RESEARCH CENTRE

Inria Centre at Rennes University

IN PARTNERSHIP WITH:

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

2023

ACTIVITY REPORT

Project-Team GALLINETTE

Gallinette: developing a new generation of proof assistants

IN COLLABORATION WITH: Laboratoire des Sciences du numerique 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]
- Assia Mahboubi [INRIA, Senior Researcher, HDR]
- Kenji Maillard [INRIA, ISFP]
- Guillaume Munch-Maccagnoni [INRIA, Researcher]
- Pierre-Marie Pédrot [INRIA, Researcher]
- Matthieu Sozeau [INRIA, Researcher]

Faculty Members

- Julien Cohen [UNIV NANTES, Associate Professor]
- Rémi Douence [IMT ATLANTIQUE, Associate Professor, HDR]
- Guilhem Jaber [UNIV NANTES, Associate Professor Delegation, until Aug 2023]

Post-Doctoral Fellows

- Yannick Forster [INRIA, Post-Doctoral Fellow, until Nov 2023]
- Koen Jacobs [INRIA, Post-Doctoral Fellow]
- Axel Kerinec [INRIA, Post-Doctoral Fellow, from Sep 2023]
- Matthieu Piquerez [INRIA, Post-Doctoral Fellow]

PhD Students

- Martin Baillon [INRIA]
- Sidney Congard [INRIA, from Oct 2023]
- Enzo Crance [INRIA, from Nov 2023]
- Enzo Crance [MITSUBISHI ELECTRIC, until Oct 2023]
- Pierre Giraud [INRIA, until Nov 2023]
- Hamza Jaafar [INRIA]
- Yann Leray [UNIV NANTES, from Sep 2023]
- Josselin Poiret [UNIV NANTES, from Sep 2023]

Technical Staff

- Sidney Congard [INRIA, Engineer, from May 2023 until Sep 2023]
- Nils Lauermann [INRIA, Engineer, until Aug 2023]
- Loic Pujet [INRIA, Engineer, until May 2023]
- Kazuhiko Sakaguchi [INRIA, Engineer]

Interns and Apprentices

- Jean Caspar [ENS Paris, Intern, from Jun 2023 until Jul 2023]
- Benoît Guillemet [ENS PARIS-SACLAY, from May 2023 until Jul 2023]
- Robin Jourde [ENS DE LYON, Intern, from Feb 2023 until Jul 2023]
- Virgil Marionneau [UNIV NANTES, from Apr 2023 until Jun 2023]

Administrative Assistant

- Anne-Claire Binetruy [INRIA]

Visiting Scientists

- Tomàs Diaz Troncoso [UNIV CHILI, from Oct 2023]
- Stefan Ignacy Malewski Correa [UNIV CHILI, from Oct 2023]
- Léo Mangel [LS2N, from Mar 2023 until May 2023]
- Eric Tanter [UNIV CHILI, from Jun 2023 until Jul 2023]

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 [39] 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 [45]. 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 [41, 43] or intricate combinations of a multitude of mathematical theories [42]. 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 [46] 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) [50] 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 [47] and Girard [40] put forward monads and co-monads as describing various compositional notions of computation. In this theory, programs can have side-effects (state, exceptions, input-output), logics can be non-intuitionistic (linear, classical), and different computational universes can interact (modal logics). Recently, the safe and automatic management of resources has also seen a coming of age (Rust, Modern C++) confirming the importance of linear logic for various programming concepts. It is now understood that the characteristic feature of the theory of effects is sensitivity to *evaluation order*, in contrast with type theory which is built around the assumption that evaluation order is irrelevant.

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

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

1. Making the notion of equality closer to what is usually assumed when doing proofs on black board, with a balance between irrelevant equality for simple structures and equality up-to equivalences for more complex ones (Section 3.2). Such a notion of equality should allow one to implement traditional model transformations that enhance the logical power of the proof assistant using distinct compilation phases.

2. Advancing the foundations of effects within the Curry-Howard approach. The objective is to pave the way for the integration of effects in proof assistants and to prototype the corresponding

implementation. This integration should allow for not only certified programming with effects, but also the expression of more powerful logics (Section 3.3).

3. Making more programming features (notably, object polymorphism) available in proof assistants, in order to scale to practical-sized developments. The objective is to enable programming styles closer to common practices. One of the key challenges here is to leverage gradual typing to dependent programming (Section 3.4).

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

3.2 Enhance the computational and logical power of proof assistants

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 [38].

3.2.2 Extensional Equalities

In the long quest towards a practical and extensional notion of equality, observational type theory, first introduced by [36] and further developped and studied in [49] and [13], 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. In particular, functor laws for type formers [33] provide a common basis to type cast operations with a structural behaviour.

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 [32], 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 Highlights of the year

- Distinguished paper at CPP'24 for [29].
- Assia Mahboubi did her inaugural lecture for professorship at Vrije Universiteit Amsterdam on April 2023.
- The team has organized a week of events the week before Xmas (Decembre 18-21), with a Coq developer meeting, two PhD defenses and a one-day worskhop.

6 New software, platforms, open data

6.1 New software

6.1.1 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

6.1.2 Equations

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

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

The main features of Equations include:

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

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

Support for mutual and nested structural recursion using with and where 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 8.13 to 8.17, with special support for the Coq-HoTT library), that allows the definition of functions by dependent pattern-matching and well-founded, mutual or nested structural recursion and compiles them into core terms. It automatically derives the clauses equations, the graph of the function and its associated elimination principle.

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

Release Contributions: This is a new major release of Equations, working with Coq 8.15 to 8.17. This version adds an improved syntax (less ,-separation), integration with the Coq-HoTT library and numerous bug fixes. See the reference manual for details.

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

Enhancements of pattern interpretation

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

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

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

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

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

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

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

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

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

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

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

Contact: Matthieu Sozeau

Participant: Matthieu Sozeau

6.1.3 Math-Components

Name: Mathematical Components library

Keyword: Proof assistant

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

Release Contributions: Major release using Hierarchy Builder to handle algebraic structures.

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

Contact: Assia Mahboubi

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

6.1.4 Math-comp-analysis

Name: Mathematical Components Analysis

Keyword: Proof assistant

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

Release Contributions: Several results in integration 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

6.1.5 MetaCoq

Keyword: Coq

Scientific Description: The MetaCoq project aims to provide a certified meta-programming environment in Coq. It builds on Template-Coq, a plugin for Coq originally implemented by Malecha (Extensible proof engineering in intensional type theory, Harvard University, 2014), which provided a reifier for Coq terms and global declarations, as represented in the Coq kernel, as well as a denotation command. Recently, it was used in the CertiCoq certified compiler project (Anand et al., in: CoqPL, Paris, France, 2017), as its front-end language, to derive parametricity properties (Anand and Morrisett, in: CoqPL'18, Los Angeles, CA, USA, 2018). However, the syntax lacked semantics, be it typing semantics or operational semantics, which should reflect, as formal specifications in Coq, the semantics of Coq 's type theory itself. The tool was also rather bare bones, providing only rudimentary quoting and unquoting commands. MetaCoq generalizes it to handle the entire polymorphic calculus of cumulative inductive constructions, as implemented by Coq, including the kernel's declaration structures for definitions and inductives, and implement a monad for general manipulation of Coq's logical environment. The MetaCoq framework allows Coq users to define many kinds of general purpose plugins, whose correctness can be readily proved in the system itself, and that can be run efficiently after extraction. Examples of implemented plugins include a parametricity translation and a certified extraction to call-by-value lambda-calculus. The meta-theory of Coq itself is verified in MetaCoq along with verified conversion, type-checking and erasure procedures providing highly trustable alternatives to the procedures in Coq's OCaml kernel. MetaCoq is hence a foundation for the development of higher-level certified tools on top of Coq's kernel. A meta-programming and proving framework for Coq.

MetaCoq is made of 4 main components:

- The entry point of the project is the Template-Coq quoting and unquoting library for Coq which allows quotation and denotation of terms between three variants of the Coq AST: the OCaml one used by Coq's kernel, the Coq one defined in MetaCoq and the one defined by the extraction of the MetaCoq AST, allowing to extract OCaml plugins from Coq implementations.

- The PCUIC component is a full formalization of Coq's typing and reduction rules, along with proofs of important metatheoretic properties: weakening, substitution, validity, subject reduction and principality. The PCUIC calculus differs slightly from the Template-Coq one and verified translations between the two are provided.

- The checker component contains verified implementations of weak-head reduction, conversion and type inference for the PCUIC calculus, along with a verified checker for Coq theories.

- The erasure 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: This new version integrates:

Support for primitive integers and floating point values, using the same typechecking mechanism as Coq's kernel, up to the erased lambda-box language. Better computational behavior of the safe checker. Support for nix and cachix (useful for CI, allows to reuse remotely compiled components) Registering of projections for inductive types defined as records More efficient eta-expansion transformation using environment maps instead of association lists.

URL: https://metacoq.github.io

Publications: 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

6.1.6 Coq

Name: The Coq Proof Assistant

Keywords: Proof, Certification, Formalisation

Scientific Description: Coq is an interactive proof assistant based on the Calculus of (Co-)Inductive Constructions, extended with universe polymorphism. This type theory features inductive and co-inductive families, an impredicative sort and a hierarchy of predicative universes, making it a very expressive logic. The calculus allows to formalize both general mathematics and computer programs, ranging from theories of finite structures to abstract algebra and categories to programming language metatheory and compiler verification. Coq is organised as a (relatively small) kernel including efficient conversion tests on which are built a set of higher-level layers: a powerful proof engine and unification algorithm, various tactics/decision procedures, a transactional document model and, at the very top an 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-8-18 .

News of the Year: Coq version 8.18 integrates changes to several parts of the system : kernel, specification language, type inference, notation, tactics, Ltac2 language, commands and options, command-line tools, CoqIDE, standard library, infrastructure and dependencies, extraction. See https://coq.inria.fr/refman/changes.html#version-8-18 for an overview of the new features and changes, along with the full list of contributors.

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

Contact: Matthieu Sozeau

Participants: Yves Bertot, 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, Érik Martin-Dorel, Guillaume Melquiond, Pierre-Marie Pedrot, Clément Pit-Claudel, Kazuhiko Sakaguchi, Vincent Semeria, Michael Soegtrop, Arnaud Spiwack, Matthieu Sozeau, Enrico Tassi, Laurent Théry, Anton Trunov, Li-Yao Xia, Theo Zimmermann

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

6.1.7 memprof-limits

Keyword: Library

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

Memprof-limits interrupts the execution by raising an asynchronous exception: an exception that can arise at almost any location in the program. It is provided with a guide on how to recover

from asynchronous exceptions and other unexpected exceptions, summarising for the first time practical knowledge acquired in OCaml by the Coq proof assistant as well as in other programming languages.

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

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

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

Release Contributions: Initial version.

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

Publication: hal-03517592

Author: Guillaume Munch

Contact: Guillaume Munch

6.1.8 ocaml-boxroot

Keywords: Interoperability, Library, Ocaml, Rust

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

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

URL: https://gitlab.com/ocaml-rust/ocaml-boxroot

Publication: hal-03910313

Contact: Guillaume Munch

Participants: Guillaume Munch, Gabriel Scherer

6.1.9 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

6.1.10 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.

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

7 New results

7.1 Type Theory

Participants: Antoine Allioux, Martin Baillon, Gaëtan Gilbert, Meven Lennon-Bertand, Assia Mahboubi, Kenji Maillard, Pierre-Marie Pédrot, Loïc Pujet, Matthieu Sozeau, Nicolas Tabareau.

Impredicative Observational Equality In dependent type theory, impredicativity is a powerful logical principle that allows the definition of propositions that quantify over arbitrarily large types, potentially resulting in self-referential propositions. Impredicativity can provide a system with increased logical strength and flexibility, but in counterpart it comes with multiple incompatibility results. In particular, Abel and Coquand showed that adding definitional uniqueness of identity proofs (UIP) to the main proof assistants that support impredicative propositions (Coq and Lean) breaks the normalization procedure, and thus the type-checking algorithm. However, it was not known whether this stems from a fundamental incompatibility between UIP and impredicativity or if a more suitable algorithm could decide type-checking for a type theory that supports both. In [13], we design a theory that handles both UIP and impredicativity by extending the recently introduced observational type theory TTobs with an impredicative universe of definitionally proof-irrelevant types, as initially proposed in the seminal work on observational equality of [36]. We prove decidability of conversion for the resulting system, that we call CCobs, by harnessing proof-irrelevance to avoid computing with impredicative proof terms. Additionally, we prove normalization for CCobs in plain Martin-Löf type theory, thereby showing that adding proof-irrelevant impredicativity does not increase the computational content of the theory.

Definitional Functoriality for Dependent (Sub)Types Dependently-typed proof assistant rely crucially on definitional equality, which relates types and terms that are automatically identified in the underlying type theory. In [33], we extend type theory with definitional functor laws, equations satisfied propositionally by a large class of container-like type constructors F : Type → Type, equipped with a mapF : $(A \to B) \to F\ A \to F\ B$, such as lists or trees. Promoting these equations to definitional ones strengthen the theory, enabling slicker proofs and more automation for functorial type constructors. This extension is used to modularly justify a structural form of coercive subtyping, propagating subtyping through type formers in a map-like fashion. We show that the resulting notion of coercive subtyping, thanks to the extra definitional equations, is equivalent to a natural and implicit form of subsumptive subtyping. The key result of decidability of type-checking in a dependent type system with functor laws for lists has been entirely mechanized in Coq.

Engineering logical relations for MLTT in Coq We report in [15] 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.

Opetopic Type Theory In his PhD thesis, defended this year, Antoine Allioux developed the theory of dependent opetopic types and the universe of polynomial monads and provides example synthetic proofs of results from higher category theory, including adjunction and representability theorems.

Pursuing Shtuck By mimicking the internal sheafification construction in an arbitrary topos, we provide a simple description of the various sheaf-related constructions in a purely type-theoretical setting [34]. Assuming mild extensionality properties on our target type theory, we show that it almost results in a syntactic model of CIC. Unfortunately, a well-known topos-theoretic issue carries over, and the resulting theory, called ShTT, does not feature universes. We propose three different solutions to this problem: making the target theory univalent, allowing effects in the source theory, or changing our point of view while using strict propositions. As a side-product, we also give a computational interpretation of sheaves that highlights their deep relationship with the well-known structure of interaction trees.

7.2 Proof Assistants

Participants: Yannick Forster, Gaëtan Gilbert, Kazuhiko Sakaguchi, Yann Leray, Assia Mahboubi, Kenji Maillard, Pierre-Marie Pédrot, Loïc Pujet, Matthieu Sozeau, Nicolas Tabareau.

Manifest Termination In formal systems combining dependent types and inductive types, such as the Coq proof assistant, non-terminating programs are frowned upon. They can indeed be made to return impossible results, thus endangering the consistency of the system, although the transient usage of a non-terminating Y combinator, typically for searching witnesses, is safe. To avoid this issue, the definition of a recursive function is allowed only if one of its arguments is of an inductive type and any recursive call is performed on a syntactically smaller argument. If there is no such argument, the user has to artificially add one, e.g., an accessibility property. Free monads can still be used to address general recursion and elegant methods make possible to extract partial functions from sophisticated recursive schemes. The latter yet rely on an inductive characterization of the domain of a function, and of its computational graph, which in turn might require a substantial effort of specification and proof. This leads to a rather frustrating situation when computations are involved. Indeed, the user first has to formally prove that

the function will terminate, then the computation can be performed, and finally a result is obtained (assuming the user waited long enough). But since the computation did terminate, what was the point of proving that it would terminate? In [22], we investigates how users of proof assistants based on variants of the Calculus of Inductive Constructions could benefit from manifestly terminating computations. A companion file showcasing the approach in the Coq proof assistant is available online.

Compositional pre-processing for automated reasoning in dependent type theory In the context of interactive theorem provers based on a dependent type theory, automation tactics (dedicated decision procedures, call of automated solvers, ...) are often limited to goals which are exactly in some expected logical fragment. This very often prevents users from applying these tactics in other contexts, even similar ones. [16] discusses the design and the implementation of pre-processing operations for automating formal proofs in the Coq proof assistant. It presents the implementation of a wide variety of predictible, atomic goal transformations, which can be composed in various ways to target different backends. A gallery of examples illustrates how it helps to expand significantly the power of automation engines.

From Lost to the River: Embracing Sort Proliferation Since their inception, proof assistants based on dependent type theory have featured some way to quantify over types. Leveraging dependent products, the most common way to do so is to introduce a type of types, known as a universe. Care has to be taken, as paradoxes lurk in the dark. Martin-Löf famously introduced in his seminal type theory MLTT a universe U with the typing rule $U : U$, only for Girard to show that this system was inconsistent. The standard solution is to introduce a hierarchy of universes $(U_i)_{i\in\mathbb{N}}$ and mandate that $U_i : U_{i+1}$.

While trivial from the point of view of the typing rules, this additional index is a major source of non-modularity. In [26] we report on the implementation of sort polymorphism in Coq to solve this modularity issue.

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 [20] 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.

Porting Coq Scripts to the Mathematical Components Library Version 2 The Mathematical Components library (hereafter, MathComp) provides, among others, a number of mathematical structures organized as hierarchies. Hierarchy Builder (hereafter, HB) is an extension of the Coq proof assistant to ease the development of hierarchies of structures. MathComp 2 is the result of the port of MathComp to HB. [30] is a technical report whose goal is to explain how to port MathComp developments to MathComp 2. It has been written by the participants of the MathComp Documentation Sprint that happened from 2023-05-03 to 2023-05-10.

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 [17], 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 [35, 14], 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.

7.3 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. We have extended this approach to support abstract types and separate compilation, 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 [25], 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.

Resource Polymorphism Thanks to the semantic understanding of resources in system programming languages mentioned in the previous paragraph, we proposed a design for extending functional programming languages towards usages from systems programming, centered around the ML language [27]. One motivation is to show the feasibility of basing linear allocation with re-use [44, 37] inside languages that would still leverage state-of-art garbage collection for non-linear values. The design includes the possibility to choose the most suitable allocation mode, with a garbage collector (GC) or with a controlled form of dynamic memory (linear allocation with reuse), based on kinds.

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.

Deciding contextual equivalence of ν-calculus with effectful contexts In [21], we prove decidability for contextual equivalence of the $\lambda\mu\nu$-calculus, that is the simply-typed call-by-value $\lambda\mu$-calculus equipped with booleans and fresh name creation, with contexts taken in $\lambda\mu$ref, that is $\lambda\mu\nu$-calculus extended with higher-order references. The proof exploits a labelled transition system capturing the interactions between $\lambda\mu\nu$ programs and $\lambda\mu$ref contexts. The induced bisimulation equivalence is characterized as the equality of certain trees, inspired by the work of Lassen. Since these trees are computable and finite, decidability follows. Bisimulation coincides also with trace equivalence, which in turn coincides with contextual equivalence.

Modular efficient deconstruction with typed pointer reversal Destructors, responsible for releasing memory and other resources in languages such as C++ and Rust, can lead to stack overflows when releasing a recursive structure that is too deep. In certain cases, it is possible to generate an efficient destructor (non-allocating and tail recursive) using a typed variant of pointer reversal. In [24], we extend this technique by making it more modular, in order to handle abstract types, separate compilation, and unboxed types.

7.4 Program Certifications and Formalisation of Mathematics

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

A Foundational Framework for Modular Cryptographic Proofs in Coq State-separating proofs (SSP) is a recent methodology for structuring game-based cryptographic proofs in a modular way, by using algebraic laws to exploit the modular structure of composed protocols. While promising, this methodology was previously not fully formalized and came with little tool support. We address this by introducing SSProve [12], the first general verification framework for machine-checked state-separating proofs. SSProve combines high-level modular proofs about composed protocols, as proposed in SSP, with a probabilistic relational program logic for formalizing the lower-level details, which together enable constructing machine-checked cryptographic proofs in the Coq proof assistant. Moreover, SSProve is itself fully formalized in Coq, including the algebraic laws of SSP, the soundness of the program logic, and the connection between these two verification styles. To illustrate SSProve, we use it to mechanize the simple security proofs of ElGamal and pseudo-random-function–based encryption. We also validate the SSProve approach by conducting two more substantial case studies: First, we mechanize an SSP security proof of the key encapsulation mechanism–data encryption mechanism (KEM-DEM) public key encryption scheme, which led to the discovery of an error in the original paper proof that has since been fixed. Second, we use SSProve to formally prove security of the sigma-protocol zero-knowledge construction, and we moreover construct a commitment scheme from a sigma-protocol to compare with a similar development in CryptHOL. We instantiate the security proof for sigma-protocols to give concrete security bounds for Schnorr's sigma-protocol.

A Computational Cantor-Bernstein and Myhill's Isomorphism Theorem in Constructive Type Theory: Proof Pearl. The Cantor-Bernstein theorem (CB) from set theory, stating that two sets which can be injectively embedded into each other are in bijection, is inherently classical in its full generality, i.e. implies the law of excluded middle, a result due to Pradic and Brown [48]. Recently, Escardó has provided a proof of CB in univalent type theory, assuming the law of excluded middle. It is a natural question to ask which restrictions of CB can be proved without axiomatic assumptions. In [19], we give a partial

answer to this question contributing an assumption-free proof of CB restricted to enumerable discrete types, i.e. types which can be computationally treated. In fact, we construct several bijections from injections: The first is by translating a proof of the Myhill isomorphism theorem from computability theory—stating that 1-equivalent predicates are recursively isomorphic-to constructive type theory, where the bijection is constructed in stages and an algorithm with an intricate termination argument is used to extend the bijection in every step. The second is also constructed in stages, but with a simpler extension algorithm sufficient for CB. The third is constructed directly in such a way that it only relies on the given enumerations of the types, not on the given injections. We aim at keeping the explanations simple, accessible, and concise in the style of a "proof pearl". All proofs are machine-checked in Coq but should transport to other foundations: they do not rely on impredicativity, on choice principles, or on large eliminations.

Constructive and Synthetic Reducibility Degrees: Post's Problem for Many-one and Truth-table Reducibility in Coq We present in [18] a constructive analysis and machine-checked theory of one-one, many-one, and truth-table reductions based on synthetic computability theory in the Calculus of Inductive Constructions, the type theory underlying the proof assistant Coq. We give elegant, synthetic, and machine-checked proofs of Post's landmark results that a simple predicate exists, is enumerable, undecidable, but many-one incomplete (Post's problem for many-one reducibility), and a hypersimple predicate exists, is enumerable, undecidable, but truth-table incomplete (Post's problem for truth-table reducibility). In synthetic computability, one assumes axioms allowing to carry out computability theory with all definitions and proofs purely in terms of functions of the type theory with no mention of a model of computation. Proofs can focus on the essence of the argument, without having to sacrifice formality. Synthetic computability also clears the lense for constructivisation. Our constructively careful definition of simple and hypersimple predicates allows us to not assume classical axioms, not even Markov's principle, still yielding the expected strong results.

A First Order Theory of Diagram Chasing In [23], 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.

Design patterns of hierarchies for order structures Using order structures in a proof assistant naturally raises the problem of working with multiple instances of a same structure over a common type of elements. This goes against the main design pattern of hierarchies used for instance in Coq's MathComp or Lean's mathlib libraries, where types are canonically associated to at most one instance and instances share a common overloaded syntax. In [31], we present new design patterns to leverage these issues, and apply them to the formalization of order structures in the MathComp library. A common idea in these patterns is underloading, i.e., a disambiguation of operators on a common type. In addition, our design patterns include a way to deal with duality in order structures in a convenient way. We hence formalize a large hierarchy which includes partial orders, semilattices, lattices as well as many variants. We finally pay a special attention to order substructures. We introduce a new kind of structure called prelattice. They are abstractions of semilattices, and allow us to deal with finite lattices and their sublattices within a common signature. As an application, we report on significant simplifications of the formalization of the face lattices of polyhedra in the Coq-Polyhedra library.

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 [32], 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.

8 Bilateral contracts and grants with industry

8.1 Bilateral Contracts with Industry

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

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

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

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).

9 Partnerships and cooperations

9.1 International initiatives

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

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.

9.2 European initiatives

9.2.1 H2020 projects

FRESCO 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: Assia Mahboubi

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.

Coqaml Coqaml project on cordis.europa.eu

Title: Verified Extraction from Coq to OCaml with GADTs

Duration: From December 1, 2021 to November 30, 2023

Partners:

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

Inria contact: Nicolas Tabareau

Coordinator: Nicolas Tabareau

Summary: The Coq proof assistant is a popular tool to verify the correctness of security-critical software. The CompCert C compiler, some implementations of blockchain languages, and the implementation of the P-256 elliptic curve in Google's BoringSSL library are all OCaml programs obtained by extraction from Coq functions.

While a type checker for Coq has recently been verified via a machine-checked mathematical proof based on the MetaCoq project for verified meta-programming, the extraction process from Coq to OCaml is still part of the trusted computing base (TCB).

The Coqaml project will minimise the TCB for extracted programs even further by also providing a machine-checked correctness proof for the extraction mechanism to OCaml. Under the supervision of Nicolas Tabareau, head of the Inria Gallinette team in Nantes, the experienced researcher (ER) will implement Coq's extraction as mechanically verified MetaCoq-plugin, obtaining the guarantee that extracted OCaml programs behave exactly like the Coq function specified.

In order to be usable in industrial applications, Coqaml will include a novel extraction targeting generalized algebraic datatypes (GADTs) in OCaml. The project includes a secondment of the ER to Nomadic Labs in Paris, who require GADTs as target for Coq's extraction. The intermediate semantic correctness proof for type and proof erasure, allowing axioms like functional extensionality or proof irrelevance in verified programs, can also be exploited in other extraction projects like the CertiCoq compiler from Coq to C code.

The Coqaml project is interdisciplanary by design, spanning logic, type theory, programming languages, and compilers. The density of some of the world's leading experts on Coq and type theory in the Gallinette team and the expertise at Nomadic Labs will ensure that the environment is

ideal for the success of the Coqaml project and the most beneficial development of the ER, greatly enhancing his future career prospects.

9.3 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.

DyVerSe

Participants: Guillaume Munch-Maccagnoni.

Title: Dynamic Versatile Semantics

Program: ANR AAPG2019,

Type: PRC, CES 48

Duration: Jan 2020 - Dec 2023

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

Local Contact: Guillaume Munch-Maccagnoni

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

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.

10 Dissemination

Participants: Rémi Douence, Guilhem Jaber, Assia Mahboubi, Kenji Maillard, Guilaume Munch Maccagnoni, Pierre-Marie Pédrot, Matthieu Sozeau, Nicolas Tabareau.

10.1 Promoting scientific activities

10.1.1 Scientific events: organisation

General chair, scientific chair

- Guillaume Munch-Maccagnoni organizes the first conference on undone science in computer science (Undone Computer Science), to be held in 2024 in Nantes.

Member of the organizing committees

- Assia Mahboubi has co-organized the workshop "Machine-Checked Mathematics" at the Lorentz Centre, Leiden, The Netherlands.
- Assia Mahboubi has co-organized the workshop "Certified Symbolic-Numeric Computation" in Lyon, France, in the frame of the special year on Recent Trends in Computer Algebra.
- Assia Mahboubi has co-organized the joint special session on Machine-Checked Mathematics for international conferences MFPS XXXIX and Calco 2023.

10.1.2 Scientific events: selection

Chair of conference program committees

- Guillaume Munch-Maccagnoni has served as chair for the program commitee of the first conference on undone science in computer science (Undone Computer Science).

Member of the conference program committees

- Assia Mahboubi has served on the program committees of the ITP 2023 and POPL'24 international conferences, and of the LFMTP'23 and TYPES'23 workshops.
- Pierre-Marie Pédrot has served in the program committees of the CPP'23 and POPL'24 conferences, and of the TYPES'23 workshop.
- Matthieu Sozeau has served on the program committees of the WITS 2024 workshop.
- Guillaume Munch-Maccagnoni has served on the program committee of the ML'23 workshop.
- Kenji Maillard has served in the program committee of the CPP'23, OOPSLA'23 and OOPSLA'24 conferences and the LANMR'23 workshop.
- Guilhem Jaber has served in the program committees of the MFPS'23 and POPL'24 international conference, and on the FICS'24 and GALOP'24 workshops.

Reviewer

- Pierre-Marie Pédrot was a reviewer for ITP'23, LICS'23 and MFCS'23.
- Guillaume Munch-Maccagnoni was a reviewer for FSCD'23.
- Kenji Maillard was a reviewer for POPL'24 and ESOP'24.
- Guilhem Jaber was a reviewer for LICS'23, FoSSaCS'23 and MFSC'23.

10.1.3 Journal

Member of the editorial boards

- Assia Mahboubi serves on the editorial board of the Journal of Automated Reasoning.
- Pierre-Marie Pédrot was the co-editor of the LIPIcs volume associated to TYPES 2022.

Reviewer - reviewing activities

- Guilhem Jaber was a reviewer for LMCS.

10.1.4 Invited talks

- Assia Mahboubi has given invited talks at th IPAM workshop on Machine Assisted Proofs (Los Angeles, USA) at the Interacciones en la Frontera 2023 on-line seminar (Mexico), at the Intercity Number Theory seminar (Amsterdam, the Netherlands), at the seminar of philosophy "Formalisation du calcul et des preuves assistées par ordinateur" (Paris, France).
- Matthieu Sozeau has given an invited lecture at the 2023 Summer School on Proof Theory and its Applications (Barcelona, Spain).

10.1.5 Leadership within the scientific community

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

10.1.6 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).

10.2 Teaching - Supervision - Juries

10.2.1 Teaching

- Licence : Julien Cohen, Discrete Mathematics, 48h, L1 (IUT), IUT Nantes, France
- Licence : Julien Cohen, Introduction to proof assistants (Coq), 8h, L2 (PEIP : IUT/Engineering school), Polytech Nantes, France
- Licence : Julien Cohen, Functional Programming (Scala), 22h, L2 (IUT), IUT Nantes, France
- Master : Julien Cohen, Object oriented programming (Java), 32h, M1 (Engineering school), Polytech Nantes, France
- Master : Julien Cohen, Functional programming (OCaml), 18h, M1 (Engineering school), Polytech Nantes, France
- Master : Julien Cohen, Tools for software engineering (proof with Frama-C, test, code management), 20h, M1 (Engineering school), Polytech Nantes, France
- Licence : Rémi Douence, Object Oriented Design and Programming, 45h, L1 (engineers), IMT-Atlantique, Nantes, France
- Licence : Rémi Douence, Object Oriented Design and Programming Project, 30h, L1 (apprenticeship), IMT-Atlantique, Nantes, France
- Master : Rémi Douence, Functional Programming with Haskell, 45h, M1 (engineers), IMT-Atlantique, Nantes, France
- Master : Rémi Douence, Functional Programming with Haskell, 20h, M1 (apprenticeship), IMT-Atlantique, Nantes, France
- Master : Rémi Douence, Formal Methods: Model checking with Alloy and from Haskell to Coq, 11h, M1 (apprenticeship), IMT-Atlantique, Nantes, France
- Master : Rémi Douence, Introduction to scientific research in computer science (Project: compilation in Java of Haskell Class Types), 45h, M2 (apprenticeship), IMT-Atlantique, Nantes, France

- Licence : Hervé Grall, Algorithms and Discrete Mathematics, 25h , L3 (engineers), IMT-Atlantique, Nantes, France
- Licence : Hervé Grall, Object Oriented Design and Programming, 25h , L3 (engineers), IMT-Atlantique, Nantes, France
- Licence, Master : Hervé Grall, Modularity and Typing, 40h, L3 and M1, IMT-Atlantique, Nantes, France
- Master : Hervé Grall, Service-oriented Computing, 40h, M1 and M2, IMT-Atlantique, Nantes, France
- Master : Hervé Grall, Research Project - (Linear) Logic Programming in Coq, 90h (1/3 supervised), M1 and M2, IMT-Atlantique, Nantes, France
- Licence : Guilhem Jaber, 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, 9h, M2, Nantes Université, France
- Master : Nicolas Tabareau, Homotopy Type Theory, 24h, M2 LMFI, Université Paris Diderot, France
- Master : Matthieu Sozeau, Proof Assistants, 24h, M2 MPRI, Université Paris Diderot, France

10.2.2 Supervision

- Antoine Allioux defended his PhD in July 2023, Structures supérieures en théorie des types homotopiques [28], Université Paris Cité, advisors: Pierre-Louis Curien, Eric Finster, Matthieu Sozeau.
- Martin Baillon has defended his PhD on December 2023, Modèles syntaxiques de la théorie des types et principes de continuité, Nantes Université, advisors: Assia Mahboubi and Pierre-Marie Pédrot.
- 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 pogress: Sidney Congard, Towards a linear functional translation for borrowing, IMT-A, advisors: Rémi Douence and Guillaume Munch-Maccagnoni.
- PhD in progress: Pierre Benjamin Giraud, Formalizing extraction of Coq to OCaml, Nantes Université, advisors: Pierre-Marie Pédrot, 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: Hamza Jaafar, Sémantique des jeux opérationnelle pour le langage de programmation OCaml, Nantes Université, advisors: Guilhem Jaber and Nicolas Tabareau.
- Assia Mahboubi is the promoter of the PhD of Alain Chavarri Villarello (Vrij Universiteit Amsterdam).
- Nicolas Tabareau has co-supervised the internshp of Nils Lauermann and Robin Jourde.
- Guillaume Munch-Maccagnoni has supervised the internship of Jean Caspar, and co-supervised the internship of Eleonor Mangel.
- Guilhem Jaber and Kenji Maillard have co-supervised the internship of Virgil Marionneau.
- Assia Mahboubi has supervised the internship of Benoît Guillemet.

10.2.3 Juries

- Assia Mahboubi has served in the PhD jury of Rebecca Zucchini, Université Paris-Saclay.
- Assia Mahboubi has served in the HDR jury of Filippo A. E. Nuccio Mortarino Majno di Capriglio, Université Jean Monnet Saint-Étienne.
- Assia Mahboubi has served on the committee of the SIF Gilles Kahn PhD prize.
- Pierre-Marie Pédrot has served in the Master 2 jury of Thomas Traversié, CentraleSupélec.
- Nicolas Tabareau has served in the PhD jury of Enzo Crance, Inria - MERCE (Mitsubishi Electrics Research Center Europe).

10.3 Popularization

10.3.1 Articles and contents

Guilhem Jaber participated in the production of a popular science magazine during the residence of journalists from "Les Autres Possible".

10.3.2 Education

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

10.3.3 Interventions

Assia Mahboubi has given a talk at the Université Ouverte Lyon 1 (Villeurbanne).

11 Scientific production

11.1 Major publications

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

[2] 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] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

11.2 Publications of the year

International journals

[12] P. Haselwarter, E. Rivas, A. van Muylder, T. Winterhalter, C. Abate, N. Sidorenco, C. Hriţcu, K. Maillard and B. Spitters. 'SSProve: A Foundational Framework for Modular Cryptographic Proofs in Coq'. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 45.3 (30th Sept. 2023), pp. 1–61. DOI: 10.1145/3594735. URL: https://hal.science/hal-04273257.

[13] 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 (11th Jan. 2023), p. 74. DOI: 10.1145/3571739. URL: https://hal.science/hal-03857705.

Invited conferences

[14] M. Sozeau. 'MetaCoq : de la métaprogrammation à l'extraction certifiée pour Coq'. In: JFLA 2024 - 35es Journées Francophones des Langages Applicatifs. Saint-Jacut-de-la-Mer, France, 2024, pp. 1–1. URL: https://inria.hal.science/hal-04407164.

International peer-reviewed conferences

[15] A. Adjedj, M. Lennon-Bertrand, K. Maillard and L. Pujet. 'Engineering logical relations for MLTT in Coq'. In: TYPES 2023 - 29th International Conference on Types for Proofs and Programs. Valencia, Spain, 2023, pp. 1–3. URL: https://inria.hal.science/hal-04379245.

[16] V. Blot, D. Cousineau, E. Crance, L. Dubois de Prisque, C. Keller, A. Mahboubi and P. Vial. 'Compositional pre-processing for automated reasoning in dependent type theory'. In: CPP 2023 - Certified Programs and Proofs. Boston, United States, 2023, pp. 1–15. DOI: 10.1145/3573105.3575676. URL: https://inria.hal.science/hal-03901019.

[17] C. Cohen, E. Crance and A. Mahboubi. 'Trocq: Proof Transfer for Free, With or Without Univalence'. In: ESOP 2024 - 33rd European Symposium on Programming. Luxembourg, Luxembourg, 2024, pp. 1–29. URL: https://hal.science/hal-04177913.

[18] Y. Forster and F. Jahn. 'Constructive and Synthetic Reducibility Degrees: Post's Problem for Many-one and Truth-table Reducibility in Coq'. In: CSL 2023 - 31st EACSL Annual Conference on Computer Science Logic. Warsaw, Poland, 13th Feb. 2023, pp. 1–21. DOI: 10.4230/LIPIcs.CSL.2023.16. URL: https://inria.hal.science/hal-03901942.

[19] Y. Forster, F. Jahn and G. Smolka. 'A Computational Cantor-Bernstein and Myhill's Isomorphism Theorem in Constructive Type Theory: Proof Pearl'. In: CPP 2023 - 12th ACM SIGPLAN International Conference on Certified Programs and Proofs. Boston, United States: ACM, 16th Jan. 2023, pp. 1–8. DOI: 10.1145/3573105.3575690. URL: https://inria.hal.science/hal-03891390.

[20] G. Gilbert, Y. Leray, N. Tabareau and T. Winterhalter. 'The Rewster: The Coq Proof Assistant with Rewrite Rules'. In: TYPES 2023 - 29th International Conference on Types for Proofs and Programs. Valencia, Spain, 2023, pp. 1–3. URL: https://inria.hal.science/hal-04403667.

[21] D. Hirschkoff, G. Jaber and E. Prebet. 'Deciding contextual equivalence of ν-calculus with effectful contexts (full version)'. In: Foundations of Software Science and Computation Structures - 26th International Conference, FoSSaCS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023. Paris, France, 22nd Apr. 2023. URL: https://hal.science/hal-03955303.

[22] A. Mahboubi and G. Melquiond. 'Manifest Termination'. In: TYPES 2023 - 29th International Conference on Types for Proofs and Programs. Valencia, Spain, 12th June 2023, pp. 1–3. URL: https://inria.hal.science/hal-04172297.

[23] 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.

National peer-reviewed Conferences

[24] J. Caspar and G. Munch-Maccagnoni. 'Modular efficient deconstruction with typed pointer reversal'. In: JFLA 2024 - 35es Journées Francophones des Langages Applicatifs. Saint-Jacut-de-la-Mer, France, 2024, pp. 1–10. URL: https://inria.hal.science/hal-04406342.

[25] S. Congard. 'Towards a linear functional translation for borrowing'. In: JFLA 2024 - 35es Journées Francophones des Langages Applicatifs. Saint-Jacut-de-la-Mer, France, 2024, pp. 1–10. URL: https://hal.science/hal-04360462.

Conferences without proceedings

[26] G. Gilbert, P.-M. Pédrot, M. Sozeau and N. Tabareau. 'From Lost to the River: Embracing Sort Proliferation'. In: TYPES 2023 - 29th International Conference on Types for Proofs and Programs. Valencia, Spain, 2023, pp. 1–2. URL: https://inria.hal.science/hal-04378939.

[27] G. Munch-Maccagnoni. 'Resource polymorphism: proposal for integrating first-class resources into ML'. In: Higher-order, Typed, Inferred, Strict: ML Family Workshop 2023. Seattle, United States, 2023. URL: https://hal.science/hal-04332484.

Doctoral dissertations and habilitation theses

[28] A. Allioux. 'Higher Structures in Homotopy Type Theory'. Université Paris Cité, 17th July 2023. URL: https://theses.hal.science/tel-04335842.

Reports & preprints

[29] A. Adjedj, M. Lennon-Bertrand, K. Maillard, P.-M. Pédrot and L. Pujet. *Martin-Löf à la Coq*. 2024. DOI: 10.1145/3636501.3636951. URL: https://hal.science/hal-04214008.

[30] R. Affeldt, Y. Bertot, C. Cohen, P. Roux, K. Sakaguchi and E. Tassi. *Porting Coq Scripts to the Mathematical Components Library Version 2*. Inria Sophia Antipolis - Méditerranée, Université Côte d'Azur; National Institute of Advanced Industrial Science and Technology (AIST), Japan; ONERA / DTIS, Université de Toulouse, France, 20th June 2023, pp. 1–12. URL: https://hal.science/hal-04225130.

[31] X. Allamigeon, Q. Canu, C. Cohen, K. Sakaguchi and P.-Y. Strub. *Design patterns of hierarchies for order structures*. 28th Feb. 2023. URL: https://inria.hal.science/hal-04008820.

[32] Y. Forster, M. Sozeau and N. Tabareau. *Verified Extraction from Coq to OCaml*. 10th Nov. 2023. URL: https://inria.hal.science/hal-04329663.

[33] T. Laurent, M. Lennon-Bertrand and K. Maillard. *Definitional Functoriality for Dependent (Sub)Types*. 23rd Oct. 2023. URL: https://hal.science/hal-04160858.

[34] P.-M. Pédrot. *Pursuing Shtuck*. 20th Oct. 2023. URL: https://inria.hal.science/hal-04251754.

[35] 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*. 21st Apr. 2023. URL: https://inria.hal.science/hal-04077552.

11.3 Cited publications

[36] T. Altenkirch, C. McBride and W. Swierstra. 'Observational equality, now!' In: *Proceedings of the ACM Workshop on Programming Languages meets Program Verification (PLPV 2007)*. Freiburg, Germany, Oct. 2007, pp. 57–68.

[37] H. G. Baker. 'Lively Linear Lisp: "Look Ma, No Garbage!"' In: *SIGPLAN Not.* 27.8 (Aug. 1992), pp. 89–98. DOI: 10.1145/142137.142162. URL: https://doi.org/10.1145/142137.142162.

[38] 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.

[39] Coq Development Team, The. *The Coq proof assistant reference manual*. Version 8.5. 2015. URL: http://coq.inria.fr.

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

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

[42] 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.

[43] 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.

[44] Y. Lafont. 'The linear abstract machine'. In: *Theoretical Computer Science* 59.1 (1988), pp. 157–180. DOI: https://doi.org/10.1016/0304-3975(88)90100-4. URL: https://www.sciencedirect.com/science/article/pii/0304397588901004.

[45] 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.

[46] 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.

[47] 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.

[48] P. Pradic and C. E. Brown. 'Cantor-Bernstein implies Excluded Middle'. Update: fixed an error on the applicability of thm 1, added some acks and a ref. Dec. 2021. URL: https://hal.science/hal-02103517.

[49] 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.

[50] Univalent Foundations Project. *Homotopy Type Theory: Univalent Foundations for Mathematics*. http://homotopytypetheory.org/book, 2013.