RESEARCH CENTRE

Inria Centre at Rennes University

IN PARTNERSHIP WITH:

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

2024

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

Inria

Contents

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]
- Valentin Blot [INRIA, Researcher, from Sep 2024]
- Assia Mahboubi [INRIA, Senior Researcher, HDR]
- Kenji Maillard [INRIA, ISFP]
- Guillaume Munch [INRIA, Researcher]
- Pierre-Marie Pedrot [INRIA, Researcher]
- Matthieu Piquerez [INRIA, Starting Research Position]
- Matthieu Sozeau [INRIA, Researcher]

Faculty Members

- Julien Cohen [Nantes University, Associate Professor]
- Rémi Douence [IMT ATLANTIQUE, Associate Professor, HDR]
- Guilhem Jaber [Nantes University, Associate Professor]

Post-Doctoral Fellows

- Thiago Felicissimo Cesar [INRIA, Post-Doctoral Fellow, from Oct 2024]
- Koen Jacobs [INRIA, Post-Doctoral Fellow]
- Axel Kerinec [INRIA, Post-Doctoral Fellow]

PhD Students

- Sidney Congard [INRIA]
- Enzo Crance [INRIA, until Jan 2024]
- Hamza Jaafar [INRIA]
- Thomas Lamiaux [Nantes University, from Sep 2024]
- Yann Leray [Nantes University]
- Josselin Poiret [Nantes University]
- Vojtech Stepancik [INRIA, from Nov 2024]
- Tomas Javier Vallejos Parada [INRIA, from Mar 2024]

Technical Staff

- Martin Baillon [INRIA, Engineer]
- Kazuhiko Sakaguchi [INRIA, Engineer, until Oct 2024]

Interns and Apprentices

- Thomas Lamiaux [ENS PARIS-SACLAY, from Mar 2024 until Jul 2024]
- Gaia Loutchmia [INRIA, Intern, from Mar 2024 until Aug 2024]
- Stefan Ignacy Malewski Correa [UNIV CHILI, from Apr 2024 until Jul 2024]
- Virgil Marionneau [INRIA, Intern, from Sep 2024]
- Oliver Benjamin Turner [INRIA, Intern, from Jun 2024 until Sep 2024]

Administrative Assistant

- Anne-Claire Binetruy [INRIA]

Visiting Scientist

- Tomas Diaz Troncoso [UNIV CHILI, until Jul 2024]

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 full-fledged 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 in computer science. *Proof assistants* such as Coq [36] 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 effectiveness in the development of important pieces of certified software such as the C compiler of the CompCert project [41]. 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 unreliable from critical applications such as embedded systems.

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 [38, 40] or intricate combinations of a multitude of mathematical theories

[39]. But formalised mathematics is more than just proof checking and proof assistants can help with the organisation of mathematical knowledge or even with the discovery of new constructions and proofs.

Unfortunately, the rigidity of the theory behind proof assistants restricts 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 limitations 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 Curry-Howard correspondence is not perceived as a perfect match anymore, but rather as 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 [42] 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) [46] that 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 [43] and Girard [37] 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:

- 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.

- 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).

- 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

3.2.1 Multiverse and Sort Polymorphism

The experience of the team on various extensions of type theory (definitional proof irrelevant propositions, observational type theory, gradual type theory, opetopic type theory) begs naturally the question of the integration of these distinct flavours of type theory in a single type theory. At a theoretical level, we will investigate type theories with multiple universe hierarchies hosting theories with potentially incompatible principles able to express efficiently a variety of mathematical situations. At a practical level, we will develop a version of the Coq proof assistant with multiple sorts, generalizing the existing situation where the sorts of types, propositions and definitionally proof-irrelevant propositions cohabit de facto. An important challenge in that direction is to design a sound mechanism of sort polymorphism to factor away the constructions common to multiple sorts and prevent the combinatorial explosion induced by a naive implementation. A somewhat related line of research is designing an efficient decision procedure for universe level constraints, following the work of [35].

3.2.2 Extensional Equalities

In the long quest towards a practical and extensional notion of equality, observational type theory, first introduced by [34] and further developped and studied in [45] and [44], represents an important milestone that should now be implemented in practice. We will pursue in parallel other extensionality principles to enhance the expressivity of type theories.

3.2.3 Adding Effects in Type Theory

The investigation of extensions of CIC with side effects, in particular that of exceptions and the addition of a case analysis operator on types, yields important insights to give sound models of the cast calculi behind gradual dependent types. We plan to go beyond these relatively simple extensions and consider other widely used side effects, for instance the addition of global state to a type theory. These type theories with a primitive support for effectful operations could provide a new approach to the verification of programs exhibiting side-effects. Extending our previous work on classical sequent calculus with dependent types, we will study the integration of classical axioms such as excluded middle and choice in rich type theory. One goal is to better integrate insights of (classical) proof theory in the state of art of type theory (or in an alternative approach thereof). We also aim to look at concrete issues met in formalized mathematics stemming from the classical/intuitionist divide.

3.3 Tools for Improving Proof Assistants

3.3.1 MetaProgramming in Coq

The MetaCoq project currently provides bare-bones meta-programming facilities of quotation and denotation. We plan to improve this to provide a full-feature meta-programming facility, and explore the possibility to give strong specifications and verify our meta-programs. A prime example of this is the development of support for verified parametricity translations that can have many uses during formalization (elimination principles, automatic transport, etc.).

3.3.2 Automatic Transport of Libraries

We aim at pursuing the study of representation independence principles and the implementation of corresponding tools, so as to dramatically reduce the practical cost of library development. The mid-term

expected outcome concerns the design of refinements libraries, which connect proof-oriented with computation-oriented data-structures, and better transport instruments for formalized mathematics, *e.g.,* automating reasoning modulo structure isomorphisms.

3.3.3 Logical Frameworks for Proof Assistants

The porting of the developement of logical relations for MLTT of Abel *et al.* from Agda to Coq paves the may to a much more modular library. We would like to extend this work by developping a generic framework for dependently-typed logical relations, and use it for a wide variety of new dependent type theories. The main goal is to establish strong metatheoretical properties: normalization, but also suitable forms of interoperability. Ultimately, we believe this framework could interface with the MetaCoq project.

3.4 Formal Verification and Semantics of Real World Programming Languages

3.4.1 Semantic foundations of resource management in programming languages

We will keep investigating the semantic foundations of features of systems programming languages from a mathematical point of view. Based on our earlier work showing a link between resources and *ordered logic*, we will study resources management in the context of the formal theory of side-effects and linearity. Existing theorems will need to be generalized in many ways (extension of the notions of effect and resource modalities, handling of order, etc.). A link with linear logic will help make tighter connections between systems programming and "linear" approaches to program semantics (ownership, linear types, etc.). The notion of "borrowing" will be studied from the angle of linear logic, with possible applications to program verification. This study should also be extended to notions of fault tolerance (exception-safety and isolation) which might show links with modal logics. The anticipated outcome is an understanding of advanced notions in programming languages that better align with proven concepts from systems programming, compared to experimental type systems originating from pure theory, while providing clear distinctions between essential and accidental aspects of these real-world languages. As concrete experiments, we will keep researching ways to integrate systems programming concepts such as resources and fault tolerance in functional programming languages (notably OCaml and the OCaml-Rust interface).

3.4.2 Interactive semantics

We will continue our work on game semantics for programming languages, with the aim of studying interoperability and compilation between languages. Indeed, these semantics are particularly well suited to studying the interaction between a program and an environment written in different languages. We believe this approach will make it possible to overcome major open problems concerning interoperability between languages equipped with abstraction properties statically enforced by parametric polymorphism, and untyped languages where such abstractions properties are enforced dynamically. We will also continue studying the automation of reasoning on these semantics, along the lines of the CAVOC project. To do this, we want to apply abstract interpretation techniques, in particular the Abstracting Abstract Machine methodology, to automatically check accessibility properties on programs, such as unverified assertions.

As part of the CANofGAS project, we also plan to apply these interactive semantics to develop compositional cost models for programs. This would provide compositional reasoning on time and space complexity for higher-order programs.

3.5 Formal Verification of Computer Assisted Certification

3.5.1 Certification of the Trusted Code Base of Coq

The MetaCoq project's Achille's heel is that it relies on an assumption of strong normalization for the calculus: there is ongoing work in the team on defining powerful logical relations in Coq without relying on inductive-recursion, that gives hope that a strong-normalization model for a large fragment of MetaCoq can be constructed in the future. The main scientific obstacle is to specify the syntactic guard/productivity

condition at the heart of termination checking in such a way that it can be reduced to an eliminator-based definition of (co-)inductive types, which is how they are usually modelled. We anticipate difficulties with nested and indexed inductive types, which might be currently accepted by Coq but difficult to emulate with eliminators. However, this can only lead to a better understanding of the theory. As part of the ReCiProg project, we also plan to establish formal links between the validation criteria derived from circular proofs and this guard condition.

3.5.2 Formally Verified Symbolic Computations

The benefits of formally verified symbolic computations is twofold: increase the trust in computer-produced mathematics and expand the automation available for users of proof assistants. The main challenge is to enable the formal verification of efficient programs, whose correctness proofs involve sophisticated mathematical ingredients rather than subtle memory or parallelism issues. This involves in particular scaling up the automatic transport of libraries, as well as the formal verification of existing imperative code from computer algebra systems (typically written in C).

3.5.3 Erasure/Extraction of Certified Programs

The MetaCoq erasure pipeline, targeting C or OCaml, provides a guarantee that the evaluation of the compiled program gives a semantically correct result. However, in general extracted programs are linked to larger programs of the target language, where we lose guarantees of correctness in most non-trivial cases of interoperability. We are hence interested in developing techniques to show interoperability results between code that is extracted through our certified compilation pipeline and external code, *e.g.*, in OCaml or C. In [14], we developped a complete verified extraction pipeline from Coq to OCaml. The goal for the future is to scale this work to allow more scenarios of interoperability with effectful target programs, using a formal semantics for the target language. In particular, we should be able to soundly interpret the primitive constructs that are already part of Coq, which are fixed-width integers, IEE-754 floating point numbers and applicative arrays. There is a point of synergy here with the previous goal of enabling the development of efficient, formally verified symbolic computation.

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 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 Social and environmental responsibility

The team actively participates in open science initiatives, sharing tools and results to avoid redundant efforts, further reducing the collective computational and material costs associated with formal verification research. It spends efforts on the usability of these tools.

The team promotes a socially healthy governance for the tools it develops, taking into account the voices of all stakeholders, in accordance with the principles outlined in the ACM Code of Ethics. In particular, we have been actively involved in the renaming process of the Coq proof assistant, which is now called the Rocq Prover. This renaming initiative originated from a request by the Coq user community.

The research activities of the Gallinette team at Inria have a relatively low environmental footprint compared to other computationally intensive fields, as their work primarily involves theoretical development, formal proofs, and software tool design.

6 Highlights of the year

6.1 Awards

- Distinguished paper at CPP'24 [19]
- Distinguished paper at PLDI'24 [14]
- Two distinguished papers at LICS'24 [24, 27]

6.2 Organisation of international conferences

- We have created, organised and funded *Undone Computer Science*, the first conference on undone science in computer science, centred on ethics and epistemology of computer science, in collaboration with Nantes Université.

7 New software, platforms, open data

7.1 New software

7.1.1 Ltac2

Keywords: Coq, Proof assistant

Functional Description: Ltac2 is a member of the ML family of languages, in the sense that it is an effectful call-by-value functional language, with static typing à la Hindley-Milner. It is commonly accepted that ML constitutes a sweet spot in PL design, as it is relatively expressive while not being either too lax (unlike dynamic typing) nor too strict (unlike, say, dependent types).

The main goal of Ltac2 is to serve as a meta-language for Coq. As such, it naturally fits in the ML lineage, just as the historical ML was designed as the tactic language for the LCF prover. It can also be seen as a general-purpose language, by simply forgetting about the Coq-specific features.

Sticking to a standard ML type system can be considered somewhat weak for a meta-language designed to manipulate Coq terms. In particular, there is no way to statically guarantee that a Coq term resulting from an Ltac2 computation will be well-typed. This is actually a design choice, motivated by backward compatibility with Ltac1. Instead, well-typedness is deferred to dynamic checks, allowing many primitive functions to fail whenever they are provided with an ill-typed term.

The language is naturally effectful as it manipulates the global state of the proof engine. This allows to think of proof-modifying primitives as effects in a straightforward way. Semantically, proof manipulation lives in a monad, which allows to ensure that Ltac2 satisfies the same equations as a generic ML with unspecified effects would do, e.g. function reduction is substitution by a value.

Contact: Pierre-Marie Pedrot

7.1.2 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 auxilliary 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 8.18 to 8.20, 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 minor update of Equations, now compatible with Coq 8.20. The main changes are fixes in the funelim tactic and simplification engine to avoid trying to simplify unrelated hypotheses (those under "block" markers). which sometimes led to slowing down the tactic. Also includes performance improvements by @ppedrot.

See https://github.com/mattam82/Coq-Equations/releases/tag/v1.3.1-8.20 for details

URL: http://mattam82.github.io/Coq-Equations/

Publications: hal-01671777, hal-01248807, inria-00628862

Contact: Matthieu Sozeau

Participant: Matthieu Sozeau

7.1.3 Math-Components

Name: Mathematical Components library

Keywords: Proof assistant, Coq, Formalisation

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

URL: https://math-comp.github.io/

Contact: Assia Mahboubi

Participants: Alexey Solovyev, Andrea Asperti, Assia Mahboubi, Cyril Cohen, Enrico Tassi, Francois Garillot, Georges Gonthier, Ioana Pasca, Jeremy Avigad, Laurence Rideau, Laurent Théry, Russell O'Connor, Sidi Ould Biha, Stéphane Le Roux, Yves Bertot

7.1.4 Math-comp-analysis

Name: Mathematical Components Analysis

Keywords: Proof assistant, Coq, Formalisation

Functional Description: This library adds definitions and theorems to the Math-components library for real numbers and their mathematical structures.

Release Contributions: First major release, several results in topology, integration, and measure theory have been added.

URL: https://github.com/math-comp/analysis

Publications: hal-02463336, hal-03917948, hal-01719918

Contact: Cyril Cohen

Participants: Cyril Cohen, Georges Gonthier, Marie Kerjean, Assia Mahboubi, Damien Rouhling, Pierre Roux, Laurence Rideau, Pierre-Yves Strub, Reynald Affeldt, Laurent Théry, Yves Bertot, Zachary Stone

Partners: Ecole Polytechnique, AIST Tsukuba, Onera

7.1.5 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 compoment 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: Release 1.3.2 of the MetaCoq project for Coq 8.19 - 8.20 is available both as source and through opam. See the website for a detailed overview of the project, introductory material and related articles and presentations.

The main changes in this new version w.r.t. v1.2.1 are:

A full integration of the typed erasure phase from the ConCert project in the erasure pipeline, with a complete correctness proof, by @mattam82. Use option MetaCoq Erase -typed to switch it on. It can be configured with the "live" erasure function inside Coq (see erasure_live_test.v) Generalizations of the correctness and simulation lemmas by @yforster @mattam82 and @tabareau, showing in particular that erasure of applications of functions from firstorder types to firstorder types is compiled to applications, justifying separate compilation of functions and their arguments. Using standardization and canonicity, we also show that erased values of programs of firstorder inductive types (non-erasable inductives types for which all constructor argument types are themselves firstorder) are in direct correspondence with their Coq counterparts, allowing sound readback of these values. In other words, evaluating the erased terms under these assumptions faithfully simulates evaluation in Coq. Based on this, CertiCoq and coq-malfunction both implement an Eval variant that reads back Coq values and can be trusted. Support for primitive ints, floats and array literal values. Primitive operations are still treated as axioms to be realized in target languages and the correctness theorems do not apply in their presence yet. Optional passes have been added that replicate the Coq Extraction plugin's functionality, without proof (yet): Inlining of defined constants (e.g. Extract Inline). Reordering of constructors (e.g. part of Extract Inductive). This allows to target different representations in target languages (typically bool in OCaml). Unboxing of singleton unary constructors. For example, exist nat (fun x : nat => x = 1) 1 p : { x : nat | x = 1 } becomes exist 1 after typed erasure and removal of constructor parameters, which can be further unboxed to just 1. CoFixpoints/CoInductives to Lazy/Inductives: cofixpoints and (co)-constructors get translated to fixpoints + lazy/force constructs in lambda-box, allowing efficient evaluation of coinductive terms in target languages (supported only in coq-malfunction/ocaml extraction for now). Beta-reduction. This reduces manifest beta-redexes in the erased terms, especially useful

after inlining. The preprint "Verified Extraction from Coq to OCaml" presents the development of the compilation pipeline from Coq to Malfunction/OCaml, including the correctness proofs mentioned above.

The preprint "Correct and Complete Type Checking and Certified Erasure for Coq, in Coq" presents the development of the sound and complete type checker based on bidirectional typing, the meta-theoretical results (subject reduction, standardization, canonicity and consistency) and the verified erasure procedure of this version of MetaCoq.

MetaCoq integrates Template-Coq, a reification and denotation plugin for Coq terms and global declarations, a Template monad for metaprogramming (including the ability to extract these metaprograms to OCaml for efficiency), a formalisation of Coq's calculus PCUIC in Coq, a relatively efficient, sound and complete type checker for PCUIC, a verified type and proof erasure procedure from PCUIC to untyped lambda calculus and a quotation library. MetaCoq provides a low-level interface to develop certified plugins like translations, compilers or tactics in Coq itself.

You can install MetaCoq directly from sources or using opam install coq-metacoq. This release will be included in an upcoming Coq Platform.

The current release includes several subpackages, which can be compiled and installed separately if desired:

the utils library contains extensions to the standard library (notably for reasoning with All/All-n type-valued predicates) (in directory utils, and as coq-metacoq-utils). the common libraries of basic definitions for the abstract syntax trees shared by multiple languages (common, coq-metacoq-common) the Template-Coq quoting library and plugin (template-coq / coq-metacoq-template) a formalisation of meta-theoretical properties of PCUIC, the calculus underlying Coq (pcuic / coq-metacoq-pcuic) a verified equivalence between Template-Coq and PCUIC typing (in directory template-pcuic and as coq-metacoq-template-pcuic) a total verified type-checker for Coq (safechecker / coq-metacoq-safechecker), usable inside Coq. a plugin interfacing with the extracted type-checker in OCaml, providing the MetaCoq SafeCheck <term> command (safechecker-plugin, coq-metacoq-safechecker-plugin) a verified type and proof erasure function for Coq (erasure / coq-metacoq-erasure), usable inside Coq. a plugin interfacing with the extracted erasure pipeline in OCaml, providing the MetaCoq Erase <term> command (erasure-plugin, coq-metacoq-erasure-plugin) a quoting library, allowing the quotation of terms and type derivations along with associated data structures as ASTs/terms (quotation / coq-metacoq-quotation). a set of example translations from Type Theory to Type Theory (translation/ coq-metacoq-translations). A good place to start are the files demo.v, safechecker_test.v, erasure_test.v in the test-suite directory.

This version of MetaCoq was developed by Yannick Forster, Jason Gross, Yann Leray, Matthieu Sozeau and Nicolas Tabareau with contributions from Yishuai Li. You are welcome to contribute by opening issues and PRs. A MetaCoq Zulip stream is also available.

The MetaCoq Team

News of the Year: This year's work was mostly concentrated on improving the erasure/extraction pipeline, integrating and unifying parts of the CertiCoq and coq-malfunction projects to allow producing code that is ammenable to compilation by a functional programming language compiler. The new pipeline supports the proof of new interoperability theorems, improving the formal guarantees provided by MetaCoq's extraction.

URL: https://metacoq.github.io

Publications: hal-04077552, hal-04329663, hal-03516619, hal-02901011, hal-02380196, hal-02167423, hal-01809681

Contact: Matthieu Sozeau

Participants: Abhishek Anand, Danil Annenkov, Meven Lennon-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, Saarland University

7.1.6 Coq

Name: The Coq Proof Assistant

Keyword: Proof assistant

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 integrated development environment (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: An overview of the new features and changes, along with the full list of contributors is available at https://coq.inria.fr/refman/changes.html#version-8-20 .

News of the Year: Coq version 8.20 adds a new rewrite rule mechanism along with a few new features, a host of improvements to the virtual machine, the notation system, Ltac2 and the standard library.

URL: http://coq.inria.fr/

Contact: Matthieu Sozeau

Participants: Yves Bertot, Frédéric Besson, Tej Chajed, Cyril Cohen, Pierre Corbineau, Pierre Courtieu, Maxime Dénès, Jim Fehrle, Julien Forest, Emilio Jesús Gallego Arias, Gaëtan Gilbert, Georges Gonthier, Benjamin Grégoire, Jason Gross, Hugo Herbelin, Vincent Laporte, Olivier Laurent, Assia Mahboubi, Kenji Maillard, Erik 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, Theo Zimmermann

7.1.7 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: Documentation improvements

URL: https://gitlab.com/gadmm/memprof-limits

Publication: hal-03517592

Contact: Guillaume Munch

7.1.8 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

Publication: hal-03910313

Contact: Guillaume Munch

Participants: Guillaume Munch, Gabriel Scherer

7.1.9 LogRel-Coq

Keyword: Proof assistant

Functional Description: This Coq library develop the metatheory of Martin-Löf Type Theory with a universe and some inductive types in order to establish consistency, normalisation, canonicity and decidability of a core theory close to that of Coq.

URL: https://github.com/CoqHott/logrel-coq

Publications: hal-04379245, hal-04214008, hal-04160858

Contact: Kenji Maillard

Participants: Meven Lennon-Bertrand, Loic Pujet, Pierre-Marie Pedrot, Kenji Maillard, Yannick Forster, Arthur Adjedj

7.1.10 Trocq

Keywords: Proof synthesis, Proof transfer, Coq, Elpi, Logic programming, Parametricity, Univalence

Functional Description: Trocq is a prototype of a modular parametricity plugin for Coq, aiming to perform proof transfer by translating the goal into an associated goal featuring the target data structures as well as a rich parametricity witness from which a function justifying the goal substitution can be extracted.

The plugin features a hierarchy of parametricity witness types, ranging from structure-less relations to a new formulation of type equivalence, gathering several pre-existing parametricity translations, including univalent parametricity and CoqEAL, in the same framework.

This modular translation performs a fine-grained analysis and generates witnesses that are rich enough to preprocess the goal yet are not always a full-blown type equivalence, allowing to perform proof transfer with the power of univalent parametricity, but trying not to pull in the univalence axiom in cases where it is not required.

The translation is implemented in Coq-Elpi and features transparent and readable code with respect to a sequent-style theoretical presentation.

Release Contributions: Support for the Prop sort

URL: https://github.com/coq-community/trocq

Publication: hal-04177913

Contact: Cyril Cohen

Participants: Cyril Cohen, Enzo Crance, Assia Mahboubi

Partner: Mitsubishi Electric R&D Centre Europe, France

8 New results

8.1 Type Theory

Participants: Martin Baillon, Gaëtan Gilbert, Yann Leray, Assia Mahboubi, Kenji Maillard, Josselin Poiret, Matthieu Piquerez, Pierre-Marie Pédrot, Matthieu Sozeau, Nicolas Tabareau.

All Your Base Are Belong to Us: Sort Polymorphism for Proof Assistants Proof assistants based on dependent type theory, such as Coq, Lean and Agda, use different universes to classify types, typically combining a predicative hierarchy of universes for computationally-relevant types, and an impredicative universe of proof-irrelevant propositions. In general, a universe is characterized by its sort, such as Type or Prop, and its level, in the case of a predicative sort. Recent research has also highlighted the potential of introducing more sorts in the type theory of the proof assistant as a structuring means to address the coexistence of different logical or computational principles, such as univalence, exceptions, or definitional proof irrelevance. This diversity raises concrete and subtle issues from both theoretical and practical perspectives. In particular, in order to avoid duplicating definitions to inhabit all (combinations of) universes, some sort of polymorphism is needed. Universe level polymorphism is well-known and effective to deal with hierarchies, but the handling of polymorphism between sorts is currently ad hoc and limited in all major proof assistants, hampering reuse and extensibility. In [16], we develop sort polymorphism and its metatheory, studying in particular monomorphization, large elimination, and parametricity. Sort polymorphism is a natural solution that effectively addresses the limitations of current approaches and prepares the ground for future multi-sorted type theories.

Observational Equality Meets CIC Equality is at the heart of dependent type theory, as it plays a fundamental role in specifications and mathematical reasoning. The standard way to handle it in mainstream proof assistants such as Agda , Lean or Coq is based on Martin-Löf's identity type, which comes straight out of the '70s—its elegance and simplicity have earned it a long-standing use, despite a major discrepancy with traditional mathematical formulations: it does not satisfy any extensionality principles. Recently, the work on observational equality has regained interest as a new way to encode equality in proof assistants that support a universe of definitionally proof-irrelevant propositions; however it has yet to be integrated in any major proof assistant, because it is not fully compatible with another

important feature of type theory: indexed inductive types. In [28], we propose a systematic integration of indexed inductive types with an observational equality, and show that this integration can only be completely satisfactory if the observational equality satisfies the computational rule of Martin-Löf's identity type. The second contribution of this paper is a formal proof that this additional computation rule, although not present in previous works on observational equality, can be integrated to the system without compromising the decidability of conversion.

Engineering logical relations for MLTT in Coq We report in [19] on a mechanization in the Coq proof assistant of the decidability of conversion and type-checking for Martin-Löf Type Theory (MLTT), extending a previous Agda formalization. Our development proves the decidability not only of conversion, but also of type-checking, using bidirectional derivations that are canonical for typing. Moreover, we wish to narrow the gap between the object theory we formalize (currently MLTT with Π, Σ, N and one universe) and the metatheory used to prove the normalization result, e.g., MLTT, to a mere difference of universe levels. We thus avoid induction-recursion or impredicativity, which are central in previous work. Working in Coq, we also investigate how its features, including universe polymorphism and the metaprogramming facilities provided by tactics, impact the development of the formalization compared to the development style in Agda. The development is freely accessible on GitHub.

"Upon This Quote I Will Build My Church Thesis" The internal Church thesis (CT) is a logical principle stating that one can associate to any function $f : \mathbb{N} \to \mathbb{N}$ a concrete code, in some Turing-complete language, that computes f. While the compatibility of CT in simpler systems has been long known, its compatibility with dependent type theory is still an open question. In [27], we answer this question positively. We define "MLTT", a type theory extending MLTT with quote operators in which CT is derivable. We furthermore prove that "MLTT" is consistent, strongly normalizing and enjoys canonicity using a rather standard logical relation model. All the results in this paper have been mechanized in Coq.

Gradual Indexed Inductive Types Indexed inductive types are essential in dependently-typed programming languages, enabling precise and expressive specifications of data structures and properties. Recognizing that programming and proving with dependent types could benefit from the smooth integration of static and dynamic checking that gradual typing offers, recent efforts have studied gradual dependent types. Gradualizing indexed inductive types however remains mostly unexplored: the standard encodings of indexed inductive types in intensional type theory, e.g., using type-level fixpoints or subset types, break in the presence of gradual features; and previous work on gradual dependent types focus on very specific instances of indexed inductive types. In [15], we contribute a general framework, named PUNK, specifically designed for exploring the design space of gradual indexed inductive types. PUNK is a versatile framework, enabling the exploration of the space between eager and lazy cast reduction semantics that arise from the interaction between casts and the inductive eliminator, allowing them to coexist and interoperate in a single system. Our work provides significant insights into the intersection of dependent types and gradual typing, by proposing a criteria for well-behaved gradual indexed inductive types, systematically addressing the outlined challenges of integrating these types. The contributions of this paper are a step forward in the quest for making gradual theorem proving and gradual dependently-typed programming a reality.

A Zoo of Continuity Properties in Constructive Type Theory Continuity principles stating that all functions are continuous play a central role in some schools of constructive mathematics. However, there are different ways to formalise the property of being continuous in constructive foundations. We analyse these continuity properties from the perspective of constructive reverse mathematics. We work in constructive type theory, which can be seen as a minimal foundation for constructive reverse mathematics. We treat continuity of functions $F : (Q \to A) \to R$, i.e. with question type Q, answer type A, and result type R. Concretely, we discuss continuity defined via moduli, making the relevant list $L : LQ$ of questions explicit, dialogue trees, making the question answer process explicit as inductive tree, and tree functions, making the question answer process explicit as function. We prove equivalences where possible and isolate necessary and sufficient axioms for equivalence proofs. Many of the results we discuss are already present in the works of Hancock, Pattinson, Ghani, Kawai, Fujiwara, Brede, Herbelin, Escardó, and others.

Our main contribution is their formulation over a uniform foundation, the observation that no choice axioms are necessary, the generalisation to arbitrary types from natural numbers where possible, and a mechanisation in the Coq proof assistant.

8.2 Proof Assistants

Participants: Gaëtan Gilbert, Yann Leray, Assia Mahboubi, Kenji Maillard, Pierre-Marie Pédrot, Matthieu Sozeau, Nicolas Tabareau.

The Rewster: The Coq Proof Assistant with Rewrite Rules Dependently typed languages such as Coq or Agda are very convenient tools to program with strong invariants and develop mathematical proofs. However, a user might be inconvenienced by things such as the fact that n and n+0 are not considered definitionally equal, or the inability to postulate one's own constructs with computation rules such as exceptions. Coq modulo theory solves the first of the two problems by extending Coq's conversion with decision procedures, e.g., for linear integer arithmetic. Rewrite rules can be used to deal with directed equalities for natural numbers, but also to implement exceptions that compute. They were introduced in Agda a few years ago, and later extended to provide more guarantees with a modular confluence checker. We present in [25] a work-in-progress extension of Coq which supports user-defined rewrite rules. While we mostly follow in the footsteps of the Agda implementation, we also have to face new issues due to the differences in the implementation and meta-theory of Coq and Agda. The most prominent one being the different treatment of universes as Coq supports cumulativity but no first-class universe levels.

Trocq: Proof Transfer for Free, With or Without Univalence In interactive theorem proving, a range of different representations may be available for a single mathematical concept, and some proofs may rely on several representations. Without automated support such as proof transfer, theorems available with different representations cannot be combined, without light to major manual input from the user. Tools with such a purpose exist, but in proof assistants based on dependent type theory, it still requires human effort to prove transfer, whereas it is obvious and often left implicit on paper. In [21, 20], we present Trocq, a new proof transfer framework, based on a generalization of the univalent parametricity translation, thanks to a new formulation of type equivalence. This translation takes care to avoid dependency on the axiom of univalence for transfers in a delimited class of statements, and may be used with relations that are not necessarily isomorphisms. We motivate and apply our framework on a set of examples designed to show that it unifies several existing proof transfer tools. The article also discusses an implementation of this translation for the Coq proof assistant, in the Coq-Elpi metalanguage.

Correct and Complete Type Checking and Certified Erasure for Coq, in Coq Coq is built around a well-delimited kernel that performs type checking for definitions in a variant of the Calculus of Inductive Constructions (CIC). Although the metatheory of CIC is very stable and reliable, the correctness of its implementation in Coq is less clear. Indeed, implementing an efficient type checker for CIC is a rather complex task, and many parts of the code rely on implicit invariants which can easily be broken by further evolution of the code. Therefore, on average, one critical bug has been found every year in Coq. In [17, 18], we present the first implementation of a type checker for the kernel of Coq (without the module system, template polymorphism and η-conversion), which is proven sound and complete in Coq with respect to its formal specification. Note that because of Gödel's second incompleteness theorem, there is no hope to prove completely the soundness of the specification of Coq inside Coq (in particular strong normalization), but it is possible to prove the correctness and completeness of the implementation assuming soundness of the specification, thus moving from a trusted code base (TCB) to a trusted theory base (TTB) paradigm. Our work is based on the MetaCoq project which provides meta-programming facilities to work with terms and declarations at the level of the kernel. We verify a relatively efficient type checker based on the specification of the typing relation of the Polymorphic, Cumulative Calculus of Inductive Constructions (PCUIC) at the basis of Coq. It is worth mentioning that during the verification process, we have found a source of incompleteness in Coq's official type checker, which has then been fixed in Coq 8.14 thanks to our work. In addition to the kernel implementation, another essential feature

of Coq is the so-called extraction mechanism: the production of executable code in functional languages from Coq definitions. We present a verified version of this subtle type and proof erasure step, therefore enabling the verified extraction of a safe type checker for Coq in the future.

8.3 Logical Foundations of Programming Languages

Participants: Sidney Congard, Hamza Jaafar, Guillhem Jaber, Guillaume Munch-Maccagnoni.

Semantic foundations of resource management in programming languages We continued previous work establishing a formal link between resource-management features from systems programming (C++/Rust), and ordered (or non-commutative) logic.

- We previously showed that there exist algorithms for clean-up functions that are efficient in time and space for ordered algebraic datatypes. In [29], we extend this technique by making it more modular, in order to handle abstract types, separate compilation, and unboxed types, in order to make it suitable for code generation, as part of Jean Caspar's internship. This addresses a long-standing conceptual problem with compiler-generated clean-up functions causing stack overflows in deep structures.
- In [30], we present a functional translation of a subset of safe Rust programs, building upon the results of Aeneas. It preserves linearity and captures a new feature, namely lifetime bounds. This is a work in progress: in particular, translation rules are not set yet.

Designing interrupts for ML and OCaml This is an ongoing work to propose a sound design for asynchronous exceptions (or interrupts, in ML parlance) in the OCaml language, inspired by fault-tolerance concepts and Rust's design for exceptions, in collaboration with researchers from Jane Street. In [32], we first describe the problems of recovering from interrupts in Standard ML and OCaml, and how these language deficiencies have been worked around, notably in the implementation of proof assistants and other interactive tools. We then aim to explain more conceptually how safe interrupt handling is possible by evolving ML exceptions. We emphasize key notions: exception-safety concepts originating from C++ and Rust's extension of Erlang's "let it fail" approach to shared mutable state. Lastly we will discuss considerations specific to the OCaml language. This work suggests, in the spirit of structured programming, that while the ML notion of exception lacks structure, it refines into better-structured abstractions for error handling, suggested by its relevant usage by programmers.

Führmann-Hasegawa theorem A result by Fürhmann and by Hasegawa is important in the type-theoretic semantics of side-effects and linearity, as it characterises what it means to be without side-effects in the context of classical and linear logics. It was admitted to be true, but lacked a written proof, for lack of a conceptual approach to the result. We proposed such a conceptual proof with Éléonore Mangel.

8.4 Program Certifications and Formalisation of Mathematics

Participants: Yannick Forster, Assia Mahboubi, Kenji Maillard, Matthieu Piquerez, Kazuhiko Sakaguchi, Matthieu Sozeau, Nicolas Tabareau.

A First Order Theory of Diagram Chasing In [26], we discuss the formalization of proofs "by diagram chasing", a standard technique for proving properties in abelian categories. We discuss how the essence of diagram chases can be captured by a simple many-sorted first-order theory, and we study the models and decidability of this theory. The longer-term motivation of this work is the design of a computer-aided instrument for writing reliable proofs in homological algebra, based on interactive theorem provers.

Machine-Checked Categorical Diagrammatic Reasoning In [23], we describe a formal proof library, developed using the Coq proof assistant, designed to assist users in writing correct diagrammatic proofs, for 1-categories. This library proposes a deep-embedded, domain-specific formal language, which features dedicated proof commands to automate the synthesis, and the verification, of the technical parts often eluded in the literature.

Geometric theories for real number algebra without sign test or dependent choice axiom In [31], we have constructed a dynamical theory that is as complete as possible to describe the algebraic properties of the real number field in constructive mathematics without a dependent choice axiom.

Verified Extraction from Coq to OCaml One of the central claims of fame of the Coq proof assistant is extraction, i.e., the ability to obtain efficient programs in industrial programming languages such as OCaml, Haskell, or Scheme from programs written in Coq's expressive dependent type theory. Extraction is of great practical usefulness, used crucially e.g., in the CompCert project. However, for such executables obtained by extraction, the extraction process is part of the trusted code base (TCB), as are Coq's kernel and the compiler used to compile the extracted code. The extraction process contains intricate semantic transformation of programs that rely on subtle operational features of both the source and target language. Its code has also evolved since the last theoretical exposition in the seminal PhD thesis of Pierre Letouzey. Furthermore, while the exact correctness statements for the execution of extracted code are described clearly in academic literature, the interoperability with unverified code has never been investigated formally, and yet is used in virtually every project relying on extraction. In [14], we describe the development of a novel extraction pipeline from Coq to OCaml, implemented and verified in Coq itself, with a clear correctness theorem and guarantees for safe interoperability. We build our work on the MetaCoq project, which aims at decreasing the TCB of Coq's kernel by reimplementing it in Coq itself and proving it correct w.r.t. a formal specification of Coq's type theory in Coq. Since OCaml does not have a formal specification, we make use of the Malfunction project specifying the semantics of the intermediate language of the OCaml compiler. Our work fills some gaps in the literature and highlights important differences between the operational semantics of Coq programs and their extracted variants. In particular, we focus on the guarantees that can be provided for interoperability with unverified code, identify guarantees that are infeasible to provide, and raise interesting open question regarding semantic guarantees that could be provided. As central result, we prove that extracted programs of first-order data type are correct and can safely interoperate, whereas for higher-order programs already simple interoperations can lead to incorrect behaviour and even outright segfaults.

Composing Biases by Using CP to Decompose Minimal Functional Dependencies for Acquiring Complex Formulae Given a table with a minimal set of input columns that functionally determines an output column, we introduce in [22] a method that tries to gradually decompose the corresponding minimal functional dependency (mfd) to acquire a formula expressing the output column in terms of the input columns. A first key element of the method is to create sub-problems that are easier to solve than the original formula acquisition problem, either because it learns formulae with fewer inputs parameters, or as it focuses on formulae of a particular class, such as Boolean formulae; as a result, the acquired formulae can mix different learning biases such as polynomials, conditionals or Boolean expressions. A second key feature of the method is that it can be applied recursively to find formulae that combine polynomial, conditional or Boolean subterms in a nested manner. The method was tested on data for eight families of combinatorial objects; new conjectures were found that were previously unattainable. The method often creates conjectures that combine several formulae into one with a limited number of automatically found Boolean terms.

9 Bilateral contracts and grants with industry

9.1 Bilateral Contracts with Industry

Contract Extension. The bilateral contracts listed below ended in 2023, but the associated overhead continued for an additional 12 months.

CoqExtra

Participants: Pierre-Marie Pédrot, Matthieu Sozeau, Nicolas Tabareau, Yannick Forster, Pierre Giraud, Kazuhiko Sakaguchi.

Title: A Formally Verified Extraction Mechanism using Precise Type Specifications

Duration: 2020 - 2023 (extended to 2024)

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-checked 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)

Participants: Assia Mahboubi, Enzo Crance.

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

Duration: 2020 - 2023 (extended to 2024)

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, that 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

Participants: Guillaume Munch-Maccagnoni.

Title: OCaml/Rust bindings

Duration: 2021-2023 (extended to 2024)

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

Participants: Guilhem Jaber, Hamza Jaafar.

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).

10 Partnerships and cooperations

10.1 International initiatives

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

GRAPA

Participants: Kenji Maillard, Nicolas Tabareau.

Title: Gradual Proof Assistants

Duration: 2023 - 2025

Coordinator: Nicolas Tabareau

Partners: Centrum Wiskunde & Informatica, Universidad de Chile (Chile)

Inria contact: Nicolas Tabareau

Summary: The main objective of this work is therefore to extend the reach of gradual typing to full-fledged type theories in order to support smooth certified programming in a new generation of proof assistants.

10.2 European initiatives

10.2.1 H2020 projects

FRESCO

Participants: Martin Baillon, Matthieu Piquerez, Vojtech Stepancik, Assia Mahboubi, Tomas Javier Vallejos Parada.

FRESCO project on cordis.europa.eu

Title: Fast and Reliable Symbolic Computation

Duration: From November 1, 2021 to October 31, 2026

Partners:

- INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET AUTOMATIQUE (INRIA), France

Inria contact: Assia Mahboubi

Coordinator:

Summary: The use of computers for formulating conjectures, but also for substantiating proof steps, pervades mathematics, even in its most abstract fields. Most computer proofs are produced by symbolic computations, using computer algebra systems. Sadly, these systems suffer from severe, intrinsic flaws, key to their amazing efficiency, but preventing any flavor of post-hoc verification.

But can computer algebra become reliable while remaining fast? Bringing a positive answer to this question represents an outstanding scientific challenge per se, which this project aims at solving.

Our starting point is that interactive theorem provers are the best tools for representing mathematics in silico. But we intend to disrupt their architecture, shaped by decades of applications in computer science, so as to dramatically enrich their programming features, while remaining compatible with their logical foundations.

We will then design a novel generation of mathematical software, based on the firm grounds of modern programming language theory. This environment will feature a new, high-level, performance-oriented programming language, devised for writing efficient and correct code easily, and for serving the frontline of research in computational mathematics. Users will have access to fast implementations, and to powerful proving technologies for verifying any component à la carte, with high productivity. Logic- and computer-based formal proofs will prevent run-time errors, and incorrect mathematical semantics.

We will maintain a close, continuous collaboration with interested high-profile mathematicians, on the verification of cutting-edge research results, today beyond the reach of formal proofs. We ambition to empower mathematical journals to install high-quality artifact evaluation, when peer-reviewing falls short of assessing computer proofs. This project will eventually impact the use of formal methods in engineering, in areas like cryptography or signal-processing.

10.3 National initiatives

NUSCAP

Participants: Enzo Crance, Assia Mahboubi.

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. Hovewer, 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

Participants: Guilhem Jaber.

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.

CANofGAS

Participants: Guilhem Jaber.

Title: Cost Analysis of Game Semantics

Program: Inria Exploratory Action,

Duration: Sep 2022 - Dec 2025

Coordinator: Beniamino Accattoli (CR Inria, LIX, PARTOUT Team) and Guilhem Jaber (MCF, LS2N, Gallinette Team)

Local Contact: Guilhem Jaber

Summary: CANofGAS aims at capturing the time and space cost of the evaluation of higher-order programs at the semantic level. The directions we plan to explore are using the advances in reasonable cost models to develop a cost-based understanding of game semantics. In particular, we aim at modelling the efficient call-by-need evaluation scheme, at work for instance in the Haskell language and in the Coq proof assistant.

11 Dissemination

Participants: Valentin Blot, Julien Cohen, Rémi Douence, Guilhem Jaber, Assia Mahboubi, Kenji Maillard, Guillaume Munch-Maccagnoni, Pierre-Marie Pédrot, Matthieu Sozeau, Nicolas Tabareau.

11.1 Promoting scientific activities

11.1.1 Scientific events: organisation

General chair, scientific chair

- Guillaume Munch-Maccagnoni was organising and PC co-chair of the Undone Computer Science conference 2024, held in the LS2N, Nantes from Feb. 5-7. Computer scientists were invited to submit 2-page abstract on the theme of "undone science" in computer science, in order to trigger discussion on ethics and epistemology of the field. The program committee selected 28 talk proposals, from a wide range of CS subdomains. The conference attracted around 70 participants from France, UK, Germany, Switzerland, USA, and Ireland, including around 30% of women. It is followed by a call for full papers in a special issue of *Philosophia Scientiæ* to be published in diamond open access in 2026 by the Laboratoire d'Histoire des Sciences et de Philosophie – Archives Henri-Poincaré.

Member of the organizing committees

- Assia Mahboubi has served in the scientific committee of the conference Lean for the Curious Mathematician 2024 (Cirm, Marseille). Assia Mahboubi has organized a workshop of the Fresco project in Nantes.

Steering commitee

- Nicolas Tabareau is a member of the steering committee of the ACM Certified Programs and Proofs (CPP) conference.

- Assia Mahboubi is a member of the steering committee of the Interactive Theorem Proving (ITP) conference.
- Valentin Blot is a member of the steering committee of the Logic In Computer Science (LICS) conference.

11.1.2 Scientific events: selection

Chair of conference program committees

- Guilhem Jaber was PC co-chair of the HOPE'24 workshop of ICFP.
- Guillaume Munch-Maccagnoni was PC co-chair of the Undone Computer Science conference 2024 (LS2N, Nantes).
- Nicolas Tabareau is serving as co-chair of the ACM Certified Programs and Proofs (CPP) conference for 2025 and 2026.

Member of the conference program committees

- Assia Mahboubi has served on the program committees of POPL'24, Fossacs 2025, FSCD 2025 international conferences.
- Pierre-Marie Pédrot has served in the program committees of POPL'24.
- Matthieu Sozeau has served on the program committees of the WITS 2024 workshop and the ICFP'24 conference.
- Kenji Maillard has served in the program committee of the OOPSLA'24 and ICFP'24 conferences.
- Guilhem Jaber has served in the program committees of POPL'24 international conference, and on the FICS'24 and GALOP'24 workshops.
- Guillaume Munch-Maccagnoni has served in the program committees of the Onwards! Papers 2024 conference (Pasadena, California) and ML workshop 2024 (Milan, Italy).

Reviewer

- Guilhem Jaber was a reviewer for LICS'24, FoSSaCS'25 and POPL'25
- Kenji Maillard was a reviewer for POPL'24 and ESOP'24.

11.1.3 Journal

Member of the editorial boards

- Assia Mahboubi serves on the editorial boards of the Journal of Automated Reasoning, Logical Methods in Computer Science and Annals of Formalized Mathematics.

Reviewer - reviewing activities

- Guilhem Jaber was a reviewer for LMCS.
- Kenji Maillard was a reviewer for TOPLAS.
- Guillaume Munch-Maccagnoni serves as co-guest editor for the special issue of Philosophia Scientiæ on undone science in computer science.
- Matthieu Sozeau was a reviewer for the Journal of Automated Reasoning.

11.1.4 Invited talks

- Matthieu Sozeau has given an invited lecture at the JFLA'24 and hosted the traditional Coq Development team Session at CoqPL'24.

11.1.5 Leadership within the scientific community

Assia Mahboubi has served in the scientific committee of the GdR Informatique Mathématique.

11.1.6 Research administration

Assia Mahboubi is an elected member of the Commission d'Évaluation Inria and member of the conseil de laboratoire of the Laboratoire des Sciences du Numérique (LS2N).

11.2 Teaching - Supervision - Juries

11.2.1 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 : Guilhem Jaber, Logic for Computer Science, 60h, L3, Nantes Université France
- Licence : Guilhem Jaber, Foundations of Computer Science, 54h, L3, Nantes Université France
- Licence : Guilhem Jaber, Functional Programming, 24h, L3, Nantes Université France
- Master : Guilhem Jaber, Verification and Formal Proofs, 12h, M1, Nantes Université, France
- Master : Guilhem Jaber, Modelisation and Verification of Concurrent Systems, 12h, M2, Nantes Université, France

11.2.2 Supervision

- PhD in progess: Sidney Congard, Towards a linear functional translation for borrowing, IMT-A, advisors: Rémi Douence and Guillaume Munch-Maccagnoni.
- Enzo Crance has defended his PhD on December 2023, Méta-programmation pour le transfert de preuve en théorie des types dépendants , Nantes Université, advisors: Denis Cousineau and Assia Mahboubi.
- PhD in progress: Thomas Lamiaux, Certifying the guard condition of Rocq, Nantes Université, advisors: Yannick Forster, Matthieu Sozeau and Nicolas Tabareau.
- PhD in progress: Josselin Poiret, A Multiverse Type Theory, Nantes Université, advisors: Kenji Maillard and Nicolas Tabareau.
- PhD in progress: Yann Leray, Putting SProp at work, Nantes Université, advisors: Matthieu Sozeau and Nicolas Tabareau.
- PhD in progress: Peio Borthelle, Operational Game Semantics in Type Theory, advisor: Tom Hirschowitz, Guilhem Jaber, Yannick Zakowski
- PhD in progress: Hamza Jaafar, Sémantique des jeux opérationnelle pour les effets algébriques génératifs, Nantes Université, advisors: Guilhem Jaber and Nicolas Tabareau.
- PhD in progress: Tomas Vallejos Parada, Fast and reliable arithmetic for interactive theorem proving, Nantes Université, advisor: Assia Mahboubi.
- PhD in progress: Vojtěch Štěpančík, Algebraic structures in dependent types theory, Nantes Univrsité, advisor: Cyril Cohen and Assia Mahboubi.
- PhD in progress: Éléonore Mangel, Constructive classical logic and dependent types with effects, Université Paris Cité, advisors: Paul-André Melliès and Guillaume Munch-Maccagnoni.

11.2.3 Juries

- Nicolas Tabareau has served in the PhD jury of Louise Dubois de Prisque, Université Paris-Saclay.
- Assia Mahboubi has served in the PhD jury of Clément Blaudeau (Université Paris Cité) and Max Zeuner (Stockholm University) and has served as reviewer for the PhD thesis of Kiran Gopinathan (National University of Singapore).
- Matthieu Sozeau has served in the PhD jury of Nathanaëlle Courant (Université Paris Cité) and was a reader of Joomy Korkut's PhD (Princeton University).

11.3 Popularization

11.3.1 Productions (articles, videos, podcasts, serious games, ...)

Assia Mahboubi has contributed a chapter to the book "Le calcul à découvert" (Cnrs éditions).

11.3.2 Education

Assia Mahboubi co-coordinates a joint computer science and mathematics departments program for Art+Science actions in schools at Nantes Université.

12 Scientific production

12.1 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] 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. URL: https://hal.archives-ouvertes.fr/hal-03510684.

[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. URL: https://hal.archives-ouvertes.fr/hal-02901011.

[4] E. Finster, A. Allioux and M. Sozeau. 'Types are internal infinity-groupoids'. In: LICS 2021. Rome, Italy, 21st June 2021. URL: https://hal.inria.fr/hal-03133144.

[5] Y. Forster, M. Sozeau and N. Tabareau. 'Verified Extraction from Coq to OCaml'. In: *Proceedings of the ACM on Programming Languages* 8.PLDI (20th June 2024), pp. 52–75. DOI: 10.1145/3656379. URL: https://inria.hal.science/hal-04329663.

[6] 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. URL: https://hal.archives-ouvertes.fr/hal-02388621.

[7] 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. URL: https://hal.inria.fr/hal-02548315.

[8] 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. URL: https://hal.archives-ouvertes.fr/hal-02383109.

[9] L. Pujet and N. Tabareau. 'Impredicative Observational Equality'. In: *POPL 2023 Proceedings of the 50th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL 2023 - 50th ACM SIGPLAN Symposium on Principles of Programming Languages. Vol. 7. Proceedings of the ACM on programming languages. Boston, United States, 15th Jan. 2023, p. 74. DOI: 10.1145/3571739. URL: https://hal.archives-ouvertes.fr/hal-03857705.

[10] 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.

[11] 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. URL: https://hal.archives-ouvertes.fr/hal-02380196.

[12] M. Sozeau, Y. Forster, M. Lennon-Bertrand, J. B. Nielsen, N. Tabareau and T. Winterhalter. 'Correct and Complete Type Checking and Certified Erasure for Coq, in Coq'. In: *Journal of the ACM (JACM)* (27th Nov. 2024), pp. 1–76. DOI: \url{https://doi.org/10.1145/3706056}. URL: https://inria.hal.science/hal-04077552.

[13] 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. URL: https://hal.inria.fr/hal-03120580.

12.2 Publications of the year

International journals

[14] Y. Forster, M. Sozeau and N. Tabareau. 'Verified Extraction from Coq to OCaml'. In: *Proceedings of the ACM on Programming Languages* 8.PLDI (20th June 2024), pp. 52–75. DOI: 10.1145/3656379. URL: https://inria.hal.science/hal-04329663 (cit. on pp. 7, 8, 19).

[15] M. Malewski, K. Maillard, N. Tabareau and É. Tanter. 'Gradual Indexed Inductive Types'. In: *Proceedings of the ACM on Programming Languages* 8.ICFP (15th Aug. 2024), pp. 544–572. DOI: 10.1145/3674644. URL: https://inria.hal.science/hal-04681546 (cit. on p. 16).

[16] J. Poiret, G. Gilbert, K. Maillard, P.-M. Pédrot, M. Sozeau, N. Tabareau and É. Tanter. 'All Your Base Are Belong to Us: Sort Polymorphism for Proof Assistants'. In: *Proceedings of the ACM on Programming Languages* 9.POPL (Jan. 2025). DOI: 10.1145/3704912. URL: https://nantes-universite.hal.science/hal-04801739 (cit. on p. 15).

[17] M. Sozeau, Y. Forster, M. Lennon-Bertrand, J. B. Nielsen, N. Tabareau and T. Winterhalter. 'Correct and Complete Type Checking and Certified Erasure for Coq, in Coq'. In: *Journal of the ACM (JACM)* (27th Nov. 2024), pp. 1–76. DOI: \url{https://doi.org/10.1145/3706056}. URL: https://inria.hal.science/hal-04077552 (cit. on p. 17).

Invited conferences

[18] M. Sozeau. 'MetaCoq : de la métaprogrammation à l'extraction certifiée pour Coq'. In: 35es Journées Francophones des Langages Applicatifs (JFLA 2024). Saint-Jacut-de-la-Mer, France, 2024. URL: https://inria.hal.science/hal-04407164 (cit. on p. 17).

International peer-reviewed conferences

[19] A. Adjedj, M. Lennon-Bertrand, K. Maillard, P.-M. Pédrot and L. Pujet. 'Martin-Löf à la Coq'. In: CPP '24: 13th ACM SIGPLAN International Conference on Certified Programs and Proofs. London UK, United Kingdom: ACM; ACM, 15th Jan. 2024, pp. 230–245. DOI: 10.1145/3636501.3636951. URL: https://hal.science/hal-04214008 (cit. on pp. 8, 16).

[20] C. Cohen, E. Crance and A. Mahboubi. 'Artifact Report: Trocq: Proof Transfer for Free, With or Without Univalence'. In: *Lecture Notes in Computer Science*. ESOP 2024 - 33rd European Symposium on Programming. Vol. LNCS-14576. Programming Languages and Systems 33rd European Symposium on Programming, ESOP 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6–11, 2024, Proceedings, Part I. Luxembourg, Luxembourg: Springer Nature Switzerland, 5th Apr. 2024, pp. 269–274. DOI: 10.1007/978-3-031-57262-3_11. URL: https://inria.hal.science/hal-04623207 (cit. on p. 17).

[21] C. Cohen, E. Crance and A. Mahboubi. 'Trocq: Proof Transfer for Free, With or Without Univalence'. In: *Lecture Notes in Computer Science*. ESOP 2024 - 33rd European Symposium on Programming. 14576. Luxembourg, Luxembourg: Springer, 2024, pp. 239–268. URL: https://hal.science/hal-04177913 (cit. on p. 17).

[22] R. Gindullin, N. Beldiceanu, J. Cheukam-Ngouonou, R. Douence and C.-G. Quimper. 'Composing Biases by Using CP to Decompose Minimal Functional Dependencies for Acquiring Complex Formulae'. In: AAAI 2024 - 38th Annual AAAI Conference on Artificial Intelligence. Vancouver, Canada, 2024. URL: https://hal.science/hal-04460576 (cit. on p. 19).

[23] B. Guillemet, A. Mahboubi and M. Piquerez. 'Machine-Checked Categorical Diagrammatic Reasoning'. In: *9th International Conference on Formal Structures for Computation and Deduction (FSCD 2024)*. 9th International Conference on Formal Structures for Computation and Deduction (FSCD 2024). Vol. 299. Leibniz International Proceedings in Informatics (LIPIcs). Tallinn, Estonia, 5th July 2024, p. 692. URL: https://inria.hal.science/hal-04471683 (cit. on p. 19).

[24] M. Kerjean and P.-M. Pédrot. '∂ is for Dialectica'. In: *39th Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS 2024 - 39th Annual ACM/IEEE Symposium on Logic in Computer Science. Tallinn, Estonia: IEEE, 2024, pp. 1–13. URL: https://inria.hal.science/hal-04583978 (cit. on p. 8).

[25] Y. Leray, G. Gilbert, N. Tabareau and T. Winterhalter. 'The Rewster: Type Preserving Rewrite Rules for the Coq Proof Assistant'. In: International Conference on Interactive Theorem Proving (ITP 2024). Vol. 15th International Conference on Interactive Theorem Proving (ITP 2024). Tbilisi, Georgia, 2nd Sept. 2024, p. 18. DOI: 10.4230/LIPIcs.ITP.2024.26. URL: https://inria.hal.science/hal-04511667 (cit. on p. 17).

[26] A. Mahboubi and M. Piquerez. 'A First Order Theory of Diagram Chasing'. In: *32nd EACSL Annual Conference on Computer Science Logic 2024 (CSL'24)*. CSL 2024 - 32nd EACSL Annual Conference on Computer Science Logic. Naples, Italy, 2024, pp. 1–19. URL: https://hal.science/hal-04266479 (cit. on p. 18).

[27] P.-M. Pédrot. '"Upon This Quote I Will Build My Church Thesis"'. In: *39th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '24)*. LICS 2024 - 39th Annual ACM/IEEE Symposium on Logic in Computer Science. Tallin, Estonia, 2024, pp. 1–12. DOI: 10.1145/3661814.3662070. URL: https://inria.hal.science/hal-04571149 (cit. on pp. 8, 16).

[28] L. Pujet and N. Tabareau. 'Observational Equality Meets CIC'. In: *Lecture Notes in Computer Science*. ESOP 2024 - 33rd European Symposium on Programming. Vol. 14576. Lecture Notes in Computer Science. Luxembourg, Luxembourg: Springer Nature Switzerland, 5th Apr. 2024, pp. 275–301. DOI: 10.1007/978-3-031-57262-3_12. URL: https://hal.science/hal-04535982 (cit. on p. 16).

National peer-reviewed Conferences

[29] J. Caspar and G. Munch-Maccagnoni. 'Modular efficient deconstruction with typed pointer reversal'. In: 35es Journées Francophones des Langages Applicatifs (JFLA 2024). Saint-Jacut-de-la-Mer, France, 2024. URL: https://inria.hal.science/hal-04406342 (cit. on p. 18).

[30] S. Congard. 'Towards a linear functional translation for borrowing'. In: 35es Journées Francophones des Langages Applicatifs (JFLA 2024). Saint-Jacut-de-la-Mer, France, 2024. URL: https://hal.science/hal-04360462 (cit. on p. 18).

Conferences without proceedings

[31] H. Lombardi and A. Mahboubi. 'Geometric theories for real number algebra without sign test or dependent choice axiom'. In: CCC 2024 - Continuity, Computability, Constructivity From Logic to Algorithms. Nice, France, 2024, pp. 1–152. URL: https://inria.hal.science/hal-04709177 (cit. on p. 19).

[32] G. Munch-Maccagnoni, L. White and S. Dolan. 'Designing interrupts for ML and OCaml'. In: ML 2024 - ACM SIGPLAN Workshop on ML Family Workshop Higher-order, Typed, Inferred, Strict. Milan, Italy: ACM, 2024. DOI: 10.1007/11818502_15. URL: https://inria.hal.science/hal-04860321 (cit. on p. 18).

Reports & preprints

[33] M. Baillon, Y. Forster, A. Mahboubi, P.-M. Pédrot and M. Piquerez. *A Zoo of Continuity Properties in Constructive Type Theory*. 23rd Jan. 2025. URL: https://inria.hal.science/hal-04908282.

12.3 Cited publications

[34] 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 (cit. on p. 5).

[35] M. Bezem and T. Coquand. 'Loop-checking and the uniform word problem for join-semilattices with an inflationary endomorphism'. In: *Theor. Comput. Sci.* 913 (2022), pp. 1–7. DOI: 10.1016/J.TCS.2022.01.017. URL: https://doi.org/10.1016/j.tcs.2022.01.017 (cit. on p. 5).

[36] Coq Development Team, The. *The Coq proof assistant reference manual*. Version 8.5. 2015. URL: http://coq.inria.fr (cit. on p. 3).

[37] J.-Y. Girard. 'Linear Logic'. In: *Theoretical Computer Science* 50 (1987), pp. 1–102 (cit. on p. 4).

[38] G. Gonthier. 'Formal proofs—the four-colour theorem'. In: *Notices of the AMS* 55.11 (2008), pp. 1382–1393 (cit. on p. 3).

[39] 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. URL: http://dx.doi.org/10.1007/978-3-642-39634-2_14 (cit. on p. 4).

[40] 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 (cit. on p. 3).

[41] 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 (cit. on p. 3).

[42] 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 (cit. on p. 4).

[43] 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 (cit. on p. 4).

[44] L. Pujet and N. Tabareau. 'Impredicative Observational Equality'. In: *Proceedings of the ACM on Programming Languages*. Proceedings of the ACM on programming languages 7.POPL (Jan. 2023), p. 74. DOI: 10.1145/3571739. URL: https://hal.science/hal-03857705 (cit. on p. 5).

[45] L. Pujet and N. Tabareau. 'Observational Equality: Now For Good'. In: *Proceedings of the ACM on Programming Languages* 6.POPL (Jan. 2022), pp. 1–29. DOI: 10.1145/3498693. URL: https://inria.hal.science/hal-03367052 (cit. on p. 5).

[46] Univalent Foundations Project. *Homotopy Type Theory: Univalent Foundations for Mathematics*. http://homotopytypetheory.org/book, 2013 (cit. on p. 4).