

RESEARCH CENTRE

Rennes - Bretagne Atlantique

IN PARTNERSHIP WITH:

Université Nantes, Ecole Nationale  
Supérieure Mines-Télécom Atlantique  
Bretagne Pays de la Loire

2021

ACTIVITY REPORT

Project-Team

GALLINETTE

## **Gallinette: developing a new generation of proof assistants**

IN COLLABORATION WITH: Laboratoire des Sciences du numérique de  
Nantes

**DOMAIN**

Algorithmics, Programming, Software  
and Architecture

**THEME**

Proofs and Verification

# Contents

<b>Project-Team GALLINETTE</b>	<b>1</b>
<b>1 Team members, visitors, external collaborators</b>	<b>2</b>
<b>2 Overall objectives</b>	<b>3</b>
<b>3 Research program</b>	<b>3</b>
3.1 Scientific Context	3
3.2 Enhance the computational and logical power of proof assistants	4
3.2.1 A definitional proof-irrelevant version of Coq	5
3.2.2 Extend the Coq proof assistant with a computational version of univalence	5
3.2.3 Extend the logical power of type theory without axioms in a modular way	5
3.2.4 Methodology: Extending type theory with different compilation phases	6
3.3 Semantic and logical foundations for effects in proof assistants based on type theory	6
3.3.1 Models for integrating effects with dependent types	7
3.3.2 Intuitionistic depolarisation	7
3.3.3 Developing the rewriting theory of calculi with effects	7
3.3.4 Direct models and categorical coherence	8
3.3.5 Models of effects and resources	8
3.4 Language extensions for the scaling of proof assistants	8
3.4.1 Gradual Certified Programming	8
3.4.2 Imperative features and object polymorphism in the Coq proof assistant	9
3.4.3 Robust tactics for proof engineering for the scaling of formalised libraries	10
3.5 Practical experiments	10
3.5.1 Certified Code Refactoring	10
3.5.2 Certified Constraint Programming	11
3.5.3 Certified Symbolic Computation	11
<b>4 Application domains</b>	<b>11</b>
<b>5 Highlights of the year</b>	<b>12</b>
<b>6 New software and platforms</b>	<b>12</b>
6.1 New software	12
6.1.1 Ltac2	12
6.1.2 Equations	12
6.1.3 Math-Components	14
6.1.4 Math-comp-analysis	14
6.1.5 MetaCoq	15
6.1.6 Coq	16
6.1.7 memprof-limits	17
6.1.8 ocaml-boxroot	18
<b>7 New results</b>	<b>18</b>
7.1 Type Theory and Proof Assistants	18
7.1.1 Type Theory	18
7.1.2 Proof Assistants	19
7.2 Logical Foundations of Programming Languages	20
7.3 Program Certifications and Formalisation of Mathematics	21
<b>8 Bilateral contracts and grants with industry</b>	<b>22</b>

<b>9 Partnerships and cooperations</b>	<b>25</b>
9.1 International initiatives	25
9.1.1 Associate Teams in the framework of an Inria International Lab or in the framework of an Inria International Program	25
9.1.2 Participation in other International Programs	25
9.2 International research visitors	25
9.2.1 Visits of international scientists	25
9.3 European initiatives	26
9.3.1 FP7 & H2020 projects	26
9.4 National initiatives	27
9.5 Regional initiatives	28
<b>10 Dissemination</b>	<b>28</b>
10.1 Promoting scientific activities	28
10.1.1 Scientific events: organisation	28
10.1.2 Scientific events: selection	29
10.1.3 Journal	29
10.1.4 Invited talks	29
10.1.5 Leadership within the scientific community	29
10.2 Teaching - Supervision - Juries	30
10.2.1 Teaching	30
10.2.2 Supervision	31
10.3 Popularization	31
10.3.1 Interventions	31
<b>11 Scientific production</b>	<b>31</b>
11.1 Major publications	31
11.2 Publications of the year	32
11.3 Cited publications	34

## **Project-Team GALLINETTE**

*Creation of the Project-Team: 2018 June 01*

### **Keywords**

#### **Computer sciences and digital sciences**

- 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 members, visitors, external collaborators

## Research Scientists

- Nicolas Tabareau [Team leader, Inria, Senior Researcher, HDR]
- Assia Mahboubi [Inria, Senior Researcher, HDR]
- Guillaume Munch [Inria, Researcher]
- Pierre-Marie Pedrot [Inria, Researcher]
- Matthieu Sozeau [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

- Yannick Forster [Inria, Dec 2021]
- Kenji Maillard [Inria, until May 2021]

## PhD Students

- Antoine Allieux [Inria, until May 2021]
- Martin Baillon [Inria]
- Meven Bertrand [Univ de Nantes]
- Enzo Crance [Mitsubishi Electric]
- Pierre Giraud [Inria]
- Christopher Hughes [Inria, from Oct 2021]
- Hamza Jaafar [Inria, from Oct 2021]
- Loic Pujet [Univ de Nantes]

## Technical Staff

- Gaetan Gilbert [Inria, Engineer]
- Kenji Maillard [Inria, Engineer, from Jun 2021]

## Interns and Apprentices

- Peio Borthelle [École Normale Supérieure de Lyon, Jan 2021]
- Hamza Jaafar [Univ de Nantes, from Mar 2021 until Aug 2021]
- Thomas Lamiaux [Inria, from Jun 2021 until Jul 2021]
- Nicolas Margulies [Ecole normale supérieure Paris-Saclay, from Feb 2021 until Jul 2021]

## Administrative Assistant

- Anne-Claire Binétruy [Inria]

## 2 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 [54] 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 [84]. 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 [65, 70] or intricate combinations of a multitude of mathematical theories [66]. But formalised mathematics is more than just proof checking and proof assistants can help with 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 appear 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 [91] 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) [113] 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 [96] and Girard [63] 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 by 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 [76] 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 [113] 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 **Prop** stands for the universe of types which are propositions. That is, when a term  $P$  has type **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 **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 **Prop** should be proof irrelevant has never been integrated to the system. Indeed, in Coq, proof irrelevance for the sort **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 **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 **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 **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  $\infty$ -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 [36], 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 [44, 48]. Bickford has initiated a promising formalisation work<sup>1</sup> in the NuPRL system. However, a complete formalisation in intensional type theory remains an open problem.

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.

### 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 [77, 76] or the sheaf construction [106]) is a classic topic of mathematical logic [52, 82]. 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

<sup>1</sup>Cubical Type Theory



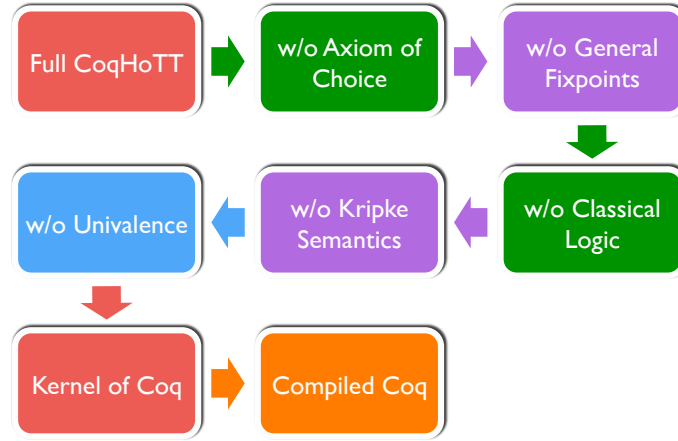


Figure 1: Multiple compilation phases to increase the logical and computational power of Coq.

structure, as it corresponds to the notion of  $\infty$ -topos recently studied by Lurie [89]. 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 [45].

### 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 [77, 76]. We believe that this flavour of compilation-based strategies offers a promising direction of investigation for the purpose 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 [96] and Girard’s linear logic [63]. 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 [85, 55].

In contrast, the type theory that forms the foundations of proof assistants is based on pure  $\lambda$ -calculus and is built on the assumption that evaluation order is irrelevant. Evaluation order is therefore the blind spot of type theory. In Moggi [97], 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 [101], and more recently  $F^*$  [112] and Idris [46], 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  $F^*$ , 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 [68]. Indeed, control operators provide computational interpretations of type isomorphisms such as  $A \cong \neg\neg A$  and  $\neg\forall xA \cong \exists x\neg A$  (e.g. [98]), 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 [113]). 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 [43, 80, 72]. 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 [79] and Simpson's reverse mathematics [110], 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*” [97] 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 [33, 114] 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 [72, 73, 83, 95], 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  $F^*$ , 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* (thinkables, linears) which arose from foundational considerations [62, 109, 99] and whose relevance was highlighted in recent works [86, 103]. 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 [72], 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 terminating 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 [108, 64]. 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 [92, 93]). 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  $F^*$  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 [31]. They can also provide proofs of computational

consistency (that terms are not all equivalent) by showing that extending calculi with new constructs is conservative [111]. In our approach, the  $\lambda$ -calculus is replaced by a calculus modelling the evaluation in an abstract machine [56]. We have shown how this approach generalises the previous semantic and proof-theoretic approaches [37, 85, 87], and overcomes their shortcomings [100].

One goal is to prove computational consistency or decidability of conversions purely using advanced rewriting techniques following a technique introduced in [111]. 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 [32, 60]. We aim to show that such properties derive from generic theorems of higher-order rewriting [107], 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 [90, 81]. We aim to explore such results for categorical models of effects [85, 55]. Our key insight is to consider the reflection between *indirect and direct models* [62, 99] 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  $\lambda$ -calculus [78, 74] 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 [55] 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 [94]. For instance, 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 as you start giving a more precise type/specification to a function, the rest of the code interacting with this function 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 [58]. 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 checks 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

**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 [102] 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 implementations are hard-to-reason-on, requiring for instance the use of separation logic. But in that case, we benefit from recent works on integration of separation logic in the Coq proof assistant and in particular the *Iris project*.

**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 [53, 35], 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 [88], 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 [61]): 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). Subtyping 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 users to define 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 [67], has been a key ingredient in at least two large scale formalisation endeavours: the Four Colour Theorem [65] and of the Odd Order Theorem [66]. 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 [115], 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 [50]. 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 [49, 51, 34]), 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 [41].

Finally, Rémi Douence has already worked (articles publishing [69, 105, 59, 42, 40, 39], PhD Thesis advising [104, 38]) with different members of the Tasc team. Currently, he supervises with Nicolas Beldiceanu the PhD Thesis of Jovial Cheukam Ngouonou in the Tasc team. He studies graph invariants to enhance learning algorithms. This work requires proofs, manually done for now, we would like to explore when these proofs could be mechanized.

### 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 semantics of the data-structures and computations.

Experiments have been run that link computer algebra systems with Coq [57, 47]. 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 [75].

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 [71].

## 4 Application domains

### Programming

- Correct and certified software engineering through the development and the advancement of Coq (e.g. gradualizing type theory, MetaCoq) and practical experiments for its application.



- More general contributions to the programming languages: theoretical works advancing semantic techniques (e.g. deciding equivalence between programs, abstract syntaxes and rewriting, models of effects and resources), and practical works for functional programming (e.g. related to OCaml and Rust).

### Foundations of mathematics

- Formalisation of mathematics
- Contributions to mathematical logic: type theory (e.g. dependent types and univalence), proof theory (e.g. constructive classical logic), categorical logic (e.g. higher algebra, models of focusing and linear logic)

## 5 Highlights of the year

- Assia Mahboubi defended her habilitation thesis [26], Jan 5th 2021.
- ERC COG [Fresco](#) started on Nov 1st 2021.
- Yannick Forster has started a Marie Skłodowska-Curie individual fellowship on Dec 2021.
- The paper [10] has been selected as a distinguished paper of the ICFP'21 conference.
- The paper [7] has been selected as a distinguished paper of the POPL'22 conference.

## 6 New software and platforms

We now list the main pieces of software partially or totally developed in the team.

### 6.1 New software

#### 6.1.1 Ltac2

**Keywords:** Coq, Proof assistant

**Functional Description:** A replacement for Ltac, the tactic language of Coq.

**Contact:** Pierre-Marie Pedrot

#### 6.1.2 Equations

**Keywords:** Coq, Dependent Pattern-Matching, Proof assistant, Functional programming

**Scientific Description:** Equations is a tool designed to help with the definition of programs in the setting of dependent type theory, as implemented in the Coq proof assistant. Equations provides a syntax for defining programs by dependent pattern-matching and well-founded recursion and compiles them down to the core type theory of Coq, using the primitive eliminators for inductive types, accessibility and equality. In addition to the definitions of programs, it also automatically derives useful reasoning principles in the form of propositional equations describing the functions, and an elimination principle for calls to this function. It realizes this using a purely definitional translation of high-level definitions to core terms, without changing the core calculus in any way, or using axioms.

The main features of Equations include:

Dependent pattern-matching in the style of Agda/Epigram, with inaccessible patterns, with and where clauses. The use of the K axiom or a proof of K is configurable, and it is able to solve unification problems without resorting to the K rule if not necessary.

Support for well-founded and mutual recursion using measure/well-foundedness annotations, even on indexed inductive types, using an automatic derivation of the subterm relation for inductive families.

Support for mutual and nested structural recursion using `with` and `where` auxiliary definitions, allowing to factor multiple uses of the same nested fixpoint definition. It proves the expected elimination principles for mutual and nested definitions.

Automatic generation of the defining equations as rewrite rules for every definition.

Automatic generation of the unfolding lemma for well-founded definitions (requiring only functional extensionality).

Automatic derivation of the graph of the function and its elimination principle. In case the automation fails to prove these principles, the user is asked to provide a proof.

A new dependent elimination tactic based on the same splitting tree compilation scheme that can advantageously replace dependent destruction and sometimes inversion as well. The `as` clause of dependent elimination allows to specify exactly the patterns and naming of new variables needed for an elimination.

A set of `Derive` commands for automatic derivation of constructions from an inductive type: its signature, no-confusion property, well-founded subterm relation and decidable equality proof, if applicable.

**Functional Description:** Equations is a function definition plugin for Coq (supporting Coq 8.13 to 8.15, with special support for the Coq-HoTT library), that allows the definition of functions by dependent pattern-matching and well-founded, mutual or nested structural recursion and compiles them into core terms. It automatically derives the clauses equations, the graph of the function and its associated elimination principle.

Equations is based on a simplification engine for the dependent equalities appearing in dependent eliminations that is also usable as a separate tactic, providing an axiom-free variant of dependent destruction.

**Release Contributions:** This is a new major release of Equations, working with Coq 8.13 and 8.14. This version adds an improved syntax (less `,`-separation), integration with the Coq-HoTT library and numerous bug fixes. See the reference manual for details.

This version introduces minor breaking changes along with the following features:

Enhancements of pattern interpretation

No explicit shadowing of pattern variables is allowed anymore. This fixes numerous bugs where generated implicit names introduced by the elaboration of patterns could shadow user-given names, leading to incorrect names in right-hand sides and confusing environments.

Improved syntax for "concise" clauses separated by `|`, at top-level or inside with subprograms. We no longer require to separate them by `,`. For example, the following definition is now accepted:

Equations `foo : nat -> nat := | 0 => 1 | S n => S (foo n)`. The old syntax is however still supported for backwards compatibility.

Multiple patterns can be separated by `,` in addition to `|`, as in:

Equations `trans {A} {x y z : A} (e : x = y) (e' : y = z) : x = z := | 1, 1 => 1`. `Require Import Equations.Equations` does not work anymore. One has to use `Require Import Equations.Prop.Equations` to load the plugin's default instance where equality is in `Prop`. From Equations `Require Import Equations` is unaffected.

Use `Require Import Equations.HoTT.All` to use the HoTT variant of the library compatible with the Coq HoTT library. The plugin then reuses the definition of paths from the HoTT library and all its constructions are universe polymorphic. As for the HoTT library alone, `coq` must be passed the arguments `-noinit -indices-matter` to use the library and plugin. The `coq-equations` opam package depends optionally on `coq-hott`, so if `coq-hott` is installed before it, `coq-equations` will



automatically install the HoTT library variant in addition to the standard one. This variant of Equations allows to write very concise dependent pattern-matchings on equality:

```
Require Import Equations.HoTT.All. Equations sym {A} {x y : A} (e : x = y) : y = x := | 1 => 1.
New attribute #[tactic=tac] to set locally the default tactic to solve remaining holes. The goals on which
the tactic applies are now always of the form  $\Gamma \mid - \tau$  where  $\Gamma$  is the context where the hole was
introduced and  $\tau$  the expected type, even when using the Obligation machinery to solve them,
resulting in a possible incompatibility if the obligation tactic treated the context differently than
the conclusion. By default, the program_simpl tactic performs a simpl call before introducing the
hypotheses, so you might need to add a simpl in * to your tactics.
```

New attributes `#[derive(equations=yes,no, eliminator=yes|no)]` can be used in place of the `(noeqns, noind)` flags which are deprecated.

**News of the Year:** Equations 1.3, released first in September 2021 brings bugfixes, an improved grammar and more robust proof automation tactics.

**URL:** <http://mattam82.github.io/Coq-Equations/>

**Publications:** [hal-01671777](#), [hal-01248807](#), [inria-00628862](#)

**Contact:** Matthieu Sozeau

**Participants:** Matthieu Sozeau, Cyprien Mangin

### 6.1.3 Math-Components

**Name:** Mathematical Components library

**Keyword:** Proof assistant

**Functional Description:** The Mathematical Components library is a set of Coq libraries that cover the prerequisite for the mechanization of the proof of the Odd Order Theorem.

**Release Contributions:** This release is compatible with Coq 8.10, 8.11 and Coq 8.12. The main changes are:

- support for Coq 8.7, 8.8 and 8.9 have been dropped,
- a change of implementation of intervals and the updated theory,
- the addition of kernel lemmas for matrices,
- generalized many lemmas for path and sorted,
- several lemma additions, name changes and bug fixes.

**URL:** <http://math-comp.github.io/math-comp/>

**Contact:** Assia Mahboubi

**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, Yves Bertot

### 6.1.4 Math-comp-analysis

**Name:** Mathematical Components Analysis

**Keyword:** Proof assistant

**Functional Description:** This library adds definitions and theorems for real numbers and their mathematical structures

**Release Contributions:** Compatible with MathComp 1.12.0, and Coq 8.11, 8.12 and 8.13.

**News of the Year:** In 2020, we unified norms and absolute values, added a topology and pseudo-metric on the extended reals, and provided a more complete theory about sequences and measure theory.

**URL:** <https://github.com/math-comp/analysis>

**Publication:** hal-01719918

**Contact:** Cyril Cohen

**Participants:** Cyril Cohen, Georges Gonthier, Marie Kerjean, Assia Mahboubi, Damien Rouhling, Laurence Rideau, Pierre-Yves Strub, Reynald Affeldt

**Partners:** Ecole Polytechnique, AIST Tsukuba

### 6.1.5 MetaCoq

**Keyword:** Coq

**Scientific Description:** The MetaCoq project aims to provide a certified meta-programming environment in Coq. It builds on Template-Coq, a plugin for Coq originally implemented by Malecha (Extensible proof engineering in intensional type theory, Harvard University, 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., in: CoqPL, Paris, France, 2017), as its front-end language, to derive parametricity properties (Anand and Morrisett, in: CoqPL'18, Los Angeles, CA, USA, 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. MetaCoq generalizes it to handle the entire polymorphic calculus of cumulative inductive constructions, 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. The MetaCoq framework 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. Examples of implemented plugins include a parametricity translation and a certified extraction to call-by-value lambda-calculus. The meta-theory of Coq itself is verified in MetaCoq along with verified conversion, type-checking and erasure procedures providing highly trustable alternatives to the procedures in Coq's OCaml kernel. MetaCoq is hence a foundation for the development of higher-level certified tools on top of Coq's kernel. A meta-programming and proving framework for Coq.

MetaCoq is made of 4 main components: - The entry point of the project is the Template-Coq quoting and unquoting library for Coq which allows quotation and denotation of terms between three variants of the Coq AST: the OCaml one used by Coq's kernel, the Coq one defined in MetaCoq and the one defined by the extraction of the MetaCoq AST, allowing to extract OCaml plugins from Coq implementations. - The PCUIC component is a full formalization of Coq's typing and reduction rules, along with proofs of important metatheoretic properties: weakening, substitution, validity, subject reduction and principality. The PCUIC calculus differs slightly from the Template-Coq one and verified translations between the two are provided. - The checker component contains verified implementations of weak-head reduction, conversion and type inference for the PCUIC calculus, along with a verified checker for Coq theories. - The erasure component contains a verified implementation of erasure/extraction from PCUIC to untyped (call-by-value) lambda calculus extended with a dummy value for erased terms.

**Functional Description:** MetaCoq is a framework containing a formalization and verified implementation of Coq's kernel in Coq along with a verified erasure procedure. It provides tools for manipulating Coq terms and developing certified plugins (i.e. translations, compilers or tactics) in Coq.

**Release Contributions:** This version is a beta-release including a fully-functional reification and denotation support, and the verified type-checking and erasure procedures. The metatheory proofs are not entirely completed.

**News of the Year:** The verification of Coq's typechecking and conversion algorithm was completed, resulting in a publication at POPL'20. During this year we improved the erasure procedure, verified completeness in addition to soundness of the conversion algorithm and completed the subject reduction and principality proofs for the PCUIC calculus. MetaCoq was used to show the confluence and subject reduction of an extension of Coq with rewrite rules, presented in an article at POPL'21.

**URL:** <https://metacoq.github.io>

**Publications:** [hal-02901011](#), [hal-02380196](#), [hal-02167423](#), [hal-01809681](#)

**Contact:** Matthieu Sozeau

**Participants:** Abhishek Anand, Danil Annenkov, Meven Bertrand, Jakob Botsch Nielsen, Simon Boulier, Cyril Cohen, Yannick Forster, Kenji Maillard, Gregory Malecha, Matthieu Sozeau, Nicolas Tabareau, Theo Winterhalter

**Partners:** Concordium Blockchain Research Center, Aarhus University, Denmark, Saarland University

### 6.1.6 Coq

**Name:** 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 Contributions:** Coq version 8.14 integrates many usability improvements, as well as an important change in the core language. The main changes include:

- The internal representation of match has changed to a more space-efficient and cleaner structure, allowing the fix of a completeness issue with cumulative inductive types in the type-checker. The internal representation is now closer to the user-level view of match, where the argument context of branches and the inductive binders in and as do not carry type annotations.
- A new coqnative binary performs separate native compilation of libraries, starting from a .vo file. It is supported by coq\_makefile.
- Improvements to typeclasses and canonical structure resolution, allowing more terms to be considered as classes or keys.
- More control over notations declarations and support for primitive types in string and number notations.
- Removal of deprecated tactics, notably omega, which has been replaced by a greatly improved lia, along with many bug fixes.
- New Ltac2 APIs for interaction with Ltac1, manipulation of inductive types and printing.

Many changes and additions to the standard library in the numbers, vectors and lists libraries. A new signed primitive integers library Sint63 is available in addition to the unsigned Uint63 library.

**News of the Year:** Coq version 8.14 integrates many usability improvements, as well as an important change in the core language. See the changelog at <https://coq.inria.fr/refman/changes.html#version-8-14> for an overview of the new features and changes, along with the full list of contributors.

**URL:** <http://coq.inria.fr/>

**Contact:** Matthieu Sozeau

**Participants:** Yves Bertot, Frederic Besson, Tej Chajed, Cyril Cohen, Pierre Corbineau, Pierre Courtieu, Maxime Denes, Jim Fehrle, Julien Forest, Emilio Jesús Gallego Arias, Gaetan Gilbert, Georges Gonthier, Benjamin Grégoire, Jason Gross, Hugo Herbelin, Vincent Laporte, Olivier Laurent, Assia Mahboubi, Kenji Maillard, Érik Martin-Dorel, Guillaume Melquiond, Pierre-Marie Pedrot, Clément Pit-Claudel, Kazuhiko Sakaguchi, Vincent Semeria, Michael Soegtrop, Arnaud Spiwack, Matthieu Sozeau, Enrico Tassi, Laurent Théry, Anton Trunov, Li-Yao Xia, Théo Zimmermann

**Partners:** CNRS, Université Paris-Sud, ENS Lyon, Université Paris-Diderot

### 6.1.7 memprof-limits

**Keyword:** Library

**Scientific Description:** Memprof-limits is an implementation of per-thread global memory limits, and per-thread allocation limits à la Haskell, and CPU-bound thread cancellation, for OCaml, compatible with multiple threads.

Memprof-limits interrupts the execution by raising an asynchronous exception: an exception that can arise at almost any location in the program. It is provided with a guide on how to recover from asynchronous exceptions and other unexpected exceptions, summarising for the first time practical knowledge acquired in OCaml by the Coq proof assistant as well as in other programming languages.

Memprof-limits is probabilistic, as it is based on the statistical memory accountant memprof. It is provided with a statistical analysis that the user can rely on to have guarantees about the enforcement of limits.

**Functional Description:** Memprof-limits is an implementation of (per-thread) global memory limits, (per-thread) allocation limits, and cancellation of CPU-bound threads, for OCaml. Memprof-limits interrupts a computation by raising an exception asynchronously and offers features to recover from them such as interrupt-safe resources.

It is provided with an extensive documentation with examples which explains what must be done to ensure one recovers from an interrupt. This documentation summarises for the first time the experience acquired in OCaml in the Coq proof assistant, as well as in other situations in other programming languages.

**Release Contributions:** Initial version.

**News of the Year:** Version 0.2.0, first official and supported (non-prototype) release.

**URL:** <https://gitlab.com/gadmm/memprof-limits>

**Publication:** [hal-03517592](https://hal.archives-ouvertes.fr/hal-03517592)

**Author:** Guillaume Munch

**Contact:** Guillaume Munch

### 6.1.8 ocaml-boxroot

**Keywords:** Interoperability, Library, Ocaml, Rust

**Scientific Description:** Boxroot is an implementation of roots for the OCaml GC based on concurrent allocation techniques. These roots are designed to support a calling convention to interface between Rust and OCaml code that reconciles the latter’s foreign function interface with the idioms from the former.

**Functional Description:** Boxroot implements fast movable roots for OCaml in C. A root is a data type which contains an OCaml value, and interfaces with the OCaml GC to ensure that this value and its transitive children are kept alive while the root exists. This can be used to write programs in other languages that interface with programs written in OCaml.

**URL:** <https://gitlab.com/ocaml-rust/ocaml-boxroot>

**Contact:** Guillaume Munch

**Participants:** Guillaume Munch, Gabriel Scherer

## 7 New results

### 7.1 Type Theory and Proof Assistants

**Participants:** Antoine Ailloux, Martin Baillon, Gaëtan Gilbert, Meven Lennon-Bertand, Assia Mahboubi, Kenji Maillard, Pierre-Marie Pédro, Loïc Pu-jet, Matthieu Sozeau, Nicolas Tabareau.

#### 7.1.1 Type Theory

**Gradualizing the Calculus of Inductive Constructions.** Acknowledging the ordeal of a fully formal development in a proof assistant such as Coq, we have investigated in [12] gradual variations on the Calculus of Inductive Construction (CIC) for swifter prototyping with imprecise types and terms. We observe, with a no-go theorem, a crucial tradeoff between graduality and the key properties of canonicity, decidability and closure of universes under dependent product that CIC enjoys. Beyond this Fire Triangle of Graduality, we explore the gradualization of CIC with three different compromises, each relaxing one edge of the Fire Triangle. We develop a parametrized presentation of Gradual CIC that encompasses all three variations, and jointly develop their metatheory. We first present a bidirectional elaboration of Gradual CIC to a dependently-typed cast calculus, which elucidates the interrelation between typing, conversion, and graduality. We then establish the metatheory of this cast calculus through both a syntactic model into CIC, which provides weak canonicity, confluence, and when applicable, normalization, and a monotone model that purports the study of the graduality of two of the three variants. This work informs and paves the way towards the development of malleable proof assistants and dependently-typed programming languages.

**Observational Equality: Now For Good.** Building on the recent extension of dependent type theory with a universe of definitionally proof-irrelevant types, we introduce in [7]  $TT^{obs}$ , a new type theory based on the setoidal interpretation of dependent type theory.  $TT^{obs}$  equips every type with an identity relation that satisfies function extensionality, propositional extensionality, and definitional uniqueness of identity proofs (UIP). Compared to other existing proposals to enrich dependent type theory with these principles, our theory features a notion of reduction that is normalizing and provides an algorithmic canonicity result, which we formally prove in Agda using the logical relation framework of Abel et al. Our paper thoroughly develops the meta-theoretical properties of  $TT^{obs}$ , such as the decidability of the conversion and of the type checking, as well as consistency. We also explain how to extend our theory with quotient types, and we introduce a setoidal version of Swan’s Id types that turn it into a proper extension of MLTT with inductive equality.

**Types are Internal  $\infty$ -groupoids.** An alternative to working with model structures is to pursue the idea of pushing synthetic Homotopy Theory further, so as to deal with higher coherences directly inside Type Theory. In [18] we show that, by extending univalent type theory with a universe of definitionally associative and unital polynomial monads, we arrive at a coinductive definition of opetopic type which is able to encode a number of fully coherent algebraic structures. In particular, our approach leads to a definition of  $\infty$ -groupoid internal to type theory and we prove that the type of such  $\infty$ -groupoids is equivalent to the universe of types. That is, every type admits the structure of an  $\infty$ -groupoid internally, and this structure is unique.

**The Multiverse: Logical Modularity for Proof Assistants.** Proof assistants play a dual role as programming languages and logical systems. As programming languages, proof assistants offer standard modularity mechanisms such as first-class functions, type polymorphism and modules. As logical systems, however, modularity is lacking, and understandably so: incompatible reasoning principles—such as univalence and uniqueness of identity proofs—can indirectly lead to logical inconsistency when used in a given development, even when they appear to be confined to different modules. The lack of logical modularity in proof assistants also hinders the adoption of richer programming constructs, such as effects. In [29], we propose the multiverse, a general type-theoretic approach to endow proof assistants with logical modularity. The multiverse consists of multiple universe hierarchies that statically describe the reasoning principles and effects available to define a term at a given type. We identify sufficient conditions for this structuring to modularly ensure that incompatible principles do not interfere, and to locally restrict the power of dependent elimination when necessary. This extensible approach generalizes the ad-hoc treatment of the sort of propositions in the Coq proof assistant. We illustrate the power of the multiverse by describing the inclusion of Coq-style propositions, the strict propositions of Gilbert et al., the exceptional type theory of Pédrot and Tabareau, and general axiomatic extensions of the logic.

**Gardening with the Pythia A model of continuity in a dependent setting.** In [16], we generalize to a rich dependent type theory a proof originally developed by Escardó that all System T functionals are continuous. It relies on the definition of a syntactic model of Baclofen Type Theory, a type theory where dependent elimination must be strict, into the Calculus of Inductive Constructions. The model is given by three translations: the axiom translation, that adds an oracle to the context; the branching translation, based on the dialogue monad, turning every type into a tree; and finally, a layer of algebraic binary parametricity, binding together the two translations. In the resulting type theory, every function  $f : (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N}$  is externally continuous.

### 7.1.2 Proof Assistants

Many of our work on proof assistants are based on the [MetaCoq project](#) [23].

**A Type Theory with Computational Assumptions.** Dependently typed programming languages and proof assistants such as Agda and Coq rely on computation to automatically simplify expressions during type checking. To overcome the lack of certain programming primitives or logical principles in those systems, it is common to appeal to axioms to postulate their existence. However, one can only postulate the bare existence of an axiom, not its computational behaviour. Instead, users are forced to postulate equality proofs and appeal to them explicitly to simplify expressions, making axioms dramatically more complicated to work with than built-in primitives. On the other hand, the equality reflection rule from extensional type theory solves these problems by collapsing computation and equality, at the cost of having no practical type checking algorithm. In [3], we introduce Rewriting Type Theory (RTT), a type theory where it is possible to add computational assumptions in the form of rewrite rules. Rewrite rules go beyond the computational capabilities of intensional type theory, but in contrast to extensional type theory, they are applied automatically so type checking does not require input from the user. To ensure type soundness of RTT—as well as effective type checking—we provide a framework where confluence of user-defined rewrite rules can be checked modularly and automatically, and where adding new rewrite rules is guaranteed to preserve subject reduction. The properties of RTT have been formally verified using the MetaCoq framework and an implementation of rewrite rules is already available in the Agda proof assistant.



**The Marriage of Univalence and Parametricity.** Reasoning modulo equivalences is natural for everyone, including mathematicians. Unfortunately, in proof assistants based on type theory, which are frequently used to mechanize mathematical results and carry out program verification efforts, equality is appallingly syntactic and, as a result, exploiting equivalences is cumbersome at best. Parametricity and univalence are two major concepts that have been explored in the literature to transport programs and proofs across type equivalences, but they fall short of achieving seamless, automatic transport. This work developed in [9] first clarifies the limitations of these two concepts when considered in isolation, and then devises a fruitful marriage between both. The resulting concept, called univalent parametricity, is an extension of parametricity strengthened with univalence that fully realizes programming and proving modulo equivalences. Our approach handles both type and term dependency, as well as type-level computation. In addition to the theory of univalent parametricity, we present a lightweight framework implemented in the Coq proof assistant that allows the user to transparently transfer definitions and theorems for a type to an equivalent one, as if they were equal. For instance, this makes it possible to conveniently switch between an easy-to-reason-about representation and a computationally-efficient representation, as soon as they are proven equivalent. The combination of parametricity and univalence supports transport à la carte: basic univalent transport, which stems from a type equivalence, can be complemented with additional proofs of equivalences between functions over these types, in order to be able to transport more programs and proofs, as well as to yield more efficient terms. We illustrate the use of univalent parametricity on several examples, including a recent integration of native integers in Coq. This work paves the way to easier-to-use proof assistants by supporting seamless programming and proving modulo equivalences.

**Complete Bidirectional Typing for the Calculus of Inductive Constructions.** In [22], we present a bidirectional type system for the Calculus of Inductive Constructions (CIC). It introduces a new judgement intermediate between the usual inference and checking, dubbed constrained inference, to handle the presence of computation in types. The key property of the system is its completeness with respect to the usual undirected one, which has been formally proven in Coq as a part of the MetaCoq project. Although it plays an important role in an ongoing completeness proof for a realistic typing algorithm, the interest of bidirectionality is wider, as it gives insights and structure when trying to prove properties on CIC or design variations and extensions. In particular, we put forward constrained inference, an intermediate between the usual inference and checking judgements, to handle the presence of computation in types.

**Extending the team with a project-specific bot.** While every other software team is adopting off-the-shelf bots to automate everyday tasks, the Coq team has made a different choice by developing and maintaining a project-specific bot from the ground up. In [30], we describe the reasons for this choice, what kind of automation this has allowed us to implement, how the many features of this custom bot have evolved based on internal feedback, and the technology and architecture choices that have made it possible.

## 7.2 Logical Foundations of Programming Languages

**Participants:** Rémi Douence, Hamza Jaafar, Guillhem Jaber, Guillaume Munch-Maccagnoni.

**Games, mobile processes, and functions.** In [21], we establish a tight connection between two models of the  $\lambda$ -calculus, namely Milner's encoding into the  $\pi$ -calculus (precisely, the Internal  $\pi$ -calculus), and operational game semantics (OGS). We first investigate the operational correspondence between the behaviours of the encoding provided by  $\pi$  and OGS. We do so for various LTSs: the standard LTS for  $\pi$  and a new 'concurrent' LTS for OGS; an 'output-prioritised' LTS for  $\pi$  and the standard alternating LTS for OGS. We then show that the equivalences induced on  $\lambda$ -terms by all these LTSs (for  $\pi$  and OGS) coincide. These connections allow us to transfer results and techniques between  $\pi$  and OGS. In particular we import up-to techniques from  $\pi$  onto OGS and we derive congruence and compositionality results for OGS from those of  $\pi$ . The study is illustrated for call-by-value; similar results hold for call-by-name.

**Complete trace models of state and control.** In [20], we consider a hierarchy of four typed call-by-value languages with either higher-order or ground-type references and with either callcc or no control operator. Our first result is a fully abstract trace model for the most expressive setting, featuring both higher-order references and callcc, constructed in the spirit of operational game semantics. Next we examine the impact of suppressing higher-order references and callcc in contexts and provide an operational explanation for the game-semantic conditions known as visibility and bracketing respectively. This allows us to refine the original model to provide fully abstract trace models of interaction with contexts that need not use higher-order references or callcc. Along the way, we discuss the relationship between error- and termination-based contextual testing in each case, and relate the two to trace and complete trace equivalence respectively. Overall, the paper provides a systematic development of operational game semantics for all four cases, which represent the state-based face of the so-called semantic cube.

**Compositional relational reasoning via operational game semantics.** In [19, 27], we show how to use operational game semantics as a guide to develop relational techniques for establishing contextual equivalences with respect to contexts drawn from a hierarchy of four call-by-value higher-order languages: with either general or ground-type references and with either call/cc or no control operator. In game semantics, differences between the contexts can be captured by the absence or presence of the O-visibility and O-bracketing conditions. The proposed technique, which we call Kripke normal-form bisimulations, combines insights from normal-form bisimulation and Kripke logical relations with game semantics. In particular, the role of the heap and the name history is abstracted away using Kripke-style world transition systems. The differences between the four kinds of contexts manifest themselves through simple local conditions that can be shown to correspond to O-visibility and O-bracketing, as applicable. The technique is sound and complete by virtue of correspondence with operational game semantics. Moreover, it sheds a new light on other related developments, such as backtracking and private transitions in Kripke logical relations, which can be related to specific phenomena in game models.

**Theorems for free from separation logic specifications.** Separation logic specifications with abstract predicates intuitively enforce a discipline that constrains when and how calls may be made between a client and a library. Thus a separation logic specification of a library intuitively enforces a protocol on the trace of interactions between a client and the library. In [10], we show how to formalize this intuition and demonstrate how to derive free theorems about such interaction traces from abstract separation logic specifications. We present several examples of free theorems. In particular, we prove that a so-called logically atomic concurrent separation logic specification of a concurrent module operation implies that the operation is linearizable. All the results presented in this paper have been mechanized and formally proved in the Coq proof assistant using the Iris higher-order concurrent separation logic framework.

**Temporal Refinements for Guarded Recursive Types.** In [24, 28], we propose a logic for temporal properties of higher-order programs that handle infinite objects like streams or infinite trees, represented via coinductive types. Specifications of programs use safety and liveness properties. Programs can then be proven to satisfy their specification in a compositional way, our logic being based on a type system. The logic is presented as a refinement type system over the guarded  $\lambda$ -calculus, a  $\lambda$ -calculus with guarded recursive types. The refinements are formulae of a modal  $\mu$ -calculus which embeds usual temporal modal logics. The semantics of our system is given within a rich structure, the topos of trees, in which we build a realizability model of the temporal refinement type system.

**Models of programming languages mixing effects and resources: Resource management in OCaml.** One goal of resource management is to ensure the correctness of programs in the face of failures and interruptions. We have proposed a model for asynchronous interrupt-safety in OCaml and developed a library based on this model that implements resource limits and thread cancellation for OCaml [25].

### 7.3 Program Certifications and Formalisation of Mathematics



**Participants:** Christopher Hughes, Assia Mahboubi.

**A Formal Proof of the Irrationality of  $\zeta(3)$ .** In [13], we present a complete formal verification of a proof that the evaluation of the Riemann zeta function at 3 is irrational, using the Coq proof assistant. This result was first presented by Apéry in 1978, and the proof we have formalized essentially follows the path of his original presentation. The crux of this proof is to establish that some sequences satisfy a common recurrence. We formally prove this result by an a posteriori verification of calculations performed by computer algebra algorithms in a Maple session. The rest of the proof combines arithmetical ingredients and asymptotic analysis, which we conduct by extending the Mathematical Components libraries.

**Unsolvability of the Quintic Formalized in Dependent Type Theory.** In [17], we describe an axiom-free Coq formalization that there does not exist a general method for solving by radicals polynomial equations of degree greater than 4. This development includes a proof of Galois' Theorem of the equivalence between solvable extensions and extensions solvable by radicals. The unsolvability of the general quintic follows from applying this theorem to a well chosen polynomial with unsolvable Galois group.

**Machine-checked computer-aided mathematics.** Proof assistants are pieces of software designed for the realization of digital libraries of formalized mathematics. The latter libraries contain definitions, statements, and proofs, all formalized in a fixed variant of logic. In particular, the verification of the well-formedness of statements, and of the correctness of proofs, boils down to a mechanical process, associated with the underlying logical formalism. The kernel of a proof assistant is the software component which performs this verification, while the actual proof assistant implements a collection of automation techniques, which allow users to conduct in practice the formalization of arbitrarily sophisticated mathematical definitions and theories. The memoir [26] presents an overview of three main contributions to the formal verification of mathematical theories in dependent type theory. The first of these contributions deals with the realization of a library of digitized mathematics covering the standard undergraduate background in algebra, as well as some more advanced chapters in finite group theory. The two other contributions are related to the issues pertaining to the formal verification of computational mathematical proofs, by the means of symbolic algorithms and of rigorous numerical methods respectively.

## 8 Bilateral contracts and grants with industry

**Participants:** Guilhem Jaber, Assia Mahboubi, Guillaume Munch-Maccagnoni, Pierre-Marie Pédro, Matthieu Sozeau, Nicolas Tabareau.

### CoqExtra

**Title:** A Formally Verified Extraction Mechanism using Precise Type Specifications

**Duration:** 2020 - 2023

**Coordinator:** Nicolas Tabareau

#### Partners:

- Inria
- Nomadic Labs

**Inria contact:** Nicolas Tabareau

**Summary:** The extraction mechanism from Coq to OCaml can be seen as a compilation phase, from a functional language with dependent types to a functional language with a weaker type system. It is very useful to be able to run and link critical pieces of code that have been certified with the rest of a software system. For instance, for Tezos, it is important to certify the Michelson language for smart contracts and then to be able to extract it to OCaml so that it interacts with the rest of the code that has been developed. Unfortunately, the current extraction mechanism of Coq suffers from two major flaws that prevent extraction from being used in complex situations—and in particular for the Michelson language. First, the extraction mechanism does not make use of new features of OCaml type system, such as Generalized Abstract Data Types (GADTs). This prevents code using indexed inductive types (Coq’s generalization of GADTs) to be extracted to code using GADTs. Therefore, in the case of Michelson, the extracted code does not correspond at all to the seminal implementation of Michelson in OCaml as it jeopardizes its type specification. The second flaw comes from the fact that extraction sometimes produces ill-typed pieces of code (even if it uses Obj.magic to cheat the type system), for instance when the arity of a function depends on some value. Therefore, the extracted program fails to type-check in OCaml and cannot be used.

**Expected Impact:** This project proposes to remedy to the situation so that the formalized Michelson implementation can be extracted to OCaml in a satisfactory and certified way. But this project is also of great interest outside Nomadic Labs as it will allow Coq users to use a better extraction mechanism and, on a longer term, it will allow OCaml developers to prove their OCaml programs using a formal semantics of (a fragment of) OCaml defined in Coq.

#### **CIFRE PhD grant, funded by Mitsubishi Electric R&D Centre Europe (MERCE)**

**Title:** Automated theorem proving and dependent types: automated reasoning for interactive proof assistants

**Duration:** 2020 - 2023

**Coordinator:** Denis Cousineau (MERCE), Assia Mahboubi (Inria)

#### **Partners:**

- Inria
- Mitsubishi Electric R&D Centre Europe (MERCE)

**Inria contact:** Assia Mahboubi

**Summary:** The aim of this project is to vastly improve the automated reasoning skills of proof assistants based on dependent type theory, and in particular of the Coq proof assistant. Automated provers, like SAT solvers or SMT solvers, can provide fast decision answers on large formulas, typically quantifier-free first order statements generated by code analysis instruments like static analyzers. Modern provers are moreover able to produce additional data, called certificates, which contain enough information for an a posteriori verification of their results, e.g., using a formal proof. In this project, we would like to use this feature to expand the automation available to users of proof assistants. The main motivation here is thus to increase the class of goals that can be proved formally and automatically by the interactive proof assistant, rather than to work on the formal verification of specific albeit large decision problems. In this case, the central research problem is to bridge the gap between the rich specification language of the proof assistant, and the restricted fragment handled by the automated prover. This project will thus investigate the design, and the implementation, of the corresponding translation phase. This translation transforms a logical statement possibly featuring user-defined data structures and higher-order quantifications, into another statement, logically stronger, than can be sent to the automated prover. We thus aim at a triple objective: expressivity, extensibility and efficiency. This grant is funding the PhD of Enzo Crance.

**Expected Impact:** Enhancing the automated reasoning skills of proof assistants based on dependent type theory will be key to their wider usage in industry. As of today, they are considered too expensive to be used in the large outside of specific niches.

### OCaml-Rust

**Title:** OCaml/Rust bindings

**Duration:** 2021-2023

**Coordinator:** Gabriel Scherer (INRIA Saclay, EPI Partout)

**Participants:** Guillaume Munch-Maccagnoni (INRIA Rennes, EPI Gallinette), Jacques-Henri Jourdan (CNRS, LRI)

**Partners:** Inria, Nomadic Labs

**Inria contact:** Gabriel Scherer

**Summary:** We often want to write programs with components in several different programming languages. Interfacing two languages typically goes through low-level, unsafe interfaces. The OCaml/Rust project studies safer interfaces between OCaml and Rust.

**Expected Impact:** We investigated safe low-level representations of OCaml values on the Rust side, representing GC ownership, and developed a calling convention that reconciles the OCaml FFI idioms with Rust idioms. We also developed Boxroot, a new API to register values with the OCaml GC, for use when interfacing with Rust (and other programming languages) and possibly when writing concurrent programs. This resulted in novel techniques which can benefit other pairs of languages in the future. These works are now integrated in the `ocaml-rs` interface between OCaml and Rust used in the industry.

### CAVOC

**Title:** Compositional Automated Verification for OCaml

**Duration:** 2021-2024

**Coordinator:** Guilhem Jaber

**Partners:**

- Inria
- Nomadic Labs

**Inria contact:** Guilhem Jaber

**Summary:** This project aims to develop a *sound and precise static analyzer* for OCaml, that can catch large classes of bugs represented by uncaught exceptions. It will deal with both user-defined exceptions, and built-in ones used to represent *error behaviors*, like the ones triggered by `failwith`, `assert`, or a match failure. Via “assert-failure” detection, it will thus be able to check that invariants annotated by users hold. The analyzer will reason *compositionally* on programs, in order to analyze them at the granularity of a function or of a module. It will be *sound* in a strong way: if an OCaml module is considered to be correct by the analyzer, then one will have the guarantee that no OCaml code interacting with this module can trigger uncaught exceptions coming from the code of this module. In order to be *precise*, it will take into account the abstraction properties provided by the type system and the module system of the language: local values, abstracted definition of types, parametric polymorphism. The goal being that most of the interactions taken into account correspond to typeable OCaml code (that do not use unsafe features of the Obj Module, or the Foreign Function Interface to some external code).

**Expected Impact:** Being modular the analyzer should be able to automatically check the absence of bugs of a large base of code written in the considered subset of OCaml. This subset will include most of the codebase developed by Nomadic Labs, which is an heavy user of GADT, for example to enforce subject reduction in the implementation of Michelson. We would then be able to get a higher degree of trust in its codebase, and possibly to find undetected bugs in it. The impact of this project could be large for the OCaml ecosystem in general, where automated analysis of programs to check soundness properties of the code could be really useful (for example for the Coq proof assistant, whose full analysis would be nonetheless too ambitious for this project).

## 9 Partnerships and cooperations

**Participants:** Yannick Forster, Guilhem Jaber, Assia Mahboubi, Guillaume Munch-Maccagnoni, Pierre-Marie Pédro, Matthieu Sozeau, Nicolas Tabareau.

### 9.1 International initiatives

#### 9.1.1 Associate Teams in the framework of an Inria International Lab or in the framework of an Inria International Program

##### GECO

**Title:** Gradual verification and robust proof Engineering for COq

**Duration:** 2018 -> 2021

**Coordinator:** Éric Tanter (etanter@dcc.uchile.cl)

##### Partners:

- Universidad de Chile

**Inria contact:** Nicolas Tabareau

#### 9.1.2 Participation in other International Programs

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

### 9.2 International research visitors

#### 9.2.1 Visits of international scientists

##### Other international visits to the team

##### Éric Tanter

**Status** Full Professor

**Institution of origin:** Universidad de Chile

**Country:** Chile

**Dates:** Oct 2021

**Context of the visit:** Inria GECO associate team

**Mobility program/type of mobility:** research stay

## 9.3 European initiatives

### 9.3.1 FP7 & H2020 projects

#### Fresco

**Title:** Fast and Reliable Symbolic Computation

**Program:** H2020

**Type:** ERC COG

**Duration:** Nov 2021 - Oct 2026

**Coordinator:** Inria

**Inria Contact:** Assia Mahboubi

**Summary:** Using computers to formulate conjectures and consolidate proof steps pervades all mathematics fields, even the most abstract. Most computer proofs are produced by symbolic computations, using computer algebra systems. However, these systems suffer from severe, intrinsic flaws, rendering computational correction and verification challenging. The EU-funded FRESKO project aims to shed light on whether computer algebra could be both reliable and fast. Researchers will disrupt the architecture of proof assistants, which serve as the best tools for representing mathematics in silico, enriching their programming features while preserving their compatibility with their logical foundations. They will also design novel mathematical software that should feature a high-level, performance-oriented programming environment for writing efficient code to boost computational mathematics.

#### Coqaml

**Title:** Verified Extraction from Coq to OCaml with GADTs

**Program:** H2020

**Type:** Marie Skłodowska-Curie individual fellowship

**Duration:** Dec 2021 - Nov 2023

**Coordinator:** Inria

**Inria Contact:** Nicolas Tabareau

**Summary:** The Coq proof assistant is a popular tool to verify the correctness of security-critical software. The CompCert C compiler, some implementations of blockchain languages, and the implementation of the P-256 elliptic curve in Google's BoringSSL library are all OCaml programs obtained by extraction from Coq functions. While a type checker for Coq has recently been verified via a machine-checked mathematical proof based on the MetaCoq project for verified meta-programming, the extraction process from Coq to OCaml is still part of the trusted computing base (TCB). The Coqaml project will minimise the TCB for extracted programs even further by also providing a machine-checked correctness proof for the extraction mechanism to OCaml. Under the supervision of Nicolas Tabareau, head of the Inria Gallinette team in Nantes, the experienced researcher will implement Coq's extraction as mechanically verified MetaCoq-plugin, obtaining the guarantee that extracted OCaml programs behave exactly like the Coq function specified.

## 9.4 National initiatives

### NUSCAP

**Title:** Numerical Safety for Computer-Aided Proofs

**Program:** ANR AAPG2020,

**Type:** PRC, CES 48

**Duration:** Feb 2021 - Jan 2024

**Coordinator:** UMR CNRS - ENS Lyon - UCB Lyon 1 - INRIA 5668

**Local Contact:** Assia Mahboubi

**Summary:** The last twenty years have seen the advent of computer-aided proofs in mathematics and this trend is getting more and more important. They request various levels of numerical safety, from fast and stable computations to formal proofs of the computations. However, the necessary tools and routines are usually ad hoc, sometimes unavailable, or inexistent. On a complementary perspective, numerical safety is also critical for complex guidance and control algorithms, in the context of increased satellite autonomy. We plan to design a whole set of theorems, algorithms and software developments, that will allow one to study a computational problem on all (or any) of the desired levels of numerical rigor. Key developments include fast and certified spectral methods and polynomial arithmetic, with subsequent formal verifications. There will be a strong feedback between the development of our tools and the applications that motivate it.

### ReCiProg

**Title:** Reasoning on Circular proofs for Programming

**Program:** ANR AAPG2021,

**Type:** PRC, CES 48

**Duration:** Jan 2022 - Jan 2025

**Coordinator:** UMR CNRS - IRIF - Université de Paris

**Local Contact:** Guilhem Jaber

**Summary:** ReCiProg is a collaborative project (Lyon-Marseille-Nantes-Paris) aiming at extending the proofs-as-programs correspondence (also known as Curry-Howard correspondence) to recursive programs and circular proofs for logic and type systems using induction and coinduction. The project will contribute both to the necessary theoretical foundations of circular proofs and to the software development allowing to enhance the use of coinductive types and coinductive reasoning in the Coq proof assistant: such coinductive types present, in the current state of the art serious defects that the project will aim at solving.

### DyVerSe

**Title:** Dynamic Versatile Semantics

**Program:** ANR AAPG2019,

**Type:** PRC, CES 48

**Duration:** Jan 2020 - Dec 2023

**Coordinator:** Pierre Clairambault (CR CNRS, LIP, UMR 5668)

**Local Contact:** Guillaume Munch-Maccagnoni

**Summary:** DyVerSe aims to develop a theoretical framework for dynamic/game semantics for programming languages, capturing in one versatile setting a spectrum of computational features, representative of the heterogeneity of software (e.g. higher-order functions, concurrency, probabilities or other quantitative aspects). Our ambition is (1) to help unify denotational semantics by providing the missing link between various incompatible models focusing on specific aspects, and (2) to provide a toolbox to reason compositionally about the dynamic behaviour of programs, with an eye towards specification and verification.

## 9.5 Regional initiatives

### Vercoma

Atlantstic 2020/Attractivity grant

**Goal:** Verified computer mathematics

**Coordinator:** Assia Mahboubi.

**Duration:** 08/2018 - 12/2021.

### ASCOC

Atlantstic 2020/Amorçage grant

**Goal:** Compositional Analysis of OCaml code

**Coordinator:** G. Jaber.

**Duration:** 09/2020 - 12/2022

## 10 Dissemination

This section involves all the permanent members of the team.

### 10.1 Promoting scientific activities

#### 10.1.1 Scientific events: organisation

##### The Seventh International Workshop on Coq for Programming Languages

**Date:** January 19 2021, fully online.

**Sponsorship:** ACM SIGPLAN, ACM SIGLOG

**Website**

**YouTube**

##### Spring School on Homotopy Type Theory

**Date:** April 12-16 2021, fully online.

**Sponsorship:** ACM SIGPLAN, ACM SIGLOG and Nomadic Labs.

**Website**

**YouTube channel**

**Github**

**Member of the organizing committees** The Gallinette team is organizing **TYPES'22** in Nantes, 20-25th June 2022.

### 10.1.2 Scientific events: selection

#### Member of the conference program committees

- Matthieu Sozeau was a Program Committee member of POPL 2021 and ITP 2021.
- Assia Mahboubi was a Program Committee member of ITP 2021, CADE 2021, CIE 2021, JFLA 2021 and of the TYPES international workshop.
- Pierre-Marie Pédrot was a Program Committee member of WITS 2022.

#### Reviewer

- Pierre-Marie Pédrot has reviewed articles for LICS 2021.
- Guillaume Munch-Maccagnoni has reviewed an article for the TYPES post-proceedings.

### 10.1.3 Journal

#### Member of the editorial boards

- Assia Mahboubi is a member of the editorial board of the Journal of Automated Reasoning.

#### Reviewer - reviewing activities

- Matthieu Sozeau has reviewed articles for the Journal of Functional Programming.
- Pierre-Marie Pédrot has reviewed articles for the Bulletin of Symbolic Logic.
- Guillaume Munch-Maccagnoni has reviewed articles for the Logical Methods in Computer Science and Mathematical Structures in Computer Science journals.

### 10.1.4 Invited talks

- Matthieu Sozeau gave invited talks at the **LFMTP'21** workshop and the **HoTTest seminar** on Meta-Coq.
- Assia Mahboubi gave invited talks at the CSL'21 conference, at the FM'21 conference, at the Journées du GDR Informatique Mathématique, at the colloquium of mathematics of the university of Ljubljana (Slovenia), at the chocola seminar (Lyon, France), at the colloquium of the School of Computer and Cyber Sciences at university of Alberta (USA), at the OWLS on-line seminar.
- Pierre-Marie Pédrot gave an invited talk at TYPES 2021.

### 10.1.5 Leadership within the scientific community

- Matthieu Sozeau has been elected as a member of the **TYPES conference** Steering Committee.
- Assia Mahboubi is a member of the steering committee of the CPP conference.



## 10.2 Teaching - Supervision - Juries

### 10.2.1 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 software 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, Object Oriented Design and Programming Project, 30h, L1 (apprenticeship), IMT-Atlantique, Nantes, France
- Master : Rémi Douence, Functional Programming with Haskell, 45h, M1 (engineers), IMT-Atlantique, Nantes, France
- Master : Rémi Douence, Functional Programming with Haskell, 20h, M1 (apprenticeship), IMT-Atlantique, Nantes, France
- Master : Rémi Douence, Formal Methods: Model checking with Alloy and from Haskell to Coq, 11h, M1 (apprenticeship), IMT-Atlantique, Nantes, France
- Master : Rémi Douence, Introduction to scientific research in computer science (Project: compilation in Java of Haskell Class Types), 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, 18h, L1, Université de Nantes France
- Licence : Guilhem Jaber, Foundations of Computer Science, 54h, L3, Université de Nantes France
- Licence : Guilhem Jaber, Logic in Computer Science, 48h, L2, Université de Nantes France
- Licence : Guilhem Jaber, Functional Programming, 36h, L3, 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
- Master: Assia Mahboubi, Machine-Checked Mathematics, 22.5h, M2, Vrije Universiteit Amsterdam, the Netherlands
- Master : Matthieu Sozeau, Proof Assistants, 24h, M2 MPRI, Université Paris Diderot, France

### 10.2.2 Supervision

- PhD in progress: Xavier Montillet, Rewriting and solvability for Call-by-push-value, Univ Nantes, advisors: Guillaume Munch-Maccagnoni 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: Loïc Pujet, Giving meaning to cubical type theory using forcing, Univ Nantes, advisors: Nicolas Tabareau
- PhD in progress: Meven Bertrand, Gradualizing the calculus of constructions, Univ Nantes, advisors: Nicolas Tabareau
- PhD in progress: Martin Baillon, Syntactic Models of Type Theory and Continuity Principles, Univ Nantes, advisors: Assia Mahboubi and Pierre-Marie Pédrot
- PhD in progress: Pierre Benjamin Giraud, Formalizing extraction of Coq to OCaml, Univ Nantes, advisors: Pierre-Marie Pédrot, Matthieu Sozeau and Nicolas Tabareau
- PhD in progress: Enzo Crance, Automated theorem proving and dependent types: automated reasoning for interactive proof assistants, Univ Nantes, advisors: Denis Cousineau and Assia Mahboubi
- PhD in progress: Antoine Allieux, Coherent Higher Structures in Homotopy Type Theory, Univ Paris Diderot, advisors: Pierre-Louis Curien (Univ. Paris Diderot), Eric Finster (Univ. Birmingham) and Matthieu Sozeau
- PhD in progress: Christopher Hughes, Transport of mathematical properties in type theory: structures, morphisms, refinements, Univ Nantes, advisors: Cyril Cohen (Inria Sophia Antipolis Méditerranée) and Assia Mahboubi
- PhD in progress: Hamza Jaafar, Operational Game Semantics for OCaml, Univ. Nantes, advisor: Guilhem Jaber and Nicolas Tabareau
- PhD in progress: Peio Borthelle, Formalized Game Semantics for Interoperability, Univ. Savoie Mont-Blanc, advisor: Tom Hirschowitz and Guilhem Jaber

#### Supervision of interns

- L3 intern from ENS Paris Saclay. June-July 2021. Thomas Lamiaux. Univalent Parametricity for inductive types. Advisor: Nicolas Tabareau

## 10.3 Popularization

### 10.3.1 Interventions

- Guilhem Jaber and Assia Mahboubi participated to an artists residence at the Athenor theater (Saint-Nazaire, France). They participated to three performances in the frame of the Instants Fertiles #9 festival.

## 11 Scientific production

### 11.1 Major publications

- [1] R. Affeldt, C. Cohen, M. Kerjean, A. Mahboubi, D. Rouhling and K. Sakaguchi. ‘Competing inheritance paths in dependent type theory: a case study in functional analysis’. In: *IJCAR 2020 - International Joint Conference on Automated Reasoning*. Paris, France, June 2020, pp. 1–19. URL: <https://hal.inria.fr/hal-02463336>.

- [2] B. Ahrens, A. Hirschowitz, A. Lafont and M. Maggesi. ‘Reduction Monads and Their Signatures’. In: *Proceedings of the ACM on Programming Languages* (Jan. 2020), pp. 1–29. DOI: [10.1145/3371099](https://doi.org/10.1145/3371099). URL: <https://hal.inria.fr/hal-02380682>.
- [3] J. Cockx, N. Tabareau and T. Winterhalter. ‘The Taming of the Rew: A Type Theory with Computational Assumptions’. In: *Proceedings of the ACM on Programming Languages*. POPL 2021 (2021). DOI: [10.1145/3434341](https://doi.org/10.1145/3434341). URL: <https://hal.archives-ouvertes.fr/hal-02901011>.
- [4] G. Jaber. ‘SyTeCi: Automating Contextual Equivalence for Higher-Order Programs with References’. In: *Proceedings of the ACM on Programming Languages* 28 (2020), pp. 1–28. DOI: [10.1145/3371127](https://doi.org/10.1145/3371127). URL: <https://hal.archives-ouvertes.fr/hal-02388621>.
- [5] P.-M. Pédrot. ‘Russian Constructivism in a Prefascist Theory’. In: *LICS 2020 - Thirty-Fifth Annual ACM/IEEE Symposium on Logic in Computer Science*. Saarbrücken, Germany: IEEE, July 2020, pp. 1–14. DOI: [10.1145/3373718.3394740](https://doi.org/10.1145/3373718.3394740). URL: <https://hal.inria.fr/hal-02548315>.
- [6] P.-M. Pédrot and N. Tabareau. ‘The Fire Triangle’. In: *Proceedings of the ACM on Programming Languages* (Jan. 2020), pp. 1–28. DOI: [10.1145/3371126](https://doi.org/10.1145/3371126). URL: <https://hal.archives-ouvertes.fr/hal-02383109>.
- [7] L. Pujet and N. Tabareau. ‘Observational Equality: Now For Good’. In: POPL. Philadelphie, United States, 17th Jan. 2022. URL: <https://hal.inria.fr/hal-03367052>.
- [8] M. Sozeau, S. Boulier, Y. Forster, N. Tabareau and T. Winterhalter. ‘Coq Coq Correct! Verification of Type Checking and Erasure for Coq, in Coq’. In: *Proceedings of the ACM on Programming Languages* (Jan. 2020), pp. 1–28. DOI: [10.1145/3371076](https://doi.org/10.1145/3371076). URL: <https://hal.archives-ouvertes.fr/hal-02380196>.
- [9] N. Tabareau, É. Tanter and M. Sozeau. ‘The Marriage of Univalence and Parametricity’. In: *Journal of the ACM (JACM)* 68.1 (15th Jan. 2021), pp. 1–44. DOI: [10.1145/3429979](https://doi.org/10.1145/3429979). URL: <https://hal.inria.fr/hal-03120580>.

## 11.2 Publications of the year

### International journals

- [10] L. Birkedal, T. Dinsdale-Young, A. Guéneau, G. Jaber, K. Svendsen and N. Tzevelekos. ‘Theorems for free from separation logic specifications’. In: *Proceedings of the ACM on Programming Languages* 5.ICFP (22nd Aug. 2021), pp. 1–29. DOI: [10.1145/3473586](https://doi.org/10.1145/3473586). URL: <https://hal.archives-ouvertes.fr/hal-03510684>.
- [11] J. Cockx, N. Tabareau and T. Winterhalter. ‘The Taming of the Rew: A Type Theory with Computational Assumptions’. In: *Proceedings of the ACM on Programming Languages*. POPL 2021 (2021). DOI: [10.1145/3434341](https://doi.org/10.1145/3434341). URL: <https://hal.archives-ouvertes.fr/hal-02901011>.
- [12] M. Lennon-Bertrand, K. Maillard, N. Tabareau and É. Tanter. ‘Gradualizing the Calculus of Inductive Constructions’. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* (2022). DOI: [10.1145/3495528](https://doi.org/10.1145/3495528). URL: <https://hal.archives-ouvertes.fr/hal-02896776>.
- [13] A. Mahboubi and T. Sibut-Pinote. ‘A formal proof of the irrationality of  $\zeta(3)$ ’. In: *Logical Methods in Computer Science* (18th Feb. 2021). DOI: [10.23638/LMCS-17\(1:16\)2021](https://doi.org/10.23638/LMCS-17(1:16)2021). URL: <https://hal.inria.fr/hal-03517003>.
- [14] L. Pujet and N. Tabareau. ‘Observational Equality: Now For Good’. In: *Proceedings of the ACM on Programming Languages* 6.POPL (15th Jan. 2022). DOI: [10.1145/3498693](https://doi.org/10.1145/3498693). URL: <https://hal.inria.fr/hal-03367052>.
- [15] N. Tabareau, É. Tanter and M. Sozeau. ‘The Marriage of Univalence and Parametricity’. In: *Journal of the ACM (JACM)* 68.1 (15th Jan. 2021), pp. 1–44. DOI: [10.1145/3429979](https://doi.org/10.1145/3429979). URL: <https://hal.inria.fr/hal-03120580>.

### International peer-reviewed conferences

- [16] M. Baillon, A. Mahboubi and P.-M. Pédrot. ‘Gardening with the PythiaA model of continuity in a dependent setting’. In: *Computer Science Logic*. Göttingen, Germany, 14th Feb. 2022. DOI: [10.4230/LIPIcs.CSL.2022.13](https://doi.org/10.4230/LIPIcs.CSL.2022.13). URL: <https://hal.inria.fr/hal-03510671>.
- [17] S. Bernard, C. Cohen, A. Mahboubi and P.-Y. Strub. ‘Unsolvability of the Quintic Formalized in Dependent Type Theory’. In: *ITP 2021 - 12th International Conference on Interactive Theorem Proving*. Rome / Virtual, France, 29th June 2021. URL: <https://hal.inria.fr/hal-03136002>.
- [18] E. Finster, A. Allieux and M. Sozeau. ‘Types are internal infinity-groupoids’. In: *LICS 2021*. Rome, Italy, 21st June 2021. URL: <https://hal.inria.fr/hal-03133144>.
- [19] G. Jaber and A. Murawski. ‘Compositional relational reasoning via operational game semantics’. In: *LICS 2021 - 36th Annual ACM/IEEE Symposium on Logic in Computer Science. Proceedings of the 36th Annual ACM/IEEE Symposium on Logic in Computer Science 23*. Rome, Italy: IEEE, 29th June 2021, pp. 1–13. DOI: [10.1109/LICS52264.2021.9470524](https://doi.org/10.1109/LICS52264.2021.9470524). URL: <https://hal.archives-ouvertes.fr/hal-03510294>.
- [20] G. Jaber and A. S. Murawski. ‘Complete trace models of state and control’. In: *ESOP 2021 - 30th European Symposium on Programming*. Vol. 12648. Lecture Notes in Computer Science. Luxembourg, Luxembourg: Springer International Publishing, 23rd Mar. 2021, pp. 348–374. DOI: [10.1007/978-3-030-72019-3\\_13](https://doi.org/10.1007/978-3-030-72019-3_13). URL: <https://hal.archives-ouvertes.fr/hal-03510374>.
- [21] G. Jaber and D. Sangiorgi. ‘Games, mobile processes, and functions’. In: *30th EACSL Annual Conference on Computer Science Logic (CSL 2022)*. Göttingen, Germany, 28th Oct. 2021. URL: <https://hal.archives-ouvertes.fr/hal-03407123>.
- [22] M. Lennon-Bertrand. ‘Complete Bidirectional Typing for the Calculus of Inductive Constructions’. In: *12th International Conference on Interactive Theorem Proving (ITP 2021)*. Vol. 193. Leibniz International Proceedings in Informatics (LIPIcs) 24. Rome, Italy, 21st June 2021, 24:1–24:19. DOI: [10.4230/LIPIcs.ITP.2021.24](https://doi.org/10.4230/LIPIcs.ITP.2021.24). URL: <https://hal.archives-ouvertes.fr/hal-03139924>.
- [23] M. Sozeau. ‘Touring the MetaCoq Project (Invited Paper)’. In: *LFMTP 2021 - Logical Frameworks and Meta-Languages: Theory and Practice*. Pittsburg, United States, 16th July 2021. URL: <https://hal.inria.fr/hal-03516619>.

### Conferences without proceedings

- [24] G. Jaber and C. Riba. ‘Temporal Refinements for Guarded Recursive Types’. In: *ESOP 2021 - 30th European Symposium on Programming*. Vol. 12648. Lecture Notes in Computer Science. 2021-04-01, Luxembourg: Springer International Publishing, 23rd Mar. 2021, pp. 548–578. DOI: [10.1007/978-3-030-72019-3\\_20](https://doi.org/10.1007/978-3-030-72019-3_20). URL: <https://hal.archives-ouvertes.fr/hal-03517430>.
- [25] G. Munch-Maccagnoni. ‘Probabilistic resource limits using StatMemprof’. In: *OCaml Workshop 2021*. Online, France, 27th Aug. 2021, p. 2. URL: <https://hal.inria.fr/hal-03517592>.

### Doctoral dissertations and habilitation theses

- [26] A. Mahboubi. ‘Machine-checked computer-aided mathematics’. Université de Nantes (UN), Nantes, FRA., 5th Jan. 2021. URL: <https://tel.archives-ouvertes.fr/tel-03107626>.

### Reports & preprints

- [27] G. Jaber and A. S. Murawski. *Complete trace models of state and control (full version)*. 20th Jan. 2021. URL: <https://hal.archives-ouvertes.fr/hal-03116698>.
- [28] G. Jaber and C. Riba. *Temporal Refinements for Guarded Recursive Types (full version)*. 7th Jan. 2021. URL: <https://hal.archives-ouvertes.fr/hal-02512655>.
- [29] K. Maillard, N. Margulies, M. Sozeau, N. Tabareau and É. Tanter. *The Multiverse: Logical Modularity for Proof Assistants*. 23rd Aug. 2021. URL: <https://hal.inria.fr/hal-03324596>.

- [30] T. Zimmermann, J. Coolen, J. Gross, P.-M. Pédrot and G. Gilbert. *Extending the team with a project-specific bot*. 14th Dec. 2021. URL: <https://hal.inria.fr/hal-03479327>.

### 11.3 Cited publications

- [31] A. Abel and T. Coquand. ‘Untyped Algorithmic Equality for Martin-Löf’s Logical Framework with Surjective Pairs’. English. In: *Typed Lambda Calculi and Applications*. Ed. by P. Urzyczyn. Vol. 3461. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, pp. 23–38. DOI: [10.1007/11417170\\_4](https://doi.org/10.1007/11417170_4). URL: [http://dx.doi.org/10.1007/11417170\\_4](http://dx.doi.org/10.1007/11417170_4).
- [32] B. Accattoli and G. Guerrieri. ‘Open Call-by-Value’. In: *Programming Languages and Systems* (1st Jan. 2016). DOI: [10.1007/978-3-319-47958-3\\_12](https://doi.org/10.1007/978-3-319-47958-3_12). URL: [http://dx.doi.org/10.1007/978-3-319-47958-3\\_12](http://dx.doi.org/10.1007/978-3-319-47958-3_12).
- [33] D. Ahman, N. Ghani and G. D. Plotkin. ‘Dependent Types and Fibred Computational Effects’. In: *Proc. FoSSaCS*. 2015.
- [34] A. Ajouli, J. Cohen and 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. DOI: [10.1109/SEAA.2013.53](https://doi.org/10.1109/SEAA.2013.53). URL: <http://dx.doi.org/10.1109/SEAA.2013.53>.
- [35] 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*. Ed. by A. L. Hosking, P. T. Eugster and R. Hirschfeld. ACM, 2013, pp. 101–116.
- [36] T. Altenkirch, C. McBride and W. Swierstra. ‘Observational equality, now!’ In: *Proceedings of the ACM Workshop on Programming Languages meets Program Verification (PLPV 2007)*. Freiburg, Germany, Oct. 2007, pp. 57–68.
- [37] J.-M. Andreoli. ‘Logic Programming with Focusing Proof in Linear Logic’. In: *Journal of Logic and Computation* 2.3 (1992), pp. 297–347.
- [38] E. Arafailova. ‘Functional description of sequence constraints and synthesis of combinatorial objects’. Theses. Ecole nationale supérieure Mines-Télécom Atlantique, Sept. 2018. URL: <https://tel.archives-ouvertes.fr/tel-01962957>.
- [39] E. Arafailova, N. Beldiceanu, R. Douence, M. Carlsson, P. Flener, M. A. F. Rodríguez, J. Pearson and H. Simonis. ‘Global Constraint Catalog, Volume II, Time-Series Constraints’. In: *CoRR* abs/1609.08925 (2016). URL: <http://arxiv.org/abs/1609.08925>.
- [40] E. Arafailova, N. Beldiceanu, R. Douence, P. Flener, M. A. F. Rodríguez, J. Pearson and 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*. Ed. by C.-G. Quimper. Vol. 9676. Lecture Notes in Computer Science. Springer, 2016, pp. 18–34. DOI: [10.1007/978-3-319-33954-2\\_2](https://doi.org/10.1007/978-3-319-33954-2_2). URL: [https://doi.org/10.1007/978-3-319-33954-2\\_2](https://doi.org/10.1007/978-3-319-33954-2_2).
- [41] A. F. Barco, J.-G. Fages, É. Vareilles, M. Aldanondo and 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*. Ed. by G. Pesant. Vol. 9255. Lecture Notes in Computer Science. Springer, 2015, pp. 508–523. DOI: [10.1007/978-3-319-23219-5\\_36](https://doi.org/10.1007/978-3-319-23219-5_36). URL: [http://dx.doi.org/10.1007/978-3-319-23219-5\\_36](http://dx.doi.org/10.1007/978-3-319-23219-5_36).
- [42] N. Beldiceanu, M. Carlsson, R. Douence and H. Simonis. ‘Using finite transducers for describing and synthesising structural time-series constraints’. In: *Constraints* 21.1 (2016), pp. 22–40. DOI: [10.1007/s10601-015-9200-3](https://doi.org/10.1007/s10601-015-9200-3). URL: <http://dx.doi.org/10.1007/s10601-015-9200-3>.
- [43] S. Berardi, M. Bezem and T. Coquand. ‘On the computational content of the axiom of choice’. In: *The Journal of Symbolic Logic* 63.02 (1998), pp. 600–622.

- [44] M. Bezem, T. Coquand and S. Huber. ‘A model of type theory in cubical sets’. In: *Preprint, September* (2013).
- [45] S. Boulrier, P.-M. Pédrot and N. Tabareau. ‘The next 700 syntactical models of type theory’. In: *Certified Programs and Proofs (CPP 2017)*. Paris, France, Jan. 2017, pp. 182–194. DOI: [10.1145/3018610.3018620](https://doi.org/10.1145/3018610.3018620). URL: <https://hal.inria.fr/hal-01445835>.
- [46] E. Brady. ‘Idris, a general-purpose dependently typed programming language: Design and implementation’. In: *J. Funct. Program.* 23.5 (2013), pp. 552–593. DOI: [10.1017/S095679681300018X](https://doi.org/10.1017/S095679681300018X). URL: <https://doi.org/10.1017/S095679681300018X>.
- [47] F. Chyzak, A. Mahboubi, T. Sibut-Pinote and E. Tassi. ‘A Computer-Algebra-Based Formal Proof of the Irrationality of  $\zeta(3)$ ’. In: *Interactive Theorem Proving*. Ed. by R. G. Gerwin Klein. Vol. 8558. Lecture Notes in Computer Science. Springer, 2014.
- [48] C. Cohen, T. Coquand, S. Huber and A. Mörtberg. *Cubical Type Theory: a constructive interpretation of the univalence axiom*. To appear in post-proceedings of Types for Proofs and Programs (TYPES 2015). 2016.
- [49] J. Cohen, R. Douence and A. Ajouli. ‘Invertible Program Restructurings for Continuing Modular Maintenance’. In: *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*. Mar. 2012, pp. 347–352. DOI: [10.1109/CSMR.2012.42](https://doi.org/10.1109/CSMR.2012.42). URL: <http://dx.doi.org/10.1109/CSMR.2012.42>.
- [50] 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*. Ed. by G. Hamilton, A. Lisitsa and A. P. Nemytykh. Vol. 216. Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, 2016, pp. 50–64. DOI: [10.4204/EPTCS.216.3](https://doi.org/10.4204/EPTCS.216.3). URL: <http://dx.doi.org/10.4204/EPTCS.216.3>.
- [51] J. Cohen and A. Ajouli. ‘Practical Use of Static Composition of Refactoring Operations’. In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13*. Coimbra, Portugal: ACM, 2013, pp. 1700–1705. DOI: [10.1145/2480362.2480684](https://doi.org/10.1145/2480362.2480684). URL: <http://dx.doi.org/10.1145/2480362.2480684>.
- [52] P. Cohen and M. Davis. *Set theory and the continuum hypothesis*. WA Benjamin New York, 1966.
- [53] 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*. Ed. by S. Arora and G. T. Leavens. ACM, 2009, pp. 557–572. DOI: [10.1145/1640089.1640133](https://doi.org/10.1145/1640089.1640133). URL: <http://doi.acm.org/10.1145/1640089.1640133>.
- [54] Coq Development Team, The. *The Coq proof assistant reference manual*. Version 8.5. 2015. URL: <http://coq.inria.fr>.
- [55] P.-L. Curien, M. Fiore and G. Munch-Maccagnoni. ‘A Theory of Effects and Resources: Adjunction Models and Polarised Calculi’. In: *Proc. POPL*. 2016. DOI: [10.1145/2837614.2837652](https://doi.org/10.1145/2837614.2837652). URL: <http://dx.doi.org/10.1145/2837614.2837652>.
- [56] P.-L. Curien and H. Herbelin. ‘The duality of computation’. In: *ACM SIGPLAN Notices* 35 (2000), pp. 233–243.
- [57] D. Delahaye and M. Mayero. ‘Dealing with algebraic expressions over a field in Coq using Maple’. In: *J. Symbolic Comput.* 39.5 (2005). Special issue on the integration of automated reasoning and computer algebra systems, pp. 569–592. DOI: [10.1016/j.jsc.2004.12.004](https://doi.org/10.1016/j.jsc.2004.12.004). URL: <http://dx.doi.org/10.1016/j.jsc.2004.12.004>.
- [58] É. Tanter and N. Tabareau. ‘Gradual Certified Programming in Coq’. In: *Proceedings of the 11th ACM Dynamic Languages Symposium (DLS 2015)*. Pittsburgh, PA, USA: ACM Press, Oct. 2015, pp. 26–40.



- [59] R. Douence, X. Lorca and N. Lorient. ‘Lazy Composition of Representations in Java’. In: *Software Composition, 8th International Conference, SC 2009, Zurich, Switzerland, July 2-3, 2009. Proceedings*. Ed. by A. Bergel and J. Fabry. Vol. 5634. Lecture Notes in Computer Science. Springer, 2009, pp. 55–71. DOI: [10.1007/978-3-642-02655-3\\_6](https://doi.org/10.1007/978-3-642-02655-3_6). URL: [https://doi.org/10.1007/978-3-642-02655-3\\_6](https://doi.org/10.1007/978-3-642-02655-3_6).
- [60] 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.
- [61] A. Frisch, G. Castagna and V. Benzaken. ‘Semantic Subtyping: Dealing Set-theoretically with Function, Union, Intersection, and Negation Types’. In: *J. ACM* 55.4 (Sept. 2008), 19:1–19:64.
- [62] C. Führtmann. ‘Direct Models for the Computational Lambda Calculus’. In: *Electr. Notes Theor. Comput. Sci.* 20 (1999), pp. 245–292.
- [63] J.-Y. Girard. ‘Linear Logic’. In: *Theoretical Computer Science* 50 (1987), pp. 1–102.
- [64] J.-Y. Girard, A. Scedrov and 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.
- [65] G. Gonthier. ‘Formal proofs—the four-colour theorem’. In: *Notices of the AMS* 55.11 (2008), pp. 1382–1393.
- [66] 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 and L. Théry. ‘A Machine-Checked Proof of the Odd Order Theorem’. In: *Interactive Theorem Proving*. Ed. by S. Blazy, C. Paulin-Mohring and D. Pichardie. Vol. 7998. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 163–179. DOI: [10.1007/978-3-642-39634-2\\_14](https://doi.org/10.1007/978-3-642-39634-2_14). URL: [http://dx.doi.org/10.1007/978-3-642-39634-2\\_14](http://dx.doi.org/10.1007/978-3-642-39634-2_14).
- [67] G. Gonthier, A. Mahboubi and E. Tassi. *A Small Scale Reflection Extension for the Coq system*. Research Report RR-6455. *The Reference Manual of the Ssreflect extension to the Coq tactic language, available at <http://hal.inria.fr/inria-00258384>*. INRIA, 2008.
- [68] 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.
- [69] Y.-G. Guéhéneuc, R. Douence and 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, p. 117. DOI: [10.1109/ASE.2002.1115000](https://doi.org/10.1109/ASE.2002.1115000). URL: <https://doi.org/10.1109/ASE.2002.1115000>.
- [70] 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 and R. Zumkeller. ‘A formal proof of the Kepler conjecture’. In: *CoRR* abs/1501.02155 (2015). URL: <http://arxiv.org/abs/1501.02155>.
- [71] H. A. Helfgott. ‘The ternary Goldbach conjecture is true’. In: *ArXiv e-prints* (Dec. 2013). arXiv: [1312.7748](https://arxiv.org/abs/1312.7748) [math.NT].
- [72] 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. URL: <https://hal.inria.fr/hal-00697240>.
- [73] H. Herbelin and É. Miquey. ‘Toward dependent choice: a classical sequent calculus with dependent types’. In: *TYPES 2015*. 2015.
- [74] T. Hirschowitz. ‘Cartesian closed 2-categories and permutation equivalence in higher-order rewriting’. In: *Logical Methods in Computer Science* 9.3 (2013). 19 pages, p. 10. DOI: [10.2168/LMCS-9\(3:10\)2013](https://doi.org/10.2168/LMCS-9(3:10)2013). URL: <https://hal.archives-ouvertes.fr/hal-00540205>.
- [75] 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*. Ed. by C. Baier and C. Tinelli. Vol. 9035. Lecture Notes in Computer Science. Springer, 2015, pp. 37–51.

- [76] G. Jaber, G. Lewertowski, P.-M. Pédrot, M. Sozeau and 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.
- [77] G. Jaber, N. Tabareau and M. Sozeau. ‘Extending type theory with forcing’. In: *Logic in Computer Science (LICS), 2012*. IEEE. 2012, pp. 395–404.
- [78] C. B. Jay and N. Ghani. ‘The Virtues of Eta-Expansion’. In: *J. Funct. Program.* 5.2 (1995), pp. 135–154.
- [79] U. Kohlenbach. *Applied proof theory: proof interpretations and their use in mathematics*. Springer Science & Business Media, 2008.
- [80] J.-L. Krivine. ‘Realizability algebras II : new models of ZF + DC’. In: *Logical Methods in Computer Science* 8.1 (2012).
- [81] J. Lambek and P. J. Scott. *Introduction to higher order categorical logic*. New York, NY, USA: Cambridge University Press, 1986.
- [82] S. M. Lane and I. Moerdijk. *Sheaves in Geometry and Logic*. Springer-Verlag, 1992.
- [83] R. Lepigre. ‘A classical realizability model for a semantical value restriction’. In: *European Symposium on Programming Languages and Systems*. Springer. 2016, pp. 476–502.
- [84] X. Leroy. ‘Formal certification of a compiler back-end or: programming a compiler with a proof assistant’. In: *ACM SIGPLAN Notices* 41.1 (2006), pp. 42–54.
- [85] P. B. Levy. *Call-By-Push-Value: A Functional/Imperative Synthesis*. Vol. 2. Semantic Structures in Computation. Springer, 2004.
- [86] P. B. Levy. ‘Contextual isomorphisms’. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM. 2017, pp. 400–414.
- [87] C. Liang and D. Miller. ‘Focusing and polarization in linear, intuitionistic, and classical logics’. In: *Theor. Comput. Sci.* 410.46 (2009), pp. 4747–4768.
- [88] Z. Luo, S. Soloviev and T. Xue. ‘Coercive subtyping: Theory and implementation’. In: *Inf. Comput.* 223 (2013), pp. 18–42.
- [89] J. Lurie. *Higher topos theory*. Annals of mathematics studies. Princeton, N.J., Oxford: Princeton University Press, 2009.
- [90] S. Mac Lane. ‘Natural associativity and commutativity’. In: *Selected Papers* (1979), pp. 415–433.
- [91] P. Martin-Löf. ‘An intuitionistic theory of types: predicative part’. In: *Logic Colloquium '73 Studies in Logic and the Foundations of Mathematics*. 80 (1975), pp. 73–118.
- [92] P.-A. Melliès. ‘Asynchronous Games 3 An Innocent Model of Linear Logic’. In: *Electr. Notes Theor. Comput. Sci.* 122 (2005), pp. 171–192.
- [93] P.-A. Melliès. ‘Asynchronous Games 4: A Fully Complete Model of Propositional Linear Logic’. In: *LICS*. 2005, pp. 386–395.
- [94] P.-A. Melliès and N. Tabareau. ‘Resource modalities in tensor logic’. In: *Ann. Pure Appl. Logic* 161.5 (2010), pp. 632–653.
- [95] É. Miquey. ‘A classical sequent calculus with dependent types’. In: *European Symposium on Programming*. Springer. 2017, pp. 777–803.
- [96] E. Moggi. ‘Computational lambda-calculus and monads’. In: *Proceedings of the Fourth Annual IEEE Symposium on Logic in Computer Science (LICS 1989)*. Pacific Grove, CA, USA: IEEE Computer Society Press, June 1989, pp. 14–23.
- [97] E. Moggi. ‘Notions of computation and monads’. In: *Inf. Comput.* 93.1 (July 1991), pp. 55–92. DOI: [10.1016/0890-5401\(91\)90052-4](https://doi.org/10.1016/0890-5401(91)90052-4). URL: [http://dx.doi.org/10.1016/0890-5401\(91\)90052-4](http://dx.doi.org/10.1016/0890-5401(91)90052-4).
- [98] 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.



- [99] G. Munch-Maccagnoni. ‘Models of a Non-Associative Composition’. In: *Proc. FoSSaCS*. Ed. by A. Muscholl. Vol. 8412. LNCS. Springer, 2014, pp. 397–412.
- [100] G. Munch-Maccagnoni. ‘Note on Curry’s style for Linear Call-by-Push-Value’. Manuscript. 3rd May 2017. URL: <https://hal.inria.fr/hal-01528857>.
- [101] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau and L. Birkedal. *Ynot: Reasoning with the awkward squad*. 2008.
- [102] C. Okasaki. *Purely functional data structures*. Cambridge University Press, 1999.
- [103] P.-M. Pédrot and N. Tabareau. ‘An Effectful Way to Eliminate Addiction to Dependence’. Jan. 2017. URL: <https://hal.inria.fr/hal-01441829>.
- [104] C. Prud’homme. ‘Contrôle de la propagation et de la recherche dans un solveur de contraintes. (Controlling propagation and search within a constraint solver)’. PhD thesis. École des mines de Nantes, France, 2014. URL: <https://tel.archives-ouvertes.fr/tel-01060921>.
- [105] C. Prud’homme, X. Lorca, R. Douence and N. Jussien. ‘Propagation engine prototyping with a domain specific language’. In: *Constraints* 19.1 (2014), pp. 57–76. DOI: [10.1007/s10601-013-9151-5](https://doi.org/10.1007/s10601-013-9151-5). URL: <https://doi.org/10.1007/s10601-013-9151-5>.
- [106] K. Quirin and N. Tabareau. ‘Lawvere-Tierney sheafification in Homotopy Type Theory’. In: *Journal of Formalized Reasoning* 9.2 (2016). DOI: [10.6092/issn.1972-5787/6232](https://doi.org/10.6092/issn.1972-5787/6232). URL: <https://hal.inria.fr/hal-01451710>.
- [107] F. van Raamsdonk. ‘Higher-order Rewriting’. In: *Proc. Rewrit. Tech. App.* Vol. 1631. LNCS. Springer, 1999, pp. 220–239.
- [108] J. C. Reynolds. ‘Types, Abstraction and Parametric Polymorphism’. In: *IFIP Congress*. 1983, pp. 513–523.
- [109] P. Selinger. ‘Control Categories and Duality: On the Categorical Semantics of the Lambda-Mu Calculus’. In: *Math. Struct in Comp. Sci.* 11.2 (2001), pp. 207–260.
- [110] S. G. Simpson. *Subsystems of Second Order Arithmetic*. Second. Cambridge Books Online. Cambridge University Press, 2009. URL: <http://dx.doi.org/10.1017/CB09780511581007>.
- [111] K. Støvring. ‘Extending the Extensional Lambda Calculus with Surjective Pairing is Conservative’. In: *Logical Methods in Computer Science* 2.2 (2006).
- [112] N. Swamy, C. Hrițcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoue and S. Zanella-Béguelin. ‘Dependent Types and Multi-Monadic Effects in F\*’. In: *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, Jan. 2016, pp. 256–270. URL: <https://www.fstar-lang.org/papers/mumon/>.
- [113] Univalent Foundations Project. *Homotopy Type Theory: Univalent Foundations for Mathematics*. <http://homotopytypetheory.org/book>, 2013.
- [114] M. Vákár. ‘A Framework for Dependent Types and Effects’. In: *arXiv preprint arXiv:1512.08009* (2015).
- [115] B. Ziliani, D. Dreyer, N. R. Krishnaswami, A. Nanevski and V. Vafeiadis. ‘Mtac: A monad for typed tactic programming in Coq’. In: *Journal of Functional Programming* 25 (2015). DOI: [10.1017/S0956796815000118](https://doi.org/10.1017/S0956796815000118). URL: <http://dx.doi.org/10.1017/S0956796815000118>.