well-known decompositions of call-by-name and call-by-value into CBPV, we show why, in presence of effects, dependent elimination must be restricted in call-by-name, and substitution must be restricted in call-by-value. To justify DCBPV and show that it is general enough to interpret many kinds of effects, we define various effectful syntactic translations from DCBPV to Martin-Löf type theory: the reader, weaning and forcing translations.

Traditional approaches to compensate for the lack of exceptions in type theories for proof assistants have severe drawbacks from both a programming and a reasoning perspective. We recently extended the Calculus of Inductive Constructions (CIC) with exceptions. The new exceptional type theory is interpreted by a translation into CIC, covering full dependent elimination, decidable type-checking and canonicity. However, the exceptional theory is inconsistent as a logical system. To recover consistency, we propose an additional translation that uses parametricity to enforce that all exceptions are caught locally. While this enforcement brings logical expressivity gains over CIC, it completely prevents reasoning about exceptional programs such as partial functions. In [6], we addresses the dilemma between exceptions and consistency in a more flexible manner, with the Reasonably Exceptional Type Theory (RETT). RETT is structured in three layers: (a) the exceptional layer, in which all terms can raise exceptions; (b) the mediation layer, in which exceptional terms must be provably parametric; (c) the pure layer, in which terms are non-exceptional, but can refer to exceptional terms. We present the general theory of RETT, where each layer is realized by a predicative hierarchy of universes, and develop an instance of RETT in Coq: the impure layer corresponds to the predicative universe hierarchy, the pure layer is realized by the impredicative universe of propositions, and the mediation layer is reified via a parametricity type class. RETT is the first full dependent type theory to support consistent reasoning about exceptional terms, and the CoqRETT plugin readily brings this ability to Coq programmers.

6.2.1.2. Eliminating Reflection from Type Theory.

Type theories with equality reflection, such as extensional type theory (ETT), are convenient theories in which to formalise mathematics, as they make it possible to consider provably equal terms as convertible. Although type-checking is undecidable in this context, variants of ETT have been implemented, for example in NuPRL and more recently in Andromeda. The actual objects that can be checked are not proof-terms, but derivations of proof-terms. This suggests that any derivation of ETT can be translated into a typecheckable proof term of intensional type theory (ITT). However, this result, investigated categorically by Hofmann in 1995, and 10 years later more syntactically by Oury, has never given rise to an effective translation. In [15], we provide the first syntactical translation from ETT to ITT with uniqueness of identity proofs and functional extensionality. This translation has been defined and proven correct in Coq and yields an executable plugin that translates a derivation in ETT into an actual Coq typing judgment. Additionally, we show how this result is extended in the context of homotopy to a two-level type theory.

6.2.1.3. Setoid type theory - a syntactic translation

[11] introduces setoid type theory, an intensional type theory with a proof-irrelevant universe of propositions and an equality type satisfying function extensionality, propositional extensionality and a definitional computation rule for transport. We justify the rules of setoid type theory by a syntactic translation into a pure type theory with a universe of propositions. We conjecture that our syntax is complete with regards to this translation.

6.2.1.4. The folk model category structure on strict ω-categories is monoidal

In [19], we prove that the folk model category structure on the category of strict ω-categories, introduced by Lafont, Métayer and Worytkiewicz, is monoidal, first, for the Gray tensor product and, second, for the join of ω-categories, introduced by the first author and Maltsiniotis. We moreover show that the Gray tensor product induces, by adjunction, a tensor product of strict (m, n)-categories and that this tensor product is also compatible with the folk model category structure. In particular, we get a monoidal model category structure on the category of strict ω-groupoids. We prove that this monoidal model category structure satisfies the monoid axiom, so that the category of Gray monoids, studied by the second author, bears a natural model category structure.

6.2.2. Proof Assistants

6.2.2.1. Metacoq

The MetaCoq project [26], [26] aims to provide a certified meta-programming environment in Coq. It builds on Template-Coq, a plugin for Coq originally implemented by Malecha (2014), which provided a reifier for Coq terms and global declarations, as represented in the Coq kernel, as well as a denotation command. Recently, it was used in the CertiCoq certified compiler project (Anand et al., 2017), as its front-end language, to derive parametricity properties (Anand and Morrisett, 2018). However, the syntax lacked semantics, be it typing semantics or operational semantics, which should reflect, as formal specifications in Coq, the semantics of Coq's type theory itself. The tool was also rather bare bones, providing only rudimentary quoting and unquoting commands. We generalize it to handle the entire Polymorphic Calculus of Cumulative Inductive Constructions (pCUIC), as implemented by Coq, including the kernel's declaration structures for definitions and inductives, and implement a monad for general manipulation of Coq's logical environment. We demonstrate how this setup allows Coq users to define many kinds of general purpose plugins, whose correctness can be readily proved in the system itself, and that can be run efficiently after extraction. We give a few examples of implemented plugins, including a parametricity translation and a certifying extraction to call-by-value λ-calculus. We also advocate the use of MetaCoq as a foundation for higher-level tools.

6.2.2.2. Verification of Type Checking and Erasure for Coq, in Coq

Coq is built around a well-delimited kernel that perfoms typechecking for definitions in a variant of the Calculus of Inductive Constructions (CIC). Although the metatheory of CIC is very stable and reliable, the correctness of its implementation in Coq is less clear. Indeed, implementing an efficient type checker for CIC is a rather complex task, and many parts of the code rely on implicit invariants which can easily be broken by further evolution of the code. Therefore, on average, one critical bug has been found every year in Coq. [8] presents the first implementation of a type checker for the kernel of Coq (without the module system and template polymorphism), which is proven correct in Coq with respect to its formal specification and axiomatisation of part of its metatheory. Note that because of Gödel's incompleteness theorem, there is no hope to prove completely the correctness of the specification of Coq inside Coq (in particular strong normalisation or canonicity), but it is possible to prove the correctness of the implementation assuming the correctness of the specification, thus moving from a trusted code base (TCB) to a trusted theory base (TTB) paradigm. Our work is based on the MetaCoq project which provides metaprogramming facilities to work with terms and declarations at the level of this kernel. Our type checker is based on the specification of the typing relation of the Polymorphic, Cumulative Calculus of Inductive Constructions (pCUIC) at the basis of Coq and the verification of a relatively efficient and sound type-checker for it. In addition to the kernel implementation, an essential feature of Coq is the so-called extraction: the production of executable code in functional languages from Coq definitions. We present a verified version of this subtle type-and-proof erasure step, therefore enabling the verified extraction of a safe type-checker for Coq.

6.2.2.3. Definitional Proof-Irrelevance without K.

Definitional equality—or conversion—for a type theory with a decidable type checking is the simplest tool to prove that two objects are the same, letting the system decide just using computation. Therefore, the more things are equal by conversion, the simpler it is to use a language based on type theory. Proof-irrelevance, stating that any two proofs of the same proposition are equal, is a possible way to extend conversion to make a type theory more powerful. However, this new power comes at a price if we integrate it naively, either by making type checking undecidable or by realising new axioms—such as uniqueness of identity proofs (UIP)—that are incompatible with other extensions, such as univalence. In [3], taking inspiration from homotopy type theory, we propose a general way to extend a type theory with definitional proof irrelevance, in a way that keeps type checking decidable and is compatible with univalence. We provide a new criterion to decide whether a proposition can be eliminated over a type (correcting and improving the so-called singleton elimination of Coq) by using techniques coming from recent development on dependent pattern matching without UIP. We show the generality of our approach by providing implementations for both Coq and Agda, both of which are planned to be integrated in future versions of those proof assistants.

6.2.2.4. Cubical Synthetic Homotopy Theory

Homotopy type theory is an extension of type theory that enables synthetic reasoning about spaces and homotopy theory. This has led to elegant computer formalizations of multiple classical results from homotopy theory. However, many proofs are still surprisingly complicated to formalize. One reason for this is the axiomatic treatment of univalence and higher inductive types which complicates synthetic reasoning as many intermediate steps, that could hold simply by computation, require explicit arguments. Cubical type theory offers a solution to this in the form of a new type theory with native support for both univalence and higher inductive types. In [14], we show how the recent cubical extension of Agda can be used to formalize some of the major results of homotopy type theory in a direct and elegant manner.

6.3. Program Certifications and Formalisation of Mathematics

Participants: Julien Cohen, Rémi Douence, Guilhem Jaber, Assia Mahboubi, Igor Zhirkov.

6.3.1. CoqTL: A Coq DSL for Rule-Based Model Transformation

In model-driven engineering, model transformation (MT) verification is essential for reliably producing software artifacts. While recent advancements have enabled automatic Hoare-style verification for non-trivial MTs, there are certain verification tasks (e.g. induction) that are intrinsically difficult to automate. Existing tools that aim at simplifying the interactive verification of MTs typically translate the MT specification (e.g. in ATL) and properties to prove (e.g. in OCL) into an interactive theorem prover. However, since the MT specification and proof phases happen in separate languages, the proof developer needs a detailed knowledge of the translation logic. Naturally, any error in the MT translation could cause unsound verification, i.e. the MT executed in the original environment may have different semantics from the verified MT. In [2] , we propose an alternative solution by designing and implementing an internal domain specific language, namely CoqTL, for the specification of declarative MTs directly in the Coq interactive theorem prover. Expressions in CoqTL are written in Gallina (the specification language of Coq), increasing the possibilities of reusing native Coq libraries in the transformation definition and proof. CoqTL specifications can be directly executed by our transformation engine encoded in Coq, or a certified implementation of the transformation can be generated by the native Coq extraction mechanism. We ensure that CoqTL has the same expressive power of Gallina (i.e. if a MT can be computed in Gallina, then it can also be represented in CoqTL). In this article, we introduce CoqTL, evaluate its practical applicability on a use case, and identify its current limitations.

6.3.2. A certificate-based approach to formally verified approximations.

In [12], we present a library to verify rigorous approximations of univariate functions on real numbers, with the Coq proof assistant. Based on interval arithmetic, this library also implements a technique of validation a posteriori based on the Banach fixed-point theorem. We illustrate this technique on the case of operations of division and square root. This library features a collection of abstract structures that organise the specification of rigorous approximations, and modularise the related proofs. Finally, we provide an implementation of verified Chebyshev approximations, and we discuss a few examples of computations.

6.3.3. Formally Verified Approximations of Definite Integrals.

Finding an elementary form for an antiderivative is often a difficult task, so numerical integration has become a common tool when it comes to making sense of a definite integral. Some of the numerical integration methods can even be made rigorous: not only do they compute an approximation of the integral value but they also bound its inaccuracy. Yet numerical integration is still missing from the toolbox when performing formal proofs in analysis. In [5], we present an efficient method for automatically computing and proving bounds on some definite integrals inside the Coq formal system. Our approach is not based on traditional quadrature methods such as Newton-Cotes formulas. Instead, it relies on computing and evaluating antiderivatives of rigorous polynomial approximations, combined with an adaptive domain splitting. Our approach also handles improper integrals, provided that a factor of the integrand belongs to a catalog of identified integrable functions. This work has been integrated to the CoqInterval library.

6.3.4. Reasoning about exact memory transformations induced by refactorings in CompCert C

[18] reports on our work in extending CompCert memory model with a relation to model relocations. It preserves undefined values unlike similar relations defined in CompCert. This relation commutes with memory operations. Our main contributions are the relation itself and mechanically checked proofs of its commutation properties. We intend to use this extension to construct and verify a refactoring tool for programs written in C.

6.3.5. Automating Contextual Equivalence for Higher-Order Programs with References

In [4], we have proposed a framework to study contextual equivalence of programs written in a call-by-value functional language with local integer references. It reduces the problem of contextual equivalence to the problem of non-reachability in a transition system of memory configurations. This reduction is complete for recursion-free programs. Restricting to programs that do not allocate references inside the body of functions, we have encoded this non-reachability problem as a set of constrained Horn clause that can then be checked for satisfiability automatically. Restricting furthermore to a language with finite data-types, we also get a new decidability result for contextual equivalence at any type.

7. Partnerships and Cooperations

7.1. Regional Initiatives

Vercoma (Atlanstic 2020/Attractivity grant)

Goal: Verified computer mathematics.

Coordinator: A. Mahboubi.

Duration: 08/2018 - 08/2021.

7.2. National Initiatives

7.2.1. ANR

FastRelax (ANR-14-CE25-0018).

Goal: Develop computer-aided proofs of numerical values, with certified and reasonably tight error bounds, without sacrificing efficiency.

Coordinator: Bruno Salvy (Inria, ENS Lyon).

Participant: A. Mahboubi.

Duration: 2014-2019.

Website: http://fastrelax.gforge.inria.fr/.

Note: This project started when A. Mahboubi was still in the Specfun project at the Saclay Île-de-France CRI. The budget is still managed there, within the Toccata project, but remains available to A. Mahboubi.

7.3. European Initiatives

7.3.1. FP7 & H2020 Projects

7.3.1.1. CoqHoTT

Title: Coq for Homotopy Type Theory

Programm: H2020

Type: ERC

Duration: June 2015 - May 2020

Coordinator: Inria

Inria contact: Nicolas TABAREAU

Every year, software bugs cost hundreds of millions of euros to companies and administrations. Hence, software quality is a prevalent notion and interactive theorem provers based on type theory have shown their efficiency to prove correctness of important pieces of software like the C compiler of the CompCert project. One main interest of such theorem provers is the ability to extract directly the code from the proof. Unfortunately, their democratization suffers from a major drawback, the mismatch between equality in mathematics and in type theory. Thus, significant Coq developments have only been done by virtuosos playing with advanced concepts of computer science and mathematics. Recently, an extension of type theory with homotopical concepts such as univalence is gaining traction because it allows for the first time to marry together expected principles of equality. But the univalence principle has been treated so far as a new axiom which breaks one fundamental property of mechanized proofs: the ability to compute with programs that make use of this axiom. The main goal of the CoqHoTT project is to provide a new generation of proof assistants with a computational version of univalence and use them as a base to implement effective logical model transformation so that the power of the internal logic of the proof assistant needed to prove the correctness of a program can be decided and changed at compile time—according to a trade-off between efficiency and logical expressivity. Our approach is based on a radically new compilation phase technique into a core type theory to modularize the difficulty of finding a decidable type checking algorithm for homotopy type theory. The impact of the CoqHoTT project will be very strong. Even if Coq is already a success, this project will promote it as a major proof assistant, for both computer scientists and mathematicians. CoqHoTT will become an essential tool for program certification and formalization of mathematics.

Program: COST

Project acronym: EUTYPES

Project title: The European research network on types for programming and verification

Duration: 21/03/2016 - 20/03/2020.

Coordinator: Herman Geuvers (Radboud Univerity, Nijmegen, The Netherlands)

Abstract: Types are pervasive in programming and information technology. A type defines a formal interface between software components, allowing the automatic verification of their connections, and greatly enhancing the robustness and reliability of computations and communications. In rich dependent type theories, the full functional specification of a program can be expressed as a type. Type systems have rapidly evolved over the past years, becoming more sophisticated, capturing new aspects of the behaviour of programs and the dynamics of their execution.

This COST Action will give a strong impetus to research on type theory and its many applications in computer science, by promoting (1) the synergy between theoretical computer scientists, logicians and mathematicians to develop new foundations for type theory, for example as based on the recent development of "homotopy type theory", (2) the joint development of type theoretic tools as proof assistants and integrated programming environments, (3) the study of dependent types for programming and its deployment in software development, (4) the study of dependent types for

verification and its deployment in software analysis and verification. The action will also tie together these different areas and promote cross-fertilisation.

Europe has a strong type theory community, ranging from foundational research to applications in programming languages, verification and theorem proving, which is in urgent need of better networking. A COST Action that crosses the borders will support the collaboration between groups and complementary expertise, and mobilise a critical mass of existing type theory research.

7.4. International Initiatives

7.4.1. Inria International Labs

Inria Chile

Associate Team involved in the International Lab:

7.4.1.1. GECO

Title: Gradual verification and robust proof Engineering for COq

International Partner (Institution - Laboratory - Researcher):

Universidad de Chile (Chile) - Centrum Wiskunde & Informatica - Éric Tanter

Start year: 2018

See also: http://geco.gforge.inria.fr

The development of tools to construct software systems that respect a given specification is a major challenge of current and future research in computer science. Interactive theorem provers based on type theory, such as Coq, have shown their effectiveness to prove correctness of important pieces of software like the C compiler of the CompCert project. Certified programming with dependent types is attracting a lot of attention recently, and Coq is the de facto standard for such endeavors, with an increasing amount of users, pedagogical material, and large-scale projects. Nevertheless, significant work remains to be done to make Coq more usable from a software engineering point of view.

This collaboration project gathers the expertise of researchers from Chile (Inria Chile, Universidad de Chile, Universidad Católica de Valparaíso) and France (Inria Nantes, Inria Paris), in different areas that are crucial to develop the vision of certified software engineering. The focus of this project is both theoretical and practical, covering novel foundations and methods, design of concrete languages and tools, and validation through specific case studies.

The end result will be a number of enhancements to the Coq proof assistant (frameworks, tactic language) together with guidelines and demonstrations of their applicability in realistic scenarios.

7.4.2. Inria International Partners

7.4.2.1. Informal International Partners

- A. Mahboubi holds a part-time endowed professor position in the Department of Mathematics at the Vrije Universiteit Amsterdam (the Netherlands).

7.5. International Research Visitors

7.5.1. Visits of International Scientists

- Matias Toro (U. Chile) visited 1 week in January to work with G. Munch-Maccagnoni.

7.5.2. Visits to International Teams

7.5.2.1. Research Stays Abroad

+ G. Munch-Maccagnoni visited E. Tanter and M. Toro (U. Chile) in March.

8. Dissemination

8.1. Promoting Scientific Activities

8.1.1. Scientific Events: Selection

8.1.1.1. Chair of Conference Program Committees

- A. Mahboubi has served as co-PC chair of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP'19).
- A. Mahboubi has served as co-chair of the CoqPL'20 workshop, satellite of POPL'20.

8.1.1.2. Member of the Conference Program Committees

- P.-M. Pédrot has been a member of the program committee of the Coq Workshop'19 and JFLA'20.
- N. Tabareau has been a member of the program committee of FSCD'19 and POPL'19.
- G. Munch-Maccagnoni has been a member of the external review committee for ICFP'19 conference, and a member of the program committees for the workshops SD'19 (affiliated with FSCD) and LOLA'19 (affilitated with LICS).
- A. Mahboubi has been a member of the program commitee of the ITP'19, Frocos'19 and CPP'20 international conferences, and of the TFP'19 workshop.
- G. Jaber has been a member of the program committee of the Student Research Competition of POPL'20.

8.1.1.3. Reviewer

- P.-M. Pédrot has served as an external reviewer for CPP'19, FoSSaCS'19, ICFP'19 and LICS'19.
- N. Tabareau has served as an external reviewer for LICS'19, CPP'19.
- G. Munch-Maccagnoni has served as an external reviewer for FoSSaCS'20.
- G. Jaber has served as an external reviewer for CONCUR'19, ITP'19, FoSSaCS'19 and POPL'20.

8.1.2. Journal

8.1.2.1. Member of the Editorial Boards

- A. Mahboubi is a member of the editorial board of the Journal of Automated Reasoning.
- A. Mahboubi is co-editing the post-proceedings of the TYPES 2019 conference.

8.1.2.2. Reviewer - Reviewing Activities

- N. Tabareau has been a reviewer for Mathematical Strucutre in Computer Science.

8.1.3. Invited Talks

- P.-M. Pédrot gave a 3 hour long invited class on effectful types theories at the JFLA 2019.
- N. Tabareau has given an invited talk at Coq Workshop 2019 in Portland, Oregon.
- G. Munch-Maccagnoni gave a talk at ITU Copenhaguen in July and a talk at the seminar of philosophy in computer science Codes Sources in Paris in June.
- A. Mahboubi has given an invited talk repectively at the MPC'19, CiE'19, Cade'19 and Types'19 international conferences, at the PxTP'19 international workshop and at the Codes Sources seminar.

8.1.4. Leadership within the Scientific Community

- N. Tabareau is a member of the scientific committee of the GdR of Algebraic Topology.
- A. Mahboubi is a member of the core managment group of the EUTypes project, and leader of the "Tools" working group.
- A. Mahboubi is a member of the steering committee of the ITP conference.
- A. Mahboubi is a member of the scientific committee of the GdR "Informatique Mathématique".

8.2. Teaching - Supervision - Juries

8.2.1. Teaching

Licence : Julien Cohen, Discrete Mathematics, 48h, L1 (IUT), IUT Nantes, France

Licence : Julien Cohen, Introduction to proof assistants (Coq), 8h, L2 (PEIP : IUT/Engineering school), Polytech Nantes, France

Licence : Julien Cohen, Functional Programming (Scala), 22h, L2 (IUT), IUT Nantes, France

Master : Julien Cohen, Object oriented programming (Java), 32h, M1 (Engineering school), Polytech Nantes, France

Master : Julien Cohen, Functional programming (OCaml), 18h, M1 (Engineering school), Polytech Nantes, France

Master : Julien Cohen, Tools for softaware engineering (proof with Frama-C, test, code management), 20h, M1 (Engineering school), Polytech Nantes, France

Licence : Rémi Douence, Object Oriented Design and Programming, 45h, L1 (engineers), IMT-Atlantique, Nantes, France

Licence : Rémi Douence, Introduction to scientific research in computer science (Project: an Haskell interpreter in Java) Project, 20h, L1 (engineers), IMT-Atlantique, Nantes, France

Licence : Rémi Douence, Object Oriented Design and Programming Project, 30h, L1 (apprenticeship), IMT-Atlantique, Nantes, France

Master : Rémi Douence, Functional Programming with Haskell, 20h, M1 (apprenticeship), IMT-Atlantique, Nantes, France

Master : Rémi Douence, Introduction to scientific research in computer science (Project: an Haskell interpreter in Java), 45h, M2 (apprenticeship), IMT-Atlantique, Nantes, France

Licence : Hervé Grall, Algorithms and Discrete Mathematics, 25h , L3 (engineers), IMT-Atlantique, Nantes, France

Licence : Hervé Grall, Object Oriented Design and Programming, 25h , L3 (engineers), IMT-Atlantique, Nantes, France

Licence, Master : Hervé Grall, Modularity and Typing, 40h, L3 and M1, IMT-Atlantique, Nantes, France

Master : Hervé Grall, Service-oriented Computing, 40h, M1 and M2, IMT-Atlantique, Nantes, France

Master : Hervé Grall, Research Project - (Linear) Logic Programming in Coq, 90h (1/3 supervised), M1 and M2, IMT-Atlantique, Nantes, France

Licence : Guilhem Jaber, Computer Tools for Science, 36h, L1, Université de Nantes France

Master : Guilhem Jaber, Verification and Formal Proofs, 18h, M1, Université de Nantes, France

Master : Nicolas Tabareau, Homotopy Type Theory, 24h, M2 LMFI, Université Paris Diderot, France

8.2.2. Supervision

PhD : Gaetan Gilbert, A new foundation for the Coq proof assistant based on the insight of Homotopy Type Theory, IMT Atlantique, advisors: Matthieu Sozeau and Nicolas Tabareau

PhD : Ambroise Lafont,Towards an unbiased approach to specify, implement, and prove properties on programming languages, IMT Atlantique, advisors: Tom Hirschowitz and Nicolas Tabareau

PhD in progress: Xavier Montillet, Rewriting theory for effects and dependent types, Univ Nantes, advisors: Guillaume Munch-Maccagnoni and Nicolas Tabareau

PhD in progress: Théo Winterhalter, Extending the flexibility of the universe hierarchy in type theory, Univ Nantes, advisors: Matthieu Sozeau and Nicolas Tabareau

PhD in progress: Joachim Hotonnier, Deep Specification for Domain-Specific Modelling, advisors: Gerson Sunye (Naomod team), Massimo Tisi (Naomod team), Hervé Grall

PhD in progress: Igor Zhirkov, Certified Refactoring of C in the Coq proof assistant, advisors: Rémi Douence and Julien Cohen.

8.2.2.1. Supervision of interns

E. Bauer has visited the team for an L3 research internship from June to July on the subject "Categorical models of differential linear logic", supervised by Marie Kerjean.

A. Ben Mansour has visited the team for an L3 research internship from June to August on the subject "Parametricity for languages with effects", supervised by G. Munch-Maccagnoni.

M. Bertrand has visited the team from February to July for an internship on the subject "Effects in Type Theory", supervised by N. Tabareau.

L. Pujet has visited the team from April to August for an internship on the subject "Interpreting Cubical Type Theory using forcing", supervised by N. Tabareau.

G. Combette is visiting the team from October 2019 to February 2020 for an internship on the subject "Axiomatic denotational semantics for resource management in systems programming", supervised by G. Munch-Maccagnoni.

L. Escot has visited the team from October to December for an internship on the subject "Univalent Parametricity at Scale", supervised by N. Tabareau.

P. Geneau de Lamarlière has visited the team for an L3 research internship from June to July on the subject "Symbolic computations in algebraic number theory", co-supervised by A. Mahboubi and S. Dahmen (VU Amsterdam).

8.2.3. Juries

- N. Tabareau has served as external member on the PhD jury of Kenji Maillard, defended November 25th at Inria Paris - Université Paris Sciences et Lettres.
- A. Mahboubi has served as external member on the PhD jury of Florian Faissole, defended December 13th at Paris Saclay University.
- A. Mahboubi has served as external member on the PhD jury of Gaëtan Gilbert, defended December 20th at IMT Atlantique.
- A. Mahboubi has served on the 2019 jury for recruiting Inria CRCN at Inria Rennes Bretagne Atlantique.
- A. Mahboubi has served on the 2019 jury for recruiting an "Agrégé Préparateur" at ENS Rennes.

8.3. Popularization

8.3.1. Education

- Hervé Grall has contributed to the project Merite, which aims to promote science learning in middle and high schools. He is the main contributor to the theme "Communication between machines". The project is coordinated by IMT-Atlantique in partnership with 7 other french higher education institutions, the rectorates of the Nantes and Rennes academies, and financed by the "Investments in the Future" and the fund FEDER Pays-de-la-Loire.
- A. Mahboubi has worked with composer Alessandro Bossetti and students of the professional high-school Lycée Michelet, on a project "Art and Mathematics", supported by the Athenor theater.

8.3.2. Interventions

- P.-M. Pédrot was invited to give a talk about his scientific activities in Le Pleynet, during the "Semaine Sport-Études" of the first year students in computer science from the ENS Lyon.

- P.-M. Pédrot gave a similar talk about his scientific activities at the LS2N, during the visit of first year students in computer science from the ENS Cachan.

8.3.3. Internal action

- A. Mahboubi participates to the Irisa/Inria mentoring program.

9. Bibliography

Publications of the year

Articles in International Peer-Reviewed Journals

[1] B. AHRENS, A. HIRSCHOWITZ, A. LAFONT, M. MAGGESI. *Reduction Monads and Their Signatures*, in "Proceedings of the ACM on Programming Languages", January 2020 [*DOI :* 10.1145/3371099], https://hal.inria.fr/hal-02380682

[2] Z. CHENG, M. TISI, R. DOUENCE. *CoqTL: A Coq DSL for Rule-Based Model Transformation*, in "Software and Systems Modeling", 2019, pp. 1-15, forthcoming [*DOI :* 10.1007/s10270-019-00765-6], https://hal.archives-ouvertes.fr/hal-02333564

[3] G. GILBERT, J. COCKX, M. SOZEAU, N. TABAREAU. *Definitional Proof-Irrelevance without K*, in "Proceedings of the ACM on Programming Languages", January 2019, pp. 1-28 [*DOI :* 10.1145/329031610.1145/3290316], https://hal.inria.fr/hal-01859964

[4] G. JABER. *SyTeCi: Automating Contextual Equivalence for Higher-Order Programs with References*, in "Proceedings of the ACM on Programming Languages", 2019, vol. 28, pp. 1-28, forthcoming [*DOI :* 10.1145/3371127], https://hal.archives-ouvertes.fr/hal-02388621

[5] A. MAHBOUBI, G. MELQUIOND, T. SIBUT-PINOTE. *Formally Verified Approximations of Definite Integrals*, in "Journal of Automated Reasoning", February 2019, vol. 62, n° 2, pp. 281-300 [*DOI :* 10.1007/s10817-018-9463-7], https://hal.inria.fr/hal-01630143

[6] P.-M. PÉDROT, N. TABAREAU, H. J. FEHRMANN, É. TANTER. *A Reasonably Exceptional Type Theory*, in "Proceedings of the ACM on Programming Languages", August 2019, vol. 3, pp. 1-29 [*DOI :* 10.1145/3341712], https://hal.inria.fr/hal-02189128

[7] P.-M. PÉDROT, N. TABAREAU. *The Fire Triangle : How to Mix Substitution, Dependent Elimination, and Effects*, in "Proceedings of the ACM on Programming Languages", January 2020 [*DOI :* 10.1145/3371126], https://hal.archives-ouvertes.fr/hal-02383109

[8] M. SOZEAU, S. BOULIER, Y. FORSTER, N. TABAREAU, T. WINTERHALTER. *Coq Coq Correct! Verification of Type Checking and Erasure for Coq, in Coq*, in "Proceedings of the ACM on Programming Languages", January 2020 [*DOI :* 10.1145/3371076], https://hal.archives-ouvertes.fr/hal-02380196

[9] N. TABAREAU, É. TANTER. *Chemical foundations of distributed aspects*, in "Distributed Computing", June 2019, vol. 32, n° Issue 3, pp. 193–216, forthcoming [*DOI :* 10.1007/s00446-018-0334-6], https://hal.inria.fr/hal-01811884

International Conferences with Proceedings

[10] B. AHRENS, A. HIRSCHOWITZ, A. LAFONT, M. MAGGESI. *Modular specification of monads through higher-order presentations*, in "FSCD 2019 - 4th International Conference on Formal Structures for Computation and Deduction", Dortmund, Germany, June 2019, pp. 1-16, https://arxiv.org/abs/1903.00922 - 17 pages [*DOI :* 10.4230/LIPICS.FSCD.2019.6], https://hal.archives-ouvertes.fr/hal-02307998

[11] T. ALTENKIRCH, S. BOULIER, A. KAPOSI, N. TABAREAU. *Setoid type theory - a syntactic translation*, in "MPC 2019 - 13th International Conference on Mathematics of Program Construction", Porto, Portugal, LNCS, Springer, October 2019, vol. 11825, pp. 155-196 [*DOI :* 10.1007/978-3-030-33636-3_7], https://hal.inria.fr/hal-02281225

[12] F. BRÉHARD, A. MAHBOUBI, D. POUS. *A certificate-based approach to formally verified approximations*, in "ITP 2019 - Tenth International Conference on Interactive Theorem Proving", Portland, United States, 2019, pp. 1-19 [*DOI :* 10.4230/LIPICS.ITP.2019.8], https://hal.laas.fr/hal-02088529

[13] É. MIQUEY. *Revisiting the duality of computation: an algebraic analysis of classical realizability models*, in "CSL 2020", Barcelone, Spain, LIPIcs, CSL 2020, January 2020, vol. 152, https://arxiv.org/abs/1910.02732 , https://hal.archives-ouvertes.fr/hal-02305560

[14] A. MÖRTBERG, L. PUJET. *Cubical Synthetic Homotopy Theory*, in "CPP 2020 - 9th ACM SIGPLAN International Conference on Certified Programs and Proofs", New Orleans, United States, ACM, January 2020 [*DOI :* 10.1145/3372885.3373825], https://hal.archives-ouvertes.fr/hal-02394145

[15] T. WINTERHALTER, M. SOZEAU, N. TABAREAU. *Eliminating Reflection from Type Theory : To the Legacy of Martin Hofmann*, in "CPP 2019 - 8th ACM SIGPLAN International Conference on Certified Programs and Proofs", Lisbonne, Portugal, ACM, January 2019, pp. 91-103 [*DOI :* 10.1145/3293880.3294095], https://hal.archives-ouvertes.fr/hal-01849166

Conferences without Proceedings

[16] C. CHENAVIER, M. LUCAS. *The Diamond Lemma for non-terminating rewriting systems using deterministic reduction strategies*, in "IWC 2019 - 8th International Workshop on Confluence", Dortmund, Germany, June 2019, pp. 1-5, https://hal.archives-ouvertes.fr/hal-02385139

[17] G. MUNCH-MACCAGNONI, R. DOUENCE. *Efficient Deconstruction with Typed Pointer Reversal (abstract)*, in "ML 2019 - Workshop", Berlin, Germany, KC Sivaramakrishnan, 2019, pp. 1-8, https://hal.inria.fr/hal-02177326

Research Reports

[18] I. ZHIRKOV, J. COHEN, R. DOUENCE. *Memory bijections: reasoning about exact memory transformations induced by refactorings in CompCert C*, LS2N, Université de Nantes, March 2019, https://hal.archives-ouvertes.fr/hal-02078356

Other Publications

[19] D. ARA, M. LUCAS. *The folk model category structure on strict ω-categories is monoidal*, 2019, https://arxiv.org/abs/1909.13564 - 62 pages, https://hal.archives-ouvertes.fr/hal-02386617

[20] E. BAUER, M. KERJEAN. *Chiralités et exponentielles: un peu de différentiation*, December 2019, working paper or preprint, https://hal.inria.fr/hal-02320704

[21] H. HERBELIN, É. MIQUEY. *Continuation-and-environment-passing style translations: a focus on call-by-need*, January 2019, working paper or preprint, https://hal.inria.fr/hal-01972846

[22] A. HIRSCHOWITZ, T. HIRSCHOWITZ, A. LAFONT. *Modules over monads and operational semantics*, October 2019, working paper or preprint, https://hal.archives-ouvertes.fr/hal-02338144

[23] M. KERJEAN. *Chiralities in topological vector spaces*, December 2019, working paper or preprint, https://hal.inria.fr/hal-02334917

[24] M. KERJEAN, J.-S. LEMAY. *Higher-order distributions for differential linear logic*, January 2019, working paper or preprint, https://hal.inria.fr/hal-01969262

[25] M. LUCAS. *An implementation of polygraphs*, 2019, working paper or preprint, https://hal.archives-ouvertes.fr/hal-02385110

[26] M. SOZEAU, A. ANAND, S. BOULIER, C. COHEN, Y. FORSTER, F. KUNZE, G. MALECHA, N. TABAREAU, T. WINTERHALTER. *The MetaCoq Project*, June 2019, working paper or preprint, https://hal.inria.fr/hal-02167423

References in notes

[27] A. ABEL, T. COQUAND. *Untyped Algorithmic Equality for Martin-Löf's Logical Framework with Surjective Pairs*, in "Typed Lambda Calculi and Applications", P. URZYCZYN (editor), Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2005, vol. 3461, pp. 23-38, http://dx.doi.org/10.1007/11417170_4

[28] B. ACCATTOLI, G. GUERRIERI. *Open Call-by-Value*, in "Programming Languages and Systems", January 2016, http://dx.doi.org/10.1007/978-3-319-47958-3_12

[29] D. AHMAN, N. GHANI, G. D. PLOTKIN. *Dependent Types and Fibred Computational Effects*, in "Proc. FoSSaCS", 2015

[30] A. AJOULI, J. COHEN, J.-C. ROYER. *Transformations between Composite and Visitor Implementations in Java*, in "Software Engineering and Advanced Applications (SEAA), 2013 39th EUROMICRO Conference on", Sept 2013, pp. 25–32, http://dx.doi.org/10.1109/SEAA.2013.53

[31] J. ALDRICH. *The power of interoperability: why objects are inevitable*, in "ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013", A. L. HOSKING, P. T. EUGSTER, R. HIRSCHFELD (editors), ACM, 2013, pp. 101–116

[32] T. ALTENKIRCH, C. MCBRIDE, W. SWIERSTRA. *Observational equality, now!*, in "Proceedings of the ACM Workshop on Programming Languages meets Program Verification (PLPV 2007)", Freiburg, Germany, October 2007, pp. 57–68

[33] J.-M. ANDREOLI. *Logic Programming with Focusing Proof in Linear Logic*, in "Journal of Logic and Computation", 1992, vol. 2, n^{o} 3, pp. 297-347

[34] E. ARAFAILOVA, N. BELDICEANU, R. DOUENCE, M. CARLSSON, P. FLENER, M. A. F. RODRÍGUEZ, J. PEARSON, H. SIMONIS. *Global Constraint Catalog, Volume II, Time-Series Constraints*, in "CoRR", 2016, vol. abs/1609.08925, http://arxiv.org/abs/1609.08925

[35] E. ARAFAILOVA, N. BELDICEANU, R. DOUENCE, P. FLENER, M. A. F. RODRÍGUEZ, J. PEARSON, H. SIMONIS. *Time-Series Constraints: Improvements and Application in CP and MIP Contexts*, in "Integration of AI and OR Techniques in Constraint Programming - 13th International Conference, CPAIOR 2016, Banff, AB, Canada, May 29 - June 1, 2016, Proceedings", C. QUIMPER (editor), Lecture Notes in Computer Science, Springer, 2016, vol. 9676, pp. 18–34, https://doi.org/10.1007/978-3-319-33954-2_2

[36] A. F. BARCO, J. FAGES, É. VAREILLES, M. ALDANONDO, P. GABORIT. *Open Packing for Facade-Layout Synthesis Under a General Purpose Solver*, in "Principles and Practice of Constraint Programming - 21st International Conference, CP 2015, Cork, Ireland, August 31 - September 4, 2015, Proceedings", G. PESANT (editor), Lecture Notes in Computer Science, Springer, 2015, vol. 9255, pp. 508–523, http://dx.doi.org/10.1007/978-3-319-23219-5_36

[37] N. BELDICEANU, M. CARLSSON, R. DOUENCE, H. SIMONIS. *Using finite transducers for describing and synthesising structural time-series constraints*, in "Constraints", 2016, vol. 21, n^{o} 1, pp. 22–40, http://dx.doi.org/10.1007/s10601-015-9200-3

[38] S. BERARDI, M. BEZEM, T. COQUAND. *On the computational content of the axiom of choice*, in "The Journal of Symbolic Logic", 1998, vol. 63, n^{o} 02, pp. 600–622

[39] M. BEZEM, T. COQUAND, S. HUBER. *A model of type theory in cubical sets*, in "Preprint, September", 2013

[40] S. BOULIER, P.-M. PÉDROT, N. TABAREAU. *The next 700 syntactical models of type theory*, in "Certified Programs and Proofs (CPP 2017)", Paris, France, January 2017, pp. 182 - 194 [*DOI :* 10.1145/3018610.3018620], https://hal.inria.fr/hal-01445835

[41] E. BRADY. *Idris, a general-purpose dependently typed programming language: Design and implementation*, in "J. Funct. Program.", 2013, vol. 23, n^{o} 5, pp. 552–593, https://doi.org/10.1017/S095679681300018X

[42] F. CHYZAK, A. MAHBOUBI, T. SIBUT-PINOTE, E. TASSI. *A Computer-Algebra-Based Formal Proof of the Irrationality of ζ(3)*, in "Interactive Theorem Proving", R. G. GERWIN KLEIN (editor), Lecture Notes in Computer Science, Springer, 2014, vol. 8558

[43] J. COHEN, A. AJOULI. *Practical Use of Static Composition of Refactoring Operations*, in "Proceedings of the 28th Annual ACM Symposium on Applied Computing", SAC '13, ACM, 2013, pp. 1700–1705, http://dx.doi.org/10.1145/2480362.2480684

[44] J. COHEN. *Renaming Global Variables in C Mechanically Proved Correct*, in "Proceedings of the Fourth International Workshop on Verification and Program Transformation, Eindhoven, The Netherlands, 2nd April 2016", G. HAMILTON, A. LISITSA, A. P. NEMYTYKH (editors), Electronic Proceedings in Theoretical Computer Science, Open Publishing Association, 2016, vol. 216, pp. 50-64, http://dx.doi.org/10.4204/EPTCS.216.3

[45] C. COHEN, T. COQUAND, S. HUBER, A. MÖRTBERG. *Cubical Type Theory: a constructive interpretation of the univalence axiom*, 2016, To appear in post-proceedings of Types for Proofs and Programs (TYPES 2015)

[46] P. COHEN, M. DAVIS. *Set theory and the continuum hypothesis*, WA Benjamin New York, 1966

[47] J. COHEN, R. DOUENCE, A. AJOULI. *Invertible Program Restructurings for Continuing Modular Maintenance*, in "Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on", March 2012, pp. 347-352, http://dx.doi.org/10.1109/CSMR.2012.42

[48] W. R. COOK. *On understanding data abstraction, revisited*, in "Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA", S. ARORA, G. T. LEAVENS (editors), ACM, 2009, pp. 557–572, http://doi.acm.org/10.1145/1640089.1640133

[49] COQ DEVELOPMENT TEAM, THE. *The Coq proof assistant reference manual*, 2015, Version 8.5, http://coq.inria.fr

[50] P.-L. CURIEN, M. FIORE, G. MUNCH-MACCAGNONI. *A Theory of Effects and Resources: Adjunction Models and Polarised Calculi*, in "Proc. POPL", 2016, http://dx.doi.org/10.1145/2837614.2837652

[51] P.-L. CURIEN, H. HERBELIN. *The duality of computation*, in "ACM SIGPLAN Notices", 2000, vol. 35, pp. 233–243

[52] D. DELAHAYE, M. MAYERO. *Dealing with algebraic expressions over a field in Coq using Maple*, in "J. Symbolic Comput.", 2005, vol. 39, n^{o} 5, pp. 569–592, Special issue on the integration of automated reasoning and computer algebra systems, http://dx.doi.org/10.1016/j.jsc.2004.12.004

[53] R. DOUENCE, X. LORCA, N. LORIANT. *Lazy Composition of Representations in Java*, in "Software Composition, 8th International Conference, SC 2009, Zurich, Switzerland, July 2-3, 2009. Proceedings", A. BERGEL, J. FABRY (editors), Lecture Notes in Computer Science, Springer, 2009, vol. 5634, pp. 55–71, https://doi.org/10.1007/978-3-642-02655-3_6

[54] T. EHRHARD. *Call-by-push-value from a linear logic point of view*, in "European Symposium on Programming Languages and Systems", Springer, 2016, pp. 202–228

[55] A. FRISCH, G. CASTAGNA, V. BENZAKEN. *Semantic Subtyping: Dealing Set-theoretically with Function, Union, Intersection, and Negation Types*, in "J. ACM", September 2008, vol. 55, n^{o} 4, pp. 19:1–19:64

[56] C. FÜHRMANN. *Direct Models for the Computational Lambda Calculus*, in "Electr. Notes Theor. Comput. Sci.", 1999, vol. 20, pp. 245-292

[57] J.-Y. GIRARD. *Linear Logic*, in "Theoretical Computer Science", 1987, vol. 50, pp. 1-102

[58] J.-Y. GIRARD, A. SCEDROV, P. J. SCOTT. *Normal Forms and Cut-Free Proofs as Natural Transformations*, in "in : Logic From Computer Science, Mathematical Science Research Institute Publications 21", Springer-Verlag, 1992, pp. 217–241

[59] G. GONTHIER, A. ASPERTI, J. AVIGAD, Y. BERTOT, C. COHEN, F. GARILLOT, S. ROUX, A. MAHBOUBI, R. O'CONNOR, S. OULD BIHA, I. PASCA, L. RIDEAU, A. SOLOVYEV, E. TASSI, L. THÉRY. *A Machine-Checked Proof of the Odd Order Theorem*, in "Interactive Theorem Proving", S. BLAZY, C. PAULIN-

MOHRING, D. PICHARDIE (editors), Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2013, vol. 7998, pp. 163-179, http://dx.doi.org/10.1007/978-3-642-39634-2_14

[60] G. GONTHIER. *Formal proofs—the four-colour theorem*, in "Notices of the AMS", 2008, vol. 55, n^{o} 11, pp. 1382-1393

[61] G. GONTHIER, A. MAHBOUBI, E. TASSI. *A Small Scale Reflection Extension for the Coq system*, Inria, 2008, n^{o} RR-6455, The Reference Manual of the Ssreflect extension to the Coq tactic language, available at http://hal.inria.fr/inria-00258384

[62] T. G. GRIFFIN. *A Formulae-as-Types Notion of Control*, in "Seventeenth Annual ACM Symposium on Principles of Programming Languages", ACM Press, 1990, pp. 47–58

[63] Y. GUÉHÉNEUC, R. DOUENCE, N. JUSSIEN. *No Java without Caffeine: A Tool for Dynamic Analysis of Java Programs*, in "17th IEEE International Conference on Automated Software Engineering (ASE 2002), 23-27 September 2002, Edinburgh, Scotland, UK", IEEE Computer Society, 2002, 117 p. , https://doi.org/10.1109/ASE.2002.1115000

[64] T. C. HALES, M. ADAMS, G. BAUER, D. T. DANG, J. HARRISON, T. L. HOANG, C. KALISZYK, V. MAGRON, S. MCLAUGHLIN, T. T. NGUYEN, T. Q. NGUYEN, T. NIPKOW, S. OBUA, J. PLESO, J. RUTE, A. SOLOVYEV, A. H. T. TA, T. N. TRAN, D. T. TRIEU, J. URBAN, K. K. VU, R. ZUMKELLER. *A formal proof of the Kepler conjecture*, in "CoRR", 2015, vol. abs/1501.02155, http://arxiv.org/abs/1501.02155

[65] H. A. HELFGOTT. *The ternary Goldbach conjecture is true*, in "ArXiv e-prints", December 2013

[66] H. HERBELIN. *A Constructive Proof of Dependent Choice, Compatible with Classical Logic*, in "LICS 2012 - 27th Annual ACM/IEEE Symposium on Logic in Computer Science", Dubrovnik, Croatia, IEEE Computer Society, June 2012, pp. 365-374, https://hal.inria.fr/hal-00697240

[67] H. HERBELIN, É. MIQUEY. *Toward dependent choice: a classical sequent calculus with dependent types*, in "TYPES 2015", 2015

[68] T. HIRSCHOWITZ. *Cartesian closed 2-categories and permutation equivalence in higher-order rewriting*, in "Logical Methods in Computer Science", 2013, vol. 9, n^{o} 3, 10 p. , 19 pages [*DOI :* 10.2168/LMCS-9(3:10)2013], https://hal.archives-ouvertes.fr/hal-00540205

[69] F. IMMLER. *Verified Reachability Analysis of Continuous Systems*, in "Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings", C. BAIER, C. TINELLI (editors), Lecture Notes in Computer Science, Springer, 2015, vol. 9035, pp. 37–51

[70] G. JABER, G. LEWERTOWSKI, P.-M. PÉDROT, M. SOZEAU, N. TABAREAU. *The Definitional Side of the Forcing*, in "Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016", 2016, pp. 367–376

[71] G. JABER, N. TABAREAU, M. SOZEAU. *Extending type theory with forcing*, in "Logic in Computer Science (LICS), 2012", IEEE, 2012, pp. 395–404

[72] C. B. JAY, N. GHANI. *The Virtues of Eta-Expansion*, in "J. Funct. Program.", 1995, vol. 5, n^{o} 2, pp. 135-154

[73] U. KOHLENBACH. *Applied proof theory: proof interpretations and their use in mathematics*, Springer Science & Business Media, 2008

[74] J.-L. KRIVINE. *Realizability algebras II : new models of ZF + DC*, in "Logical Methods in Computer Science", 2012, vol. 8, n^{o} 1

[75] J. LAMBEK, P. J. SCOTT. *Introduction to higher order categorical logic*, Cambridge University Press, New York, NY, USA, 1986

[76] S. M. LANE, I. MOERDIJK. *Sheaves in Geometry and Logic*, Springer-Verlag, 1992

[77] R. LEPIGRE. *A classical realizability model for a semantical value restriction*, in "European Symposium on Programming Languages and Systems", Springer, 2016, pp. 476–502

[78] X. LEROY. *Formal certification of a compiler back-end or: programming a compiler with a proof assistant*, in "ACM SIGPLAN Notices", 2006, vol. 41, n^{o} 1, pp. 42–54

[79] P. B. LEVY. *Call-By-Push-Value: A Functional/Imperative Synthesis*, Semantic Structures in Computation, Springer, 2004, vol. 2

[80] P. B. LEVY. *Contextual isomorphisms*, in "Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages", ACM, 2017, pp. 400–414

[81] C. LIANG, D. MILLER. *Focusing and polarization in linear, intuitionistic, and classical logics*, in "Theor. Comput. Sci.", 2009, vol. 410, n^{o} 46, pp. 4747-4768

[82] Z. LUO, S. SOLOVIEV, T. XUE. *Coercive subtyping: Theory and implementation*, in "Inf. Comput.", 2013, vol. 223, pp. 18–42

[83] J. LURIE. *Higher topos theory*, Annals of mathematics studies, Princeton University Press, Princeton, N.J., Oxford, 2009

[84] S. MAC LANE. *Natural associativity and commutativity*, in "Selected Papers", 1979, pp. 415–433

[85] P. MARTIN-LÖF. *An intuitionistic theory of types: predicative part*, in "Logic Colloquium '73", 1975, vol. Studies in Logic and the Foundations of Mathematics, n^{o} 80, pp. 73–118

[86] P.-A. MELLIÈS. *Asynchronous Games 3 An Innocent Model of Linear Logic*, in "Electr. Notes Theor. Comput. Sci.", 2005, vol. 122, pp. 171-192

[87] P.-A. MELLIÈS. *Asynchronous Games 4: A Fully Complete Model of Propositional Linear Logic*, in "LICS", 2005, pp. 386-395

[88] P.-A. MELLIÈS, N. TABAREAU. *Resource modalities in tensor logic*, in "Ann. Pure Appl. Logic", 2010, vol. 161, n^{o} 5, pp. 632-653

[89] É. MIQUEY. *A classical sequent calculus with dependent types*, in "European Symposium on Programming", Springer, 2017, pp. 777–803

[90] E. MOGGI. *Computational lambda-calculus and monads*, in "Proceedings of the Fourth Annual IEEE Symposium on Logic in Computer Science (LICS 1989)", IEEE Computer Society Press, June 1989, pp. 14–23

[91] E. MOGGI. *Notions of computation and monads*, in "Inf. Comput.", July 1991, vol. 93, n° 1, pp. 55–92, http://dx.doi.org/10.1016/0890-5401(91)90052-4

[92] G. MUNCH-MACCAGNONI. *Formulae-as-Types for an Involutive Negation*, in "Proceedings of the joint meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (CSL-LICS)", 2014

[93] G. MUNCH-MACCAGNONI. *Models of a Non-Associative Composition*, in "Proc. FoSSaCS", A. MUSCHOLL (editor), LNCS, Springer, 2014, vol. 8412, pp. 397–412

[94] G. MUNCH-MACCAGNONI. *Note on Curry's style for Linear Call-by-Push-Value*, 2017, https://hal.inria.fr/hal-01528857

[95] A. NANEVSKI, G. MORRISETT, A. SHINNAR, P. GOVEREAU, L. BIRKEDAL. *Ynot: Reasoning with the awkward squad*, 2008

[96] C. OKASAKI. *Purely functional data structures*, Cambridge University Press, 1999

[97] C. PRUD'HOMME, X. LORCA, R. DOUENCE, N. JUSSIEN. *Propagation engine prototyping with a domain specific language*, in "Constraints", 2014, vol. 19, n° 1, pp. 57–76, https://doi.org/10.1007/s10601-013-9151-5

[98] C. PRUD'HOMME. *Contrôle de la propagation et de la recherche dans un solveur de contraintes. (Controlling propagation and search within a constraint solver)*, École des mines de Nantes, France, 2014, https://tel.archives-ouvertes.fr/tel-01060921

[99] P.-M. PÉDROT, N. TABAREAU. *An Effectful Way to Eliminate Addiction to Dependence*, January 2017, https://hal.inria.fr/hal-01441829

[100] K. QUIRIN, N. TABAREAU. *Lawvere-Tierney sheafification in Homotopy Type Theory*, in "Journal of Formalized Reasoning", 2016, vol. 9, n° 2 [*DOI :* 10.6092/ISSN.1972-5787/6232], https://hal.inria.fr/hal-01451710

[101] J. C. REYNOLDS. *Types, Abstraction and Parametric Polymorphism*, in "IFIP Congress", 1983, pp. 513-523

[102] P. SELINGER. *Control Categories and Duality: On the Categorical Semantics of the Lambda-Mu Calculus*, in "Math. Struct in Comp. Sci.", 2001, vol. 11, n° 2, pp. 207–260

[103] S. G. SIMPSON. *Subsystems of Second Order Arithmetic*, Second, Cambridge University Press, 2009, Cambridge Books Online, http://dx.doi.org/10.1017/CBO9780511581007

[104] K. STØVRING. *Extending the Extensional Lambda Calculus with Surjective Pairing is Conservative*, in "Logical Methods in Computer Science", 2006, vol. 2, n^{o} 2

[105] N. SWAMY, C. HRIŢCU, C. KELLER, A. RASTOGI, A. DELIGNAT-LAVAUD, S. FOREST, K. BHARGAVAN, C. FOURNET, P.-Y. STRUB, M. KOHLWEISS, J.-K. ZINZINDOHOUE, S. ZANELLA-BÉGUELIN. *Dependent Types and Multi-Monadic Effects in F**, in "43nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)", ACM, January 2016, pp. 256-270, https://www.fstar-lang.org/papers/mumon/

[106] É. TANTER, N. TABAREAU. *Gradual Certified Programming in Coq*, in "Proceedings of the 11th ACM Dynamic Languages Symposium (DLS 2015)", Pittsburgh, PA, USA, ACM Press, October 2015, pp. 26–40

[107] UNIVALENT FOUNDATIONS PROJECT. *Homotopy Type Theory: Univalent Foundations for Mathematics*, http://homotopytypetheory.org/book, 2013

[108] M. VÁKÁR. *A Framework for Dependent Types and Effects*, in "arXiv preprint arXiv:1512.08009", 2015

[109] B. ZILIANI, D. DREYER, N. R. KRISHNASWAMI, A. NANEVSKI, V. VAFEIADIS. *Mtac: A monad for typed tactic programming in Coq*, in "Journal of Functional Programming", 2015, vol. 25, http://dx.doi.org/10.1017/S0956796815000118

[110] F. VAN RAAMSDONK. *Higher-order Rewriting*, in "Proc. Rewrit. Tech. App.", LNCS, Springer, 1999, vol. 1631, pp. 220-239