RESEARCH CENTRE

Rennes - Bretagne Atlantique

IN PARTNERSHIP WITH:

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

2020
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

Contents

Project-Team GALLINETTE

Creation of the Team: 2017 May 01, updated into 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, Researcher, HDR]

- Guillaume Munch [Inria, Researcher]

- Pierre-Marie Pédrot [Inria, Researcher]

- Matthieu Sozeau [Inria, Researcher]

Faculty Members

- Julien Cohen [Univ de Nantes, Associate Professor]

- Rémi Douence [IMT Atlantique, Associate Professor, HDR]

- Hervé Grall [IMT Atlantique, Associate Professor]

- Guilhem Jaber [Univ de Nantes, Associate Professor]

Post-Doctoral Fellows

- Marie Kerjean [Inria, until Feb 2020]

- Maxime Lucas [Inria, until Aug 2020]

- Kenji Maillard [Inria]

PhD Students

- Antoine Allioux [Inria]

- Martin Baillon [Inria, from Oct 2020]

- Meven Bertrand [Univ de Nantes]

- Enzo Crance [Mitsubishi Electric, from Nov 2020]

- Gaetan Gilbert [Inria, until Jan 2020]

- Pierre Giraud [Inria, from Dec 2020]

- Xavier Montillet [Univ de Nantes, until Nov 2020]

- Loic Pujet [Univ de Nantes]

- Theo Winterhalter [Univ de Nantes, until Sep 2020]

Technical Staff

- Simon Boulier [Inria, Engineer, until Sep 2020]

- Gaetan Gilbert [Inria, Engineer, from Feb 2020]

- Marie Kerjean [Inria, Engineer, from Mar 2020 until Sep 2020]

Interns and Apprentices

- Martin Baillon [Inria, from Mar 2020 until Aug 2020]

- Peio Borthelle [École Normale Supérieure de Lyon, from Oct 2020]

- Guillaume Combette [École Normale Supérieure de Lyon, until Feb 2020]

- Enzo Crance [Mitsubishi Electric, from Feb 2020 until Jul 2020]

- Theo Vignon [Ecole normale supérieure Paris-Saclay, from Jun 2020 until Jul 2020]

Administrative Assistant

- Anne-Claire Binétruy [Inria]

2 Overall objectives

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

- The goal is to advance proof assistants both as certified programming languages and mechanised logical systems. Advanced programming and mathematical paradigms must be integrated, notably dependent types and effects. The distinctive approach is to implement new programming and logical paradigms on top of Coq by considering the latter as a target language for compilation.

- The aim of foundational investigations is to extend the boundaries of the Curry-Howard correspondence. It is seen both as providing foundations for programming languages and logic, and as a purveyor of techniques essential to the development of proof assistants. Under this perspective, the development of proof assistants is seen as a total experiment using the correspondence in every aspect: programming languages, type theory, proof theory, rewriting and algebra.

3 Research program

3.1 Scientific Context

Software quality is a requirement that is becoming more and more prevalent, by now far exceeding the traditional scope of embedded systems. The development of tools to construct software that respects a given specification is a major challenge facing computer science. *Proof assistants* such as Coq [54] provide a formal method whose central innovation is to produce *certified programs* by transforming the very activity of programming. Programming and proving are merged into a single development activity, informed by an elegant but rigid mathematical theory inspired by the correspondence between programming, logic and algebra: the *Curry-Howard correspondence*. For the certification of programs, this approach has shown its efficiency in the development of important pieces of certified software such as the C compiler of the CompCert project [84]. The extracted CompCert compiler is reliable and efficient, running only 15% slower than GCC 4 at optimisation level 2 (`gcc -O2`), a level of optimisation that was considered before to be highly unreliable.

Proof assistants can also be used to *formalise mathematical theories*: they not only provide a means of representing mathematical theories in a form amenable to computer processing, but their internal logic provides a language for reasoning about such theories. In the last decade, proof assistants have been used to verify extremely large and complicated proofs of recent mathematical results, sometimes requiring either intensive computations [65, 70] or intricate combinations of a multitude of mathematical theories [66]. But formalised mathematics is more than just proof checking and proof assistants can help with organisation mathematical knowledge or even with the discovery of new constructions and proofs.

Unfortunately, the rigidity of the theory behind proof assistants impedes their expressiveness both as programming languages and as logical systems. For instance, a program extracted from Coq only uses a purely functional subset of OCaml, leaving behind important means of expression such as side-effects

and objects. Limitations also appear in the formalisation of advanced mathematics: proof assistants do not cope well with classical axioms such as excluded middle and choice which are sometimes used crucially. The fact of the matter is that the development of proof assistants cannot be dissociated from a reflection on the nature of programs and proofs coming from the Curry-Howard correspondence. In the EPC Gallinette, we propose to address several drawbacks of proof assistants by pushing the boundaries of this correspondence.

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

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

- **(Martin-Löf) Type theory:** introduced by Martin-Löf in 1971, this formalism [91] is both a programming language and a logical system. The central ingredient is the use of *dependent types* to allow fine-grained invariants to be expressed in program types. In 1985, Coquand and Huet developed a similar system called the *calculus of constructions*, which served as logical foundation of the first implementation of Coq. This kind of systems is still under active development, especially with the recent advent of homotopy type theory (HoTT) [113] which gives a new point of view on types and the notion of equality in type theory.

- **The theory of effects:** starting in the 1980's, Moggi [96] and Girard [63] put forward monads and co-monads as describing various compositional notions of computation. In this theory, programs can have side-effects (state, exceptions, input-output), logics can be non-intuitionistic (linear, classical), and different computational universes can interact (modal logics). Recently, the safe and automatic management of resources has also seen a coming of age (Rust, Modern C++) confirming the importance of linear logic for various programming concepts. It is now understood that the characteristic feature of the theory of effects is sensitivity to *evaluation order*, in contrast with type theory which is built around the assumption that evaluation order is irrelevant.

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

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

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

2. Advancing the foundations of effects within the Curry-Howard approach. The objective is to pave the way for the integration of effects in proof assistants and to prototype the corresponding implementation. This integration should allow for not only certified programming with effects, but also the expression of more powerful logics (Section 3.3).

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

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

3.2   Enhance the computational and logical power of proof assistants

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

We propose to investigate how expressive logical transformations such as forcing [76] and sheaf construction might be used to enhance the computational and logical power of proof assistants—with a particular emphasis on their implementation in the Coq proof assistant by the means of effective translations (or compilation phases). One of the main topics of this task, in connection to the ERC project CoqHoTT, is the integration in Coq of new concepts inspired by homotopy type theory [113] such as the univalence principle, and higher inductive types.

3.2.1   A definitional proof-irrelevant version of Coq.

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

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

The univalence principle is becoming widely accepted as a very promising avenue to provide new foundations for mathematics and type theory. However, this principle has not yet been incorporated into a proof assistant. Indeed, the very mathematical structures (known as ∞-groupoids) motivating the theory remain to this day an active area of research. Moreover, a correct and decidable type checking procedure for the whole theory raises both computational complexity and logical coherence issues. Observational type theory [36], as implemented in Epigram, provides a first-stage approximation to homotopy type theory, but only deals with functional extensionality and does not capture univalence. Coquand and his collaborators have obtained significant results on the computational meaning of univalence using cubical sets [44, 48]. Bickford has initiated a promising formalisation work [1] in the NuPRL system. However, a complete formalisation in intensional type theory remains an open problem.

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

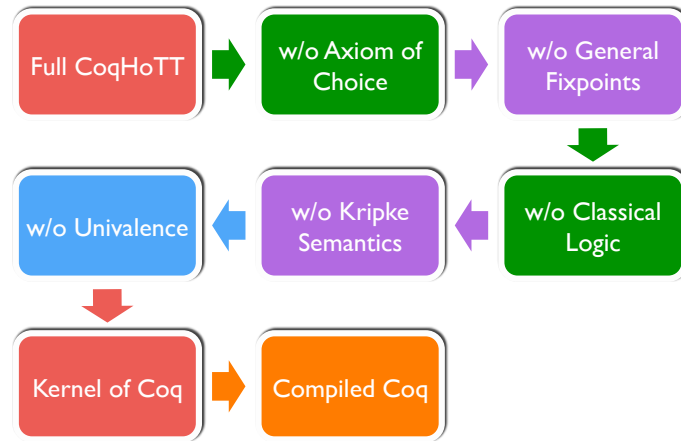[1] http://www.nuprl.org/wip/Mathematics/cubical!type!theory/index.html

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

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

Extending the power of a logic using model transformations (*e.g.,* forcing transformation [77, 76] or the sheaf construction [106]) is a classic topic of mathematical logic [52, 82]. However, these ideas have not been much investigated in the setting of type theory, even though they may provide a useful framework for extending the logical power of proof assistant in a modular way. There is a good reason for this: with a syntactic notion of equality, the underlying structure of type theory does not conform to the structure of topos used in mathematical logic. A direct incorporation of the standard techniques is therefore not possible. However, a univalent notion of equality brings type theory closer to the required algebraic structure, as it corresponds to the notion of ∞-topos recently studied by Lurie [89]. The goal of this task is to revisit model transformations in the light of the univalence principle, and to obtain in this way new internal transformations in type theory which can in turn be seen as compilation phases. The general notion of an internal syntactical translation has already been investigated in the team [45].

3.2.4   Methodology: Extending type theory with different compilation phases

The Gallinette project advocates the use of distinct compilation phases as a methodology for the design of a new generation of proof assistants featuring modular extensions of a core logic. The essence of a compiler is the separation of the complexity of a translation process into modular stages, and the organization of their re-composition. This idea finds a natural application in the design of complex proof assistants (Figure 1). For instance, the definition of type classes in Coq follows this pattern, and is morally given by the means of a translation into a type-class free kernel. More recently, a similar approach by compilation stages, using the forcing transformation, was used to relax the strict positivity condition guarding inductive types [77, 76]. We believe that this flavour of compilation-based strategies offers a promising direction of investigation for the purpose of defining a decidable type checking algorithm for HoTT.

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

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

We mean *effects* in a broad sense that encompasses both Moggi's monads [96] and Girard's linear logic [63]. These two seminal works have given rise to respective theories of effects (monads) and resources (co-monads). Recent advances, however have unified these two lines of thought: it is now clear that the defining feature of effects, in the broad sense, is sensitivity to evaluation order [85, 55].

In contrast, the type theory that forms the foundations of proof assistants is based on pure λ-calculus and is built on the assumption that evaluation order is irrelevant. Evaluation order is therefore the blind spot of type theory. In Moggi [97], integrating the dependent types of type theory with monads is *"the next difficult step [...] currently under investigation"*.

Any realistic program contains effects: state, exceptions, input-output. More generally, evaluation order may simply be important for complexity reasons. With this in mind, many works have focused on certified programming with effects: notably Ynot [101], and more recently F* [112] and Idris [46], which propose various ways for encapsulating effects and restricting the dependency of types on effectful terms. Effects are either specialised, such as the monads with Hoare-style pre- and post-conditions found in Ynot or F*, or more general, such as the algebraic effects implemented in Idris. But whereas there are several experiments and projects pursuing the certification of programs with effects, each making its own choices on how effects and dependency should be merged, there is on the other hand a deficit of logical and semantic investigations.

We propose to develop the foundations of a type theory with effects taking into account the logical and semantic aspects, and to study their practical and theoretical consequences. A type theory that integrates effects would have logical, algebraic and computational implications when viewed through the Curry-Howard correspondence. For instance, effects such as control operators establish a link with classical proof theory [68]. Indeed, control operators provide computational interpretations of type isomorphisms such as $A \cong \neg\neg A$ and $\neg\forall x A \cong \exists x \neg A$ (e.g. [98]), whereas the conventional wisdom of type theory holds that such axioms are non-constructive (this is for instance the point of view that has been advocated so far in homotopy type theory [113]). Another example of an effect with logical content is state (more precisely memoization) which is used to provide constructive content to the classical dependent axiom of choice [43, 80, 72]. In the long term, a whole body of literature on the constructive content of classical proofs is to be explored and integrated, providing rich sources of inspiration: Kohlenbach's proof mining [79] and Simpson's reverse mathematics [110], for instance, are certainly interesting to investigate from the Curry-Howard perspective.

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

3.3.1 Models for integrating effects with dependent types

A crucial step is the integration of dependent types with effects, a topic which has remained *"currently under investigation"* [97] ever since the beginning. The difficulty resides in expressing the dependency of types on terms that can perform side-effects during the computation. On the side of denotational semantics, several extensions of categorical models for effects with dependent types have been proposed [33, 114] using axioms that should correspond to restrictions in terms of expressivity but whose practical implications, however, are not immediately transparent. On the side of logical approaches [72, 73, 83, 95], one first considers a drastic restriction to terms that do not compute, which is then relaxed by semantic means. On the side of systems for certified programming such as F*, the type system ensures that types only depend on pure and terminating terms.

Thus, the recurring idea is to introduce restrictions on the dependency in order to establish an encapsulation of effects. In our approach, we seek a principled description of this idea by developing the concept of *semantic value* (thunkables, linears) which arose from foundational considerations [62, 109, 99] and whose relevance was highlighted in recent works [86, 103]. The novel aspect of our approach is to seek a proper extension of type theory which would provide foundations for a classical type theory with axiom of choice in the style of Herbelin [72], but which moreover could be generalised to effects other than just control by exploiting an abstract and adaptable notion of semantic value.

3.3.2 Intuitionistic depolarisation

In our view, the common idea that evaluation order does not matter for pure and terminating computations should serve as a bridge between our proposals for dependent types in the presence of effects and traditional type theory. Building on the previous goal, we aim to study the relationship between semantic values, purity, and parametricity theorems [108, 64]. Our goal is to characterise parametricity as a form of

intuitionistic *depolarisation* following the method by which the first game model of full linear logic was given (Melliès [92, 93]). We have two expected outcomes in mind: enriching type theory with intensional content without losing its properties, and giving an explanation of the dependent types in the style of Idris and F* where purity- and termination-checking play a role.

3.3.3 Developing the rewriting theory of calculi with effects

An integrated type theory with effects requires an understanding of evaluation order from the point of view of rewriting. For instance, rewriting properties can entail the decidability of some conversions, allowing the automation of equational reasoning in types [31]. They can also provide proofs of computational consistency (that terms are not all equivalent) by showing that extending calculi with new constructs is conservative [111]. In our approach, the λ-calculus is replaced by a calculus modelling the evaluation in an abstract machine [56]. We have shown how this approach generalises the previous semantic and proof-theoretic approaches [37, 85, 87], and overcomes their shortcomings [100].

One goal is to prove computational consistency or decidability of conversions purely using advanced rewriting techniques following a technique introduced in [111]. Another goal is the characterisation of weak reductions: extensions of the operational semantics to terms with free variables that preserve termination, whose iteration is equivalent to strong reduction [32, 60]. We aim to show that such properties derive from generic theorems of higher-order rewriting [107], so that weak reduction can easily be generalised to richer systems with effects.

3.3.4 Direct models and categorical coherence

Proof theory and rewriting are a source of *coherence theorems* in category theory, which show how calculations in a category can be simplified with an embedding into a structure with stronger properties [90, 81]. We aim to explore such results for categorical models of effects [85, 55]. Our key insight is to consider the reflection between *indirect and direct models* [62, 99] as a coherence theorem: it allows us to embed the traditional models of effects into structures for which the rewriting and proof-theoretic techniques from the previous section are effective.

Building on this, we are further interested in connecting operational semantics to 2-category theory, in which a second dimension is traditionally considered for modelling conversions of programs rather than equivalences. This idea has been successfully applied for the λ-calculus [78, 74] but does not scale yet to more realistic models of computation. In our approach, it has already been noticed that the expected symmetries coming from categorical dualities are better represented, motivating a new investigation into this long-standing question.

3.3.5 Models of effects and resources

The unified theory of effects and resources [55] prompts an investigation into the semantics of safe and automatic resource management, in the style of Modern C++ and Rust. Our goal is to show how advanced semantics of effects, resources, and their combination arise by assembling elementary blocks, pursuing the methodology applied by Melliès and Tabareau in the context of continuations [94]. For instance, combining control flow (exceptions, return) with linearity allows us to describe in a precise way the "Resource Acquisition Is Initialisation" idiom in which the resource safety is ensured with scope-based destructors. A further step would be to reconstruct uniqueness types and borrowing using similar ideas.

3.4 Language extensions for the scaling of proof assistants

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

3.4.1 Gradual Certified Programming

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

Indeed, on the way to full verification, programmers can take advantage of a gradual approach in which some properties are simply asserted instead of proven, subject to dynamic verification. Tabareau and Tanter have made preliminary progress in this direction [58]. This work, however, suffers from a number of limitations, the most important being the lack of a mechanism for handling the possibility of runtime errors within Coq. Instead of relying on axioms, this project will explore the application of Section 3.3 to embed effects in Coq. This way, instead of postulating axioms for parts of the development that are too hard/marginal to be dealt with, the system adds dynamic checks. Then, after extraction, we get a program that corresponds to the initial program but with dynamic checks for parts that have not been proven, ensuring that the program will raise an error instead of going outside its specification.

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

3.4.2 Imperative features and object polymorphism in the Coq proof assistant

Imperative features. Abstract data types (ADTs) become useful as the size of programs grows since they provide for a modular approach, allowing abstractions about data to be expressed and then instantiated. Moreover, ADTs are natural concepts in the calculus of inductive constructions. But while it is easy to declare an ADT, it is often difficult to implement an efficient one. Compare this situation with, for example, Okasaki's purely functional data structures [102] which implement ADTs like queues in languages with imperative features. Of course, Okasaki's queues enforce some additional properties for free, such as persistence, but the programmer may prefer to use and to study a simpler implementation without those additional properties. Also in certified symbolic computation (see 3.5.3), an efficient functional implementation of ADTs is often not available, and efficiency is a major challenge in this area. Relying on the theoretical work done in 3.3, we will equip Coq with imperative features and we will demonstrate how they can be used to provide efficient implementations of ADTs. However, it is also often the case that imperative implementations are hard-to-reason-on, requiring for instance the use of separation logic. But in that case, we benefit from recent works on integration of separation logic in the Coq proof assistant and in particular the Iris project http://iris-project.org/.

Object polymorphism. Object-oriented programming has evolved since its foundation based on the representation of computations as an exchange of messages between objects. In modern programming languages like Scala, which aims at a synthesis between object-oriented and functional programming, object-orientation concretely results in the use of hierarchies of interfaces ordered by the subtyping relation and the definition of interface implementations that can interoperate. As observed by Cook and Aldrich [53, 35], interoperability can be considered as the essential feature of objects and is a requirement for many modern frameworks and ecosystems: it means that two different implementations of the same interface can interoperate.

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

For subtyping, the natural solution in type theory is coercive subtyping [88], as implemented in Coq, with an explicit operator for coercions. This should lead to a shallow embedding, but has limitations: indeed, while it allows subtyping to be faithfully represented, it does not provide a direct means to represent union and intersection types, which are often associated with subtyping (for instance intersection types are present in Scala). A more ambitious solution would be to resort to subsumptive subtyping (or semantic subtyping [61]): in its more general form, a type algebra is extended with boolean operations (union, intersection, complementing) to get a boolean algebra with operators (the original type

constructors). Subtyping is then interpreted as the natural partial order of the boolean algebra.

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

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

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

Completing and maintaining a large corpus of formalised mathematics requires a well-designed tactic language. This language should both accommodate the possible specific needs of the theories at stake, and help with diagnostics at refactoring time. Coq's tactic language is in fact two-leveled. First, it includes a basic tactic language, to organise the deductive steps in a proof script and to perform the elementary bureaucracy. Its second layer is a meta-programming language, which allows users to define their own new tactics at toplevel. Our first direction of work consists in the investigation of the appropriate features of the *basic tactic language*. For instance, the design of the Ssreflect tactic language, and its support for the small scale reflection methodology [67], has been a key ingredient in at least two large scale formalisation endeavours: the Four Colour Theorem [65] and of the Odd Order Theorem [66]. Building on our experience with the Ssreflect tactic language, we will contribute to the ongoing work on the basic tactic language for Coq. The second objective of this task is to contribute to the design of a *typed tactic language*. In particular, we will build on the work of Ziliani and his collaborators [115], extending it with reasoning about the effects that tactics have on the "state of a proof" (e.g. number of sub-goals, metavariables in context). We will also develop a novel approach for incremental type checking of proof scripts, so that programmers gain access to a richer discovery- engineering interaction with the proof assistant.

3.5   Practical experiments

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

3.5.1   Certified Code Refactoring

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

We are currently working on the formalisation of refactoring tools in Coq [50]. Automatic refactoring of programs in industrial languages is difficult because of the large number of potential interactions between language features that are difficult to predict and to test. Indeed, all available refactoring tools suffer from bugs : they fail to ensure that the generated program has the same behaviour as the input program. To cope with that difficulty, we have chosen to build a refactoring tool with Coq : a program transformation is written in the Coq programming language, then proven correct on all possible inputs, and then an OCaml executable program is generated by the platform. We rely on the CompCert C

formalisation of the C language. CompCert is currently the most complete formalisation of an industrial language, which justifies that choice. We have three goals in that project :

- Build a refactoring tool that programmers can rely on and make it available in a popular platform (such as Eclipse, IntelliJ or Frama-C).

- Explore large, drastic program transformations such as replacing a design architecture for an other one, by applying a sequence of small refactoring operations (as we have done for Java and Haskell programs before [49, 51, 34]), while ensuring behaviour preservation.

- Explore the use of enhancements of proof systems on large developments. For instance, refactoring tools are usually developed in the imperative/object paradigm, so the extension of Coq with side effects or with object features proposed in the team can find a direct use-case here.

3.5.2   Certified Constraint Programming

We plan to make use of the internalisation of the object-oriented paradigm in the context of constraint programming. Indeed, this domain is made of very complex algorithms that are often developed using object-oriented programming (as it is the case for instance for CHOCO, which is developed in the Tasc Group at IMT Atlantique, Nantes). We will in particular focus on filtering algorithms in constraint solvers, for which research publications currently propose new algorithms with manual proofs. Their formalisation in Coq is challenging. Another interesting part of constraint solving to formalise is the part that deals with program generation (as opposed to extraction). However, when there are numerous generated pieces of code, it is not realistic to prove their correctness manually, and it can be too difficult to prove the correctness of a generator. So we intend to explore a middle path that consists in generating a piece of code along with its corresponding proof (script or proof term). A target application could be interval constraints (for instance Allen interval algebra or region connection calculus) that can generate thousands of specialised filtering algorithms for a small number of variables [41].

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

3.5.3   Certified Symbolic Computation

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

Experiments have been run that link computer algebra systems with Coq [57, 47]. These bridges rely on the implementation of formal proof-producing core algorithms like normalisation procedures. Incidentally, they require non trivial maintenance work to survive the evolution of both systems. Other proof assistants like the Isabelle/HOL system make use of so-called reflection schemes: the proof assistant can produce code in an external programming language like SML, but also allows to import the values output by these extracted programs back inside the formal proofs. This feature extends the trusted base of code quite significantly but it has been used for major achievements like a certified symbolic/numeric ODE solver [75].

We would like to bring Coq closer to the efficiency and user-friendliness of computer algebra systems: for now it is difficult to use the Coq programming language so that certified implementations of computer algebra algorithms have the right, observable, complexity when they are executed inside Coq. We see the addition of effects to the proof assistant as an opportunity to ease these implementations, for instance by making use of caching mechanisms or of profiling facilities. Such enhancements should enable the

verification of computation-intensive mathematical proofs that are currently beyond reach, like the validation of Helfgott's proof of the weak Goldbach conjecture [71].

4   Application domains

Programming

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

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

Foundations of mathematics

- Formalisation of mathematics

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

5   Highlights of the year

- The Gallinette team members have published 4 papers at POPL'20.

- Assia Mahboubi defended her habilitation thesis [24], Jan 5th 2021.

- Assia Mahboubi got an ERC Considator Grant at the 2020 call.

6   New software and platforms

6.1   New software

6.1.1   Ltac2

Keywords: Coq, Proof assistant

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

Contact: Pierre-Marie Pédrot

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 8.11 to 8.13, 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 version of Equations is based on an improved simplification engine for the dependent equalities appearing during dependent eliminations that is also usable as a separate dependent elimination tactic, providing an axiom-free variant of dependent destruction and a more powerful form of inversion.

This is a bugfix release of version 1.2 working with Coq 8.11 to Coq 8.13 See https://mattam82.github.io/Coq-Equations/equations/2019/05/17/1.2.html for the 1.2 release notes. New in this version:

Fixed issue #297: dependent elimination simplification mistreated let-bindings Fixed issue #295: discrepancy between the syntax in the manual and the implementation Ensure that NoConfusion is derived before EqDec as it is necessary to solve the corresponding obligation.

News of the Year: Equations 1.2.3, released first in June 2020 brings bugfixes, a fixed grammar and more robust proof automation tactics.

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

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

Authors: Cyprien Mangin, Matthieu Sozeau, Matthieu Sozeau

Contact: Matthieu Sozeau

Participants: Matthieu Sozeau, Cyprien Mangin

6.1.3 Math-Components

Name: Mathematical Components library

Keyword: Proof assistant

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

Release Contributions: This releases is compatible with Coq 8.10, 8.11 and Coq 8.12 The main changes are: - support for Coq 8.7, 8.8 and 8.9 have been dropped, - a change of implementation of intervals and the updated theory, - the addition of kernel lemmas for matrices, - generalized many lemmas for path and sorted, - several lemma additions, name changes and bug fixes.

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

Contacts: Assia Mahboubi, Enrico Tassi, Georges Gonthier

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

6.1.4 math-comp-analysis

Name: Mathematical Components Analysis

Keyword: Proof assistant

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

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

News of the Year: In 2019, there were 3 releases.

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

Publication: hal-01719918

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

Partners: Ecole Polytechnique, AIST Tsukuba

6.1.5 MetaCoq

Keyword: Coq

Scientific Description: The MetaCoq project aims to provide a certified meta-programming environment in Coq. It builds on Template-Coq, a plugin for Coq originally implemented by Malecha (Extensible proof engineering in intensional type theory, Harvard University, 2014), which provided a reifier for Coq terms and global declarations, as represented in the Coq kernel, as well as a denotation command. Recently, it was used in the CertiCoq certified compiler project (Anand et al., in: CoqPL, Paris, France, 2017), as its front-end language, to derive parametricity properties (Anand and Morrisett, in: CoqPL'18, Los Angeles, CA, USA, 2018). However, the syntax lacked semantics, be it typing semantics or operational semantics, which should reflect, as formal specifications in Coq, the semantics of Coq 's type theory itself. The tool was also rather bare bones, providing only rudimentary quoting and unquoting commands. MetaCoq generalizes it to handle the entire polymorphic calculus of cumulative inductive constructions, as implemented by Coq, including the kernel's declaration structures for definitions and inductives, and implement a monad for general manipulation of Coq's logical environment. The MetaCoq framework allows Coq users

to define many kinds of general purpose plugins, whose correctness can be readily proved in the system itself, and that can be run efficiently after extraction. Examples of implemented plugins include a parametricity translation and a certified extraction to call-by-value lambda-calculus. The meta-theory of Coq itself is verified in MetaCoq along with verified conversion, type-checking and erasure procedures providing higly 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 version is a beta-release including a fully-functional reification and denotation support, and the verified type-checking and erasure procedures. The metatheory proofs are not entirely completed.

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

URL: https://metacoq.github.io

Publications: hal-02901011, hal-02380196, hal-02167423, hal-01809681

Contact: Matthieu Sozeau

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

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

6.1.6 Coq

Name: The Coq Proof Assistant

Keywords: Proof, Certification, Formalisation

Scientific Description: Coq is an interactive proof assistant based on the Calculus of (Co-)Inductive Constructions, extended with universe polymorphism. This type theory features inductive and co-inductive families, an impredicative sort and a hierarchy of predicative universes, making it a very expressive logic. The calculus allows to formalize both general mathematics and computer programs, ranging from theories of finite structures to abstract algebra and categories to programming language metatheory and compiler verification. Coq is organised as a (relatively small) kernel

including efficient conversion tests on which are built a set of higher-level layers: a powerful proof engine and unification algorithm, various tactics/decision procedures, a transactional document model and, at the very top an IDE.

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

Release Contributions: Some highlights from this release are:

- Introduction of primitive persistent arrays in the core language, implemented using imperative persistent arrays. - Introduction of definitional proof irrelevance for the equality type defined in the SProp sort. - Many improvements to the handling of notations, including number notations, recursive notations and notations with bindings. A new algorithm chooses the most precise notation available to print an expression, which might introduce changes in printing behavior.

See the Zenodo citation for more information on this release: https://zenodo.org/record/4501022#.YB00r5NKjlw

News of the Year: Coq version 8.13 integrates many usability improvements, as well as extensions of the core language. The main changes include: - Introduction of primitive persistent arrays in the core language, implemented using imperative persistent arrays. - Introduction of definitional proof irrelevance for the equality type defined in the SProp sort. - Cumulative record and inductive type declarations can now specify the variance of their universes. - Various bugfixes and uniformization of behavior with respect to the use of implicit arguments and the handling of existential variables in declarations, unification and tactics. - New warning for unused variables in catch-all match branches that match multiple distinct patterns. - New warning for Hint commands outside sections without a locality attribute, whose goal is to eventually remove the fragile default behavior of importing hints only when using Require. The recommended fix is to declare hints as export, instead of the current default global, meaning that they are imported through Require Import only, not Require. See the following rationale and guidelines for details. - General support for boolean attributes. - Many improvements to the handling of notations, including number notations, recursive notations and notations with bindings. A new algorithm chooses the most precise notation available to print an expression, which might introduce changes in printing behavior. - Tactic improvements in lia and its zify preprocessing step, now supporting reasoning on boolean operators such as Z.leb and supporting primitive integers Int63. - Typing flags can now be specified per-constant / inductive. - Improvements to the reference manual including updated syntax descriptions that match Coq's grammar in several chapters, and splitting parts of the tactics chapter to independent sections.

See the changelog for an overview of the new features and changes, along with the full list of contributors. https://coq.github.io/doc/v8.13/refman/changes.html#version-8-13

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

Authors: Bruno Barras, Yves Bertot, Frédéric Besson, Pierre Corbineau, Cristina Cornes, Judicaël Courant, Pierre Courtieu, Pierre Crégut, David Delahaye, Maxime Denes, Jean-Christophe Filliâtre, Julien Forest, Emilio Jesus Gallego Arias, Gaëtan Gilbert, Georges Gonthier, Benjamin Grégoire, Hugo Herbelin, Gérard Huet, Vincent Laporte, Pierre Letouzey, Assia Mahboubi, Pascal Manoury, Guillaume Melquiond, César Munoz, Chetan Murthy, Amokrane Saibi, Catherine Parent, Christine Paulin Mohring, Pierre-Marie Pédrot, Loïc Pottier, Matthieu Sozeau, Arnaud Spiwack, Enrico Tassi, Laurent Théry, Benjamin Werner, Théo Zimmermann

Contacts: Hugo Herbelin, Matthieu Sozeau

Participants: Yves Bertot, Frédéric Besson, Tej Chajed, Cyril Cohen, Pierre Corbineau, Pierre Courtieu, Maxime Denes, 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 Pédrot, ClÉment Pit-Claudel, Kazuhiko Sakaguchi, Vincent Semeria, Michael Soegtrop, Arnaud Spiwack, Matthieu Sozeau, Enrico Tassi, Laurent Théry, Anton Trunov, Li-Yao Xia, Théo Zimmermann

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

6.1.7 memprof-limits

Keyword: Library

Scientific Description: Memprof-limits is an implementation of per-thread global memory limits, and per-thread allocation limits à la Haskell, 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, thereby summarising practical knowledge acquired in OCaml by the Coq proof assistant as well as in other programming languages, to my knowledge told here for the first time.

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 a probabilistic implementation of per-thread global memory limits, and per-thread allocation limits, for the OCaml language.

Per-thread global memory limits let you bound the memory consumption of specific parts of your program, in terms of memory used by the whole program.

Per-thread allocation limits let you bound the execution of parts of the program measured in number of allocation, analogous to the same feature in Haskell. Allocation limits count allocations but not deallocations, and is therefore a measure of the work done which can be more suitable than execution time.

Release Contributions: Initial version.

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

Author: Guillaume Munch

Contact: Guillaume Munch

7 New results

7.1 Logical Foundations of Programming Languages

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

7.1.1 Reasoning about equivalence of programs

Automating Contextual Equivalence for Higher-Order Programs with References   In [12], we have proposed a framework to study contextual equivalence of programs written in a call-by-value functional language with local integer references. It reduces the problem of contextual equivalence to the problem of non-reachability in a transition system of memory configurations. This reduction is complete for recursion-free programs. Restricting to programs that do not allocate references inside the body of functions, we have encoded this non-reachability problem as a set of constrained Horn clauses that can then be checked for satisfiability automatically. Restricting furthermore to a language with finite data-types, we also get a new decidability result for contextual equivalence at any type.

7.1.2   Classical Logic

Dependent Type Theory in Polarised Sequent Calculus   Thanks to several works on classical logic in proof theory, it is now well-established that continuation-passing style (CPS) translations in call by name and call by value correspond to different polarisations of formulae. Extending this observation and building on Curien and Herbelin's abstract-machine-like calculi, a term assignment has been proposed for a polarised sequent calculus (where the polarities of formulae determine the evaluation order) in which various calculi from the literature can be obtained with macros responsible for the choices of polarities. It aims to explain several CPS translations from the literature by decomposing them through a single CPS for sequent calculus. It has later proved to be a fruitful setting to study the addition of effects and resource modalities, providing a categorical proof theory of Call By Push Value semantics. In [22], we propose to bring together a dependently-typed theory (ECC) and polarised sequent calculus, by presenting a calculus Ldep suitable as a vehicle for compilation and representation of effectful computations. As a first step in that direction, we show that Ldep advantageously factorizes a dependently typed continuation-passing style translation for ECC+call/cc. To avoid the inconsistency of type theory with control operators, we restrict their interaction. Nonetheless, in the pure case, we obtain an unrestricted translation from ECC to itself, thus opening the door to the definition of dependently typed compilation transformations.

Revisiting the duality of computation: an algebraic analysis of classical realizability models   In an impressive series of papers, Krivine showed at the edge of the last decade how classical realizability provides a surprising technique to build models for classical theories. In particular, he proved that classical realizability subsumes Cohen's forcing, and even more, gives rise to unexpected models of set theories. Pursuing the algebraic analysis of these models that was first undertaken by Streicher, Miquel recently proposed to lay the algebraic foundation of classical realizability and forcing within new structures which he called implicative algebras. These structures are a generalization of Boolean algebras based on an internal law representing the implication. Notably, implicative algebras allow for the adequate interpretation of both programs (i.e. proofs) and their types (i.e. formulas) in the same structure. The very definition of implicative algebras takes position on a presentation of logic through universal quantification and the implication and, computationally, relies on the call-by-name λ-calculus. In [19], we investigate the relevance of this choice, by introducing two similar structures. On the one hand, we define disjunctive algebras, which rely on internal laws for the negation and the disjunction and which we show to be particular cases of implicative algebras. On the other hand, we introduce conjunctive algebras, which rather put the focus on conjunctions and on the call-by-value evaluation strategy. We finally show how disjunctive and conjunctive algebras algebraically reflect the well-known duality of computation between call-by-name and call-by-value.

7.1.3   Syntax and Rewriting Systems

Reduction Monads and Their Signatures   In [8], we study reduction monads, which are essentially the same as monads relative to the free functor from sets into multigraphs. Reduction monads account for two aspects of the lambda calculus: on the one hand, in the monadic viewpoint, the lambda calculus is an object equipped with a well-behaved substitution; on the other hand, in the graphical viewpoint, it is an oriented multigraph whose vertices are terms and whose edges witness the reductions between two terms. We study presentations of reduction monads. To this end, we propose a notion of reduction signature. As usual, such a signature plays the role of a virtual presentation, and specifies arities for generating operations-possibly subject to equations-together with arities for generating reduction rules. For each such signature, we define a category of models; any model is, in particular, a reduction monad. If the initial object of this category of models exists, we call it the reduction monad presented (or specified) by the given reduction signature. Our main result identifies a class of reduction signatures which specify a reduction monad in the above sense. We show in the examples that our approach covers several standard variants of the lambda calculus.

Modules over monads and operational semantics   [18] is a contribution to the search for efficient and high-level mathematical tools to specify and reason about (abstract) programming languages or calculi. Generalising the reduction monads of Ahrens et al., we introduce operational monads, thus covering

new applications such as the-calculus, Positive GSOS specifications, and the big-step, simply-typed, call-by-value-calculus. Finally, we design a notion of signature for operational monads that covers all our examples.

Chiralities in topological vector spaces   Chiralities are categories introduced by Mellies to account for a game semantics point of view on negation. In [29], we uncover instances of this structure in the theory of topological vector spaces, thus constructing several new polarized models of Multiplicative Linear Logic. These models improve previously known smooth models of Differential Linear Logic, showing the relevance of chiralities to express topological properties of vector spaces. They are the first denotational polarized models of Multiplicative Linear Logic, based on the pre-existing theory of topological vector spaces, in which two distinct sets of formulas, two distinct negations, and two shifts appear naturally.

7.2   Models of programming languages mixing effects and resources

Resource management in OCaml   The current multicore OCaml implementation bans so-called "naked pointers", pointers to outside the heap owned by the OCaml garbage collector, unless these pointers follow drastic restrictions. A backwards-incompatible change has been proposed to make way for the new multicore GC in OCaml. Building on our investigations for the design of a resource-management extension for OCaml, we argued in [23] that out-of-heap pointers are not an anomaly, but are part of a better mixing of systems and functional programming, and we designed an alternative approach that would better preserve the backwards-compatiblity aims of OCaml multicore without affecting performances.

7.3   Type Theory and Proof Assistants

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

7.3.1   Type Theory

Effects in Type Theory.   There is a critical tension between substitution, dependent elimination and effects in type theory. In this paper, we crystallize this tension in the form of a no-go theorem that constitutes the fire triangle of type theory. To release this tension, we propose in [13] DCBPV, an extension of call-by-push-value (CBPV)-a general calculus of effects-to dependent types. Then, by extending to CBPV the well-known decompositions of call-by-name and call-by-value into CBPV, we show why, in presence of effects, dependent elimination must be restricted in call-by-name, and substitution must be restricted in call-by-value. To justify DCBPV and show that it is general enough to interpret many kinds of effects, we define various effectful syntactic translations from DCBPV to Martin-Löf type theory: the reader, weaning and forcing translations.

Gradualizing the Calculus of Inductive Constructions   Acknowledging the ordeal of a fully formal development in a proof assistant such as Coq, we have investigated in [27] gradual variations on the Calculus of Inductive Construction (CIC) for swifter prototyping with imprecise types and terms. We observe, with a no-go theorem, a crucial tradeoff between graduality and the key properties of canonicity, decidability and closure of universes under dependent product that CIC enjoys. Beyond this Fire Triangle of Graduality, we explore the gradualization of CIC with three different compromises, each relaxing one edge of the Fire Triangle. We develop a parametrized presentation of Gradual CIC that encompasses all three variations, and jointly develop their metatheory. We first present a bidirectional elaboration of Gradual CIC to a dependently-typed cast calculus, which elucidates the interrelation between typing, conversion, and graduality. We then establish the metatheory of this cast calculus through both a syntactic model into CIC, which provides weak canonicity, confluence, and when applicable, normalization, and a monotone model that purports the study of the graduality of two of the three variants. This work

informs and paves the way towards the development of malleable proof assistants and dependently-typed programming languages.

The folk model category structure on strict ω-categories is monoidal   In [9], we prove that the folk model category structure on the category of strict ω-categories, introduced by Lafont, Métayer and Worytkiewicz, is monoidal, first, for the Gray tensor product and, second, for the join of ω-categories, introduced by the first author and Maltsiniotis. We moreover show that the Gray tensor product induces, by adjunction, a tensor product of strict (m, n)-categories and that this tensor product is also compatible with the folk model category structure. In particular, we get a monoidal model category structure on the category of strict ω-groupoids. We prove that this monoidal model category structure satisfies the monoid axiom, so that the category of Gray monoids, studied by the second author, bears a natural model category structure.

Model structure on the universe of all types in interval type theory   Model categories constitute the major context for doing homotopy theory. More recently, Homotopy Type Theory has been introduced as a context for doing synthetic homotopy theory. In [10], we show that a slight generalization of Homotopy Type Theory, called Interval Type Theory, allows to define a model structure on the universe of all types, which, through the model interpretation, corresponds to defining a model structure on the category of cubical sets. This work generalizes previous works of Gambino, Garner and Lumsdaine from the universe of fibrant types to the universe of all types. Our definition of Interval Type Theory comes from the work of Orton and Pitts to define a syntactic approximation of the internal language of the category of cubical sets. We extend the work of Orton and Pitts by introducing the notion of degenerate fibrancy, which allows to define a fibrant replacement, at the heart of the model structure on the universe of all types. All our definitions and propositions have been formalized using the Coq proof assistant.

Types are Internal ∞-groupoids   An alternative to working with model structures is to pursue the idea of pushing synthetic Homotopy Theory further, so as to deal with higher coherences directly inside Type Theory. In [25] we show that, by extending univalent type theory with a universe of definitionally associative and unital polynomial monads, we arrive at a coinductive definition of opetopic type which is able to encode a number of fully coherent algebraic structures. In particular, our approach leads to a definition of ∞-groupoid internal to type theory and we prove that the type of such ∞-groupoids is equivalent to the universe of types. That is, every type admits the structure of an ∞-groupoid internally, and this structure is unique.

Russian Constructivism in a Prefascist Theory   In [21] we give two different results. First, we provide a purely syntactic presheaf model of CIC. Contrarily to similar endeavours, this variant both preserves conversion and interprets full dependent elimination. Then, using a particular instance of this model, we show how to extend CIC with Markov's principle, while preserving all good meta-theoretical properties like canonicity and decidability of type-checking. The resulting construction can be seen as a synthetic presentation of Coquand-Hofmann's syntactic model of PRAω + MP as the composition of Pédrot-Tabareau's exceptional model with our presheaf interpretation.

7.3.2   Proof Assistants

Metacoq   The MetaCoq project [14] aims to provide a certified meta-programming environment in Coq. It builds on Template-Coq, a plugin for Coq originally implemented by Malecha (2014), which provided a reifier for Coq terms and global declarations, as represented in the Coq kernel, as well as a denotation command. Recently, it was used in the CertiCoq certified compiler project (Anand et al., 2017), as its front-end language, to derive parametricity properties (Anand and Morrisett, 2018). However, the syntax lacked semantics, be it typing semantics or operational semantics, which should reflect, as formal specifications in Coq, the semantics of Coq's type theory itself. The tool was also rather bare bones, providing only rudimentary quoting and unquoting commands. We generalize it to handle the entire Polymorphic Calculus of Cumulative Inductive Constructions (pCUIC), as implemented by Coq, including the kernel's declaration structures for definitions and inductives, and implement a monad for

general manipulation of Coq's logical environment. We demonstrate how this setup allows Coq users to define many kinds of general purpose plugins, whose correctness can be readily proved in the system itself, and that can be run efficiently after extraction. We give a few examples of implemented plugins, including a parametricity translation and a certifying extraction to call-by-value λ-calculus. We also advocate the use of MetaCoq as a foundation for higher-level tools.

Verification of Type Checking and Erasure for Coq, in Coq   Coq is built around a well-delimited kernel that perfoms typechecking for definitions in a variant of the Calculus of Inductive Constructions (CIC). Although the metatheory of CIC is very stable and reliable, the correctness of its implementation in Coq is less clear. Indeed, implementing an efficient type checker for CIC is a rather complex task, and many parts of the code rely on implicit invariants which can easily be broken by further evolution of the code. Therefore, on average, one critical bug has been found every year in Coq. [15] presents the first implementation of a type checker for the kernel of Coq (without the module system and template polymorphism), which is proven correct in Coq with respect to its formal specification and axiomatisation of part of its metatheory. Note that because of Gödel's incompleteness theorem, there is no hope to prove completely the correctness of the specification of Coq inside Coq (in particular strong normalisation or canonicity), but it is possible to prove the correctness of the implementation assuming the correctness of the specification, thus moving from a trusted code base (TCB) to a trusted theory base (TTB) paradigm. Our work is based on the MetaCoq project which provides metaprogramming facilities to work with terms and declarations at the level of this kernel. Our type checker is based on the specification of the typing relation of the Polymorphic, Cumulative Calculus of Inductive Constructions (pCUIC) at the basis of Coq and the verification of a relatively efficient and sound type-checker for it. In addition to the kernel implementation, an essential feature of Coq is the so-called extraction: the production of executable code in functional languages from Coq definitions. We present a verified version of this subtle type-and-proof erasure step, therefore enabling the verified extraction of a safe type-checker for Coq.

A Type Theory with Computational Assumptions   Dependently typed programming languages and proof assistants such as Agda and Coq rely on computation to automatically simplify expressions during type checking. To overcome the lack of certain programming primitives or logical principles in those systems, it is common to appeal to axioms to postulate their existence. However, one can only postulate the bare existence of an axiom, not its computational behaviour. Instead, users are forced to postulate equality proofs and appeal to them explicitly to simplify expressions, making axioms dramatically more complicated to work with than built-in primitives. On the other hand, the equality reflection rule from extensional type theory solves these problems by collapsing computation and equality, at the cost of having no practical type checking algorithm. In [11], we introduce Rewriting Type Theory (RTT), a type theory where it is possible to add computational assumptions in the form of rewrite rules. Rewrite rules go beyond the computational capabilities of intensional type theory, but in contrast to extensional type theory, they are applied automatically so type checking does not require input from the user. To ensure type soundness of RTT-as well as effective type checking-we provide a framework where confluence of user-defined rewrite rules can be checked modularly and automatically, and where adding new rewrite rules is guaranteed to preserve subject reduction. The properties of RTT have been formally verified using the MetaCoq framework and an implementation of rewrite rules is already available in the Agda proof assistant.

7.4   Program Certifications and Formalisation of Mathematics

Cubical Synthetic Homotopy Theory   Homotopy type theory is an extension of type theory that enables synthetic reasoning about spaces and homotopy theory. This has led to elegant computer formalizations of multiple classical results from homotopy theory. However, many proofs are still surprisingly complicated to formalize. One reason for this is the axiomatic treatment of univalence and higher inductive types which complicates synthetic reasoning as many intermediate steps, that could hold simply by computation, require explicit arguments. Cubical type theory offers a solution to this in the form of a new type theory with native support for both univalence and higher inductive types. In [20], we show how the recent cubical extension of Agda can be used to formalize some of the major results of homotopy type theory in a direct and elegant manner.

Competing inheritance paths in dependent type theory: a case study in functional analysis   In [17], we discuss the design of a hierarchy of structures which combine linear algebra with concepts related to limits, like topology and norms, in dependent type theory. This hierarchy is the backbone of a new library of formalized classical analysis, for the Coq proof assistant. It extends the Mathematical Components library, geared towards algebra, with topics in analysis. Issues of a more general nature related to the inheritance of poorer structures from richer ones arise due to this combination. We present and discuss a solution, coined forgetful inheritance, based on packed classes and unification hints.

8   Bilateral contracts and grants with industry

CoqExtra

Title: A Formally Verified Extraction Mechanism using Precise Type Specifications

Duration: 2020 - 2022

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)

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 use this feature to expand the automation available to users of proof assistants. The main motivation here is thus to increase the class of goals that can be proved formally and automatically by the interactive proof assistant, rather than to work on the formal verification of specific albeit large decision problems. In this case, the central research problem is to bridge the gap between the rich specification language of the proof assistant, and the restricted fragment handled by the automated prover. This project will thus investigate the design, and the implementation, of the corresponding translation phase. This translation transforms a logical statement possibly featuring user-defined data structures and higher-order quantifications, into another statement, logically stronger, than can be sent to the automated prover. We thus aim at a triple objective: expressivity, extensibility and efficiency. This grant is funding the PhD of Enzo Crance.

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

OCaml-Evolution   The team received part of the Nomadic-Labs grant OCaml-Evolution (Coordinator and Inria contact: Gabriel scherer) for G. Munch-Maccagnoni's continued work on the OCaml compiler.

9   Partnerships and cooperations

9.1   International initiatives

9.1.1   Inria International Labs

GECO

Title: Gradual verification and robust proof Engineering for COq

Duration: 2018 - 2020

Coordinator: Nicolas Tabareau

Partners:

- Centrum Wiskunde & Informatica, Universidad de Chile (Chile)

Inria contact: Nicolas Tabareau

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

software engineering point of view. This collaboration project gathers the expertise of researchers from Chile (Inria Chile, Universidad de Chile, Universidad Católica de Valparaíso) and France (Inria Nantes, Inria Paris), in different areas that are crucial to develop the vision of certified software engineering.

Outcome: One of the main achievements is the development of a library that mixes parametricity and univalence (coming from homotopy type theory), in order to provide easier-to-use proof assistants by supporting seamless programming and proving modulo equivalences (ICFP'18, extended version accepted in Journal of the ACM). We have also proposed a new framework that is able to mix gradual typing and dependent types, allowing the definition of gradual proof assistants, in which it is possible to both program and prove theorems in a gradual way, that is by postponing part of definitions and proofs, without blocking the development process. We hope that this work is the first step toward a more massive use of proof assistants in the industry.

Informal international partners

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

9.2 International research visitors

9.2.1 Visits to international teams

- G. Munch-Maccagnoni visited E. Tanter and M. Toro (U. Chile) from Januray to March (visit ended by the pandemic crisis).

- N. Tabareau visited E. Tanter (U. Chile) from Januray to March (visit ended by the pandemic crisis).

9.3 European initiatives

9.3.1 FP7 & H2020 Projects

CoqHoTT

- Title: Coq for Homotopy Type Theory

- Program: H2020

- Type: ERC

- Duration: June 2015 - May 2021 (1 year extension because of COVID-19 crisis)

- Coordinator: Inria

- Inria contact: Nicolas TABAREAU

- Every year, software bugs cost hundreds of millions of euros to companies and administrations. Hence, software quality is a prevalent notion and interactive theorem provers based on type theory have shown their efficiency to prove correctness of important pieces of software like the C compiler of the CompCert project. One main interest of such theorem provers is the ability to extract directly the code from the proof. Unfortunately, their democratization suffers from a major drawback, the mismatch between equality in mathematics and in type theory. Thus, significant Coq developments have only been done by virtuosos playing with advanced concepts of computer science and mathematics. Recently, an extension of type theory with homotopical concepts such as univalence is gaining traction because it allows for the first time to marry together expected principles of equality. But the univalence principle has been treated so far as a new axiom which breaks one fundamental property of mechanized proofs: the ability to compute with programs that make use of this axiom. The main goal of the CoqHoTT project is to provide a new generation of proof assistants with a computational version of univalence and use them as a base

to implement effective logical model transformation so that the power of the internal logic of the proof assistant needed to prove the correctness of a program can be decided and changed at compile time—according to a trade-off between efficiency and logical expressivity. Our approach is based on a radically new compilation phase technique into a core type theory to modularize the difficulty of finding a decidable type checking algorithm for homotopy type theory. The impact of the CoqHoTT project will be very strong. Even if Coq is already a success, this project will promote it as a major proof assistant, for both computer scientists and mathematicians. CoqHoTT will become an essential tool for program certification and formalization of mathematics.

9.4   Regional initiatives

- **Vercoma** (Atlanstic 2020/Attractivity grant)

- Goal: Verified computer mathematics.

- Coordinator: A. Mahboubi.

- Duration: 08/2018 - 08/2021.

10   Dissemination

10.1   Promoting Scientific Activities

10.1.1   Scientific Events: Organisation

CASS Andes summer school

- Jan 6-10, 2020

- The Coq Andes Summer School (CASS) was a one-week immersive summer school on type theory in general, and on the Coq proof assistant in particular. CASS was open to advanced and motivated undergraduate and postgraduate students, as well as young academics and professionals.

- We received 94 applications to CASS. Considering the housing capacity of Santuario del Río (49 beds) and the number of speakers (7) and the local organizer, we were able to accept 41 participants. Only two had to cancel at the last minute, leaving us with a total of 39 student participants. Most participants are from Latin America (31), then Europe (6), and then North America (1) and Asia (1).

10.1.2   Scientific Events: Selection

Chair of Conference Program Committees

- A. Mahboubi has been co-chair of the workshop CoqPL'20, satellite of POPL 2020.

Member of the Conference Program Committees

- N. Tabareau has been a member of the external research committee of ICFP'19.

- P.-M. Pédrot has been a PC member of JFLA'20.

- A. Mahboubi has been a PC member of IJCAR'20 and CPP'20.

- M. Sozeau has been a PC member of POPL'21.

- G. Munch-Maccagnoni has been a PC member of IWACO'20.

Reviewer

- P.-M. Pédrot acted as a reviewer for PPDP'20 and a special Festschrift issue of the Journal of Functional Programming.

- A. Mahboubi acted as a reviewer for ISSAC'20 and LPAR'20.

- G. Munch-Maccagnoni acted as a reviewer for the post-proceedings of TYPES'20.

10.1.3   Journal

Member of the Editorial Boards

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

Reviewer - Reviewing Activities

- M. Sozeau acted as a reviewer for the Journal of Functional Programming.

- G. Munch-Maccagnoni acted as a reviewer for LMCS.

10.1.4   Invited Talks

- P.-M. Pédrot was planned to give an invited talk at TYPES'20 (cancelled at the last minute due to COVID outbreak).

- P.-M. Pédrot gave an invited talk at MSFP'20.

- P.-M. Pédrot gave a talk at the Birmingham Computer Science Seminar.

- A. Mahboubi gave an invited talk at Colloquium of Mathematics of the University of Ljubljana.

- A. Mahboubi gave a talk at the CRM - Computer Assisted Mathematical Proofs Seminar.

10.1.5   Leadership within the Scientific Community

- N. Tabareau is a member of the scientific committee of the GdR of Algebraic Topology.

- A. Mahboubi is a member of the core managment group of the EUTypes project, and leader of the "Tools" working group.

- A. Mahboubi is a member of the steering committee of the ITP conference.

- A. Mahboubi is a member of the steering committee of the CPP conference.

- A. Mahboubi is a member of the scientific committee of the GdR "Informatique Mathématique".

10.1.6   Scientific Expertise

G. Munch-Maccagnoni acted as an expert for Nomadic Labs on OCaml-Rust bindings.

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 softaware engineering (proof with Frama-C, test, code management), 20h, M1 (Engineering school), Polytech Nantes, France

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

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

- Master : Rémi Douence, Functional Programming with Haskell, 45h, M1 (engineers), IMT-Atlantique, Nantes, France

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

- Master : Rémi Douence, Formal Methods: Model checking with Alloy and from Haskell to Coq, 11h, M1 (apprenticeship), IMT-Atlantique, Nantes, France

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

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

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

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

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

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

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

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

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

- Master: Assia Mahboubi, Machine-Checked Mathematics, 22.5h, M2, Vrije Universiteit Amsterdam, the Netherlands

- Master : Matthieu Sozeau, Proof Assistants, 24h, M2 MPRI, Université Paris Diderot, France

- Master : Guillaume Munch-Maccagnoni, Initiation to safe systems programming with Rust, 7h, M1 (apprenticeship), IMT-Atlantique, Nantes, France

10.2.2   Supervision

- PhD defended on Sept 20: Théo Winterhalter, Formalisation and Meta-Theory of Type Theory, Univ Nantes, advisors: Matthieu Sozeau and Nicolas Tabareau

- PhD in progress: Xavier Montillet, Rewriting and solvability for Call-by-push-value, Univ Nantes, advisors: Guillaume Munch-Maccagnoni and Nicolas Tabareau

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

- PhD in progress: Loïc Pujet, Giving meaning to cubical type theory using forcing, Univ Nantes, advisors: Nicolas Tabareau

- PhD in progress: Meven Bertrand, Gradualizing the calculus of constructions, Univ Nantes, advisors: Nicolas Tabareau

- PhD in progress: Martin Baillon, Syntactic Models of Type Theory and Continuity Principles, Univ Nantes, advisors: Assia Mahboubi and Pierre-Marie Pédrot

- PhD in progress: Pierre Benjamin Giraud, Formalizing extraction of Coq to OCaml, Univ Nantes, advisors: Pierre-Marie Pédrot, Matthieu Sozeau and Nicolas Tabareau

- PhD in progress: Enzo Crance, Automated theorem proving and dependent types: automated reasoning for interactive proof assistants, Univ Nantes, advisors: Denis Cousineau and Assia Mahboubi

- PhD in progress: Antoine Allioux, Coherent Higher Structures in Homotopy Type Theory, Univ Paris Diderot, advisors: Pierre-Louis Curien (Univ. Paris Diderot), Eric Finster (Univ. Birmingham) and Matthieu Sozeau

Supervision of interns

- M2 intern, March–August 2020: Martin Baillon, Syntactic Models of Type Theory and Continuity Principles, advisors: Assia Mahboubi and Pierre-Marie Pédrot

- M2 intern, February–July 2020: Enzo Crance, Automated theorem proving and dependent types, advisors: Denis Cousineau and Assia Mahboubi

- M2 intern, October 2019–February 2020: Guillaume Combette, Axiomatic denotational semantics for resource management in systems programming, advisor: Guillaume Munch-Maccagnoni.

- L3 (ENS de Lyon) intern, June – July 2020: Théo Vignon, Banach - Steinhaus Theorem formalized in Coq, advisors: Marie Kerjean and Assia Mahboubi

10.2.3   Juries

- N. Tabareau has served as examiner for the PhD of Ambre Williams defended December 14th at Inria Paris.

- N. Tabareau has served as examiner for the PhD of Thibaut Benjamin defended November 5th at École Polythecnique / Institut Polytechnique de Paris.

- N. Tabareau has served as external member on the PhD jury of Houssem Hachmaoui defended October 16th at Université Paris Saclay.

- A. Mahboubi has served as member in the jury of the 2020 Gilles Kahn PhD prize.

- A. Mahboubi has served as external member on the PhD jury of Théo Winterhalter defended on September 18th at Nantes University.

10.3 Popularization

10.3.1 Internal or external Inria responsibilities

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

10.3.2 Education

- Hervé Grall has contributed to the project Merite, which aims to promote science learning in middle and high schools. He is the main contributor to the theme "Communication between machines". The project is coordinated by IMT-Atlantique in partnership with 7 other french higher education institutions, the rectorates of the Nantes and Rennes academies, and financed by the "Investments in the Future" and the fund FEDER Pays-de-la-Loire.

11 Scientific production

11.1 Major publications

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

[2] B. Ahrens, A. Hirschowitz, A. Lafont and M. Maggesi. 'Reduction Monads and Their Signatures'. In: *Proceedings of the ACM on Programming Languages* (Jan. 2020), pp. 1–29. DOI: 10.1145/3371099. URL: https://hal.inria.fr/hal-02380682.

[3] J. Cockx, N. Tabareau and T. Winterhalter. 'The Taming of the Rew: A Type Theory with Computational Assumptions'. In: *Proceedings of the ACM on Programming Languages*. POPL 2021 (2020). URL: https://hal.archives-ouvertes.fr/hal-02901011.

[4] G. Jaber. 'SyTeCi: Automating Contextual Equivalence for Higher-Order Programs with References'. In: *Proceedings of the ACM on Programming Languages* 28 (2020), pp. 1–28. DOI: 10.1145/33711 27. URL: https://hal.archives-ouvertes.fr/hal-02388621.

[5] P.-M. Pédrot. 'Russian Constructivism in a Prefascist Theory'. In: *LICS 2020 - Thirty-Fifth Annual ACM/IEEE Symposium on Logic in Computer Science*. Saarbrücken, Germany: IEEE, July 2020, pp. 1–14. DOI: 10.1145/3373718.3394740. URL: https://hal.inria.fr/hal-02548315.

[6] P.-M. Pédrot and N. Tabareau. 'The Fire Triangle'. In: *Proceedings of the ACM on Programming Languages* (Jan. 2020), pp. 1–28. DOI: 10.1145/3371126. URL: https://hal.archives-ouvertes.fr/hal-02383109.

[7] 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.

11.2 Publications of the year

International journals

[8] B. Ahrens, A. Hirschowitz, A. Lafont and M. Maggesi. 'Reduction Monads and Their Signatures'. In: *Proceedings of the ACM on Programming Languages* (19th Jan. 2020), pp. 1–29. DOI: 10.1145 /3371099. URL: https://hal.inria.fr/hal-02380682.

[9] D. Ara and M. Lucas. 'The folk model category structure on strict ω-categories is monoidal'. In: *Theory and Applications of Categories* 35.21 (20th May 2020), pp. 745–808. URL: https://hal.archives-ouvertes.fr/hal-02386617.

[10] S. Boulier and N. Tabareau. 'Model structure on the universe of all types in interval type theory'. In: *Mathematical Structures in Computer Science* (14th Oct. 2020), pp. 1–32. DOI: 10.1017/S0960 129520000213. URL: https://hal.inria.fr/hal-02966633.

[11] J. Cockx, N. Tabareau and T. Winterhalter. 'The Taming of the Rew: A Type Theory with Computational Assumptions'. In: *Proceedings of the ACM on Programming Languages*. POPL 2021 (2020). URL: https://hal.archives-ouvertes.fr/hal-02901011.

[12] 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: 10.1145/33711 27. URL: https://hal.archives-ouvertes.fr/hal-02388621.

[13] P.-M. Pédrot and N. Tabareau. 'The Fire Triangle: How to Mix Substitution, Dependent Elimination, and Effects'. In: *Proceedings of the ACM on Programming Languages* (22nd Jan. 2020), pp. 1–28. DOI: 10.1145/3371126. URL: https://hal.archives-ouvertes.fr/hal-02383109.

[14] M. Sozeau, A. Anand, S. Boulier, C. Cohen, Y. Forster, F. Kunze, G. Malecha, N. Tabareau and T. Winterhalter. 'The MetaCoq Project'. In: *Journal of Automated Reasoning* (Feb. 2020). DOI: 10.1007/s10817-019-09540-0. URL: https://hal.inria.fr/hal-02167423.

[15] 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* (19th Jan. 2020), pp. 1–28. DOI: 10.1145/3371076. URL: https://hal.archives-ouvertes.fr/hal-02380196.

[16] 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.

International peer-reviewed conferences

[17] 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, 29th June 2020, pp. 1–19. URL: https://hal.inria.fr/hal-02463336.

[18] A. Hirschowitz, T. Hirschowitz and A. Lafont. 'Modules over monads and operational semantics'. In: *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)*. 5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020). Vol. 167. Leibniz International Proceedings in Informatics (LIPIcs). Paris, France, 2020, 12:1–12:23. DOI: 10.4230/LIPIcs.FSCD.2020.12. URL: https://hal.archives-ouvertes.f r/hal-02338144.

[19] É. Miquey. 'Revisiting the duality of computation: an algebraic analysis of classical realizability models'. In: CSL 2020 - Conference on Computer Science Logic. Vol. 152. LIPIcs, CSL 2020. Barcelone, Spain, 13th Jan. 2020, pp. 1–52. DOI: 10.4230/LIPIcs.CSL.2020.30. URL: https://hal.archives-ouvertes.fr/hal-02305560.

[20] A. Mörtberg and L. Pujet. 'Cubical Synthetic Homotopy Theory'. In: CPP 2020 - 9th ACM SIGPLAN International Conference on Certified Programs and Proofs. New Orleans, United States: https://popl20.sigplan.org/home/CPP-2020, 20th Jan. 2020, pp. 1–14. DOI: 10.1145/3372885.3 373825. URL: https://hal.archives-ouvertes.fr/hal-02394145.

[21] 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, Nov. 2020, pp. 1–14. DOI: 10.1145/3373718.3394740. URL: https://hal.inria.fr/hal-02548315.

Conferences without proceedings

[22] É. Miquey, X. Montillet and G. Munch-Maccagnoni. 'Dependent Type Theory in Polarised Sequent Calculus (abstract)'. In: TYPES 2020 - 26th International Conference on Types for Proofs and Programs. Torino, Italy: https://types2020.di.unito.it/, 2nd Mar. 2020, pp. 1–3. URL: https://hal.inria.fr/hal-02505671.

[23]   G. Munch-Maccagnoni. 'Towards better systems programming in OCaml with out-of-heap alloca-
       tion'. In: ML Workshop. Jersey City, United States, 27th Aug. 2020. URL: https://hal.inria.fr
       /hal-03142386.

Doctoral dissertations and habilitation theses

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

Reports & preprints

[25]   A. Allioux, E. Finster and M. Sozeau. *Types are internal infinity-groupoids*. 25th Jan. 2021. URL:
       https://hal.inria.fr/hal-03133144.

[26]   S. Bernard, C. Cohen, A. Mahboubi and P.-Y. Strub. *Unsolvability of the Quintic Formalized in
       Dependent Type Theory*. 9th Feb. 2021. URL: https://hal.inria.fr/hal-03136002.

[27]   M. Bertrand, K. Maillard, N. Tabareau and É. Tanter. *Gradualizing the Calculus of Inductive
       Constructions*. 20th Nov. 2020. URL: https://hal.archives-ouvertes.fr/hal-02896776.

[28]   G. Jaber and A. S. Murawski. *Complete trace models of state and control*. 20th Jan. 2021. URL:
       https://hal.archives-ouvertes.fr/hal-03116698.

[29]   M. Kerjean. *Chiralities in topological vector spaces*. 13th Feb. 2020. DOI: 10.4230/LIPIcs. URL:
       https://hal.inria.fr/hal-02334917.

[30]   M. Lennon-Bertrand. *Complete Bidirectional Typing for the Calculus of Inductive Constructions*.
       12th Feb. 2021. URL: https://hal.archives-ouvertes.fr/hal-03139924.

11.3   Cited publications

[31]   A. Abel and T. Coquand. 'Untyped Algorithmic Equality for Martin-Löf's Logical Framework
       with Surjective Pairs'. English. In: *Typed Lambda Calculi and Applications*. Ed. by P. Urzyczyn.
       Vol. 3461. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, pp. 23–38. DOI:
       10.1007/11417170_4. URL: http://dx.doi.org/10.1007/11417170_4.

[32]   B. Accattoli and G. Guerrieri. 'Open Call-by-Value'. In: *Programming Languages and Systems*
       (1st Jan. 2016). DOI: 10.1007/978-3-319-47958-3_12. URL: http://dx.doi.org/10.1007
       /978-3-319-47958-3_12.

[33]   D. Ahman, N. Ghani and G. D. Plotkin. 'Dependent Types and Fibred Computational Effects'. In:
       Proc. FoSSaCS. 2015.

[34]   A. Ajouli, J. Cohen and J.-C. Royer. 'Transformations between Composite and Visitor Implementa-
       tions in Java'. In: *Software Engineering and Advanced Applications (SEAA), 2013 39th EUROMICRO
       Conference on*. Sept. 2013, pp. 25–32. DOI: 10.1109/SEAA.2013.53. URL: http://dx.doi.org
       /10.1109/SEAA.2013.53.

[35]   J. Aldrich. 'The power of interoperability: why objects are inevitable'. In: *ACM Symposium on New
       Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis,
       IN, USA, October 26-31, 2013*. Ed. by A. L. Hosking, P. T. Eugster and R. Hirschfeld. ACM, 2013,
       pp. 101–116.

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

[37]   J.-M. Andreoli. 'Logic Programming with Focusing Proof in Linear Logic'. In: *Journal of Logic and
       Computation* 2.3 (1992), pp. 297–347.

[38]   E. Arafailova. 'Functional description of sequence constraints and synthesis of combinatorial
       objects'. Theses. Ecole nationale supérieure Mines-Télécom Atlantique, Sept. 2018. URL: https:
       //tel.archives-ouvertes.fr/tel-01962957.

[39]   E. Arafailova, N. Beldiceanu, R. Douence, M. Carlsson, P. Flener, M. A. F. Rodríguez, J. Pearson and H. Simonis. 'Global Constraint Catalog, Volume II, Time-Series Constraints'. In: *CoRR* abs/1609.08925 (2016). URL: http://arxiv.org/abs/1609.08925.

[40]   E. Arafailova, N. Beldiceanu, R. Douence, P. Flener, M. A. F. Rodríguez, J. Pearson and H. Simonis. 'Time-Series Constraints: Improvements and Application in CP and MIP Contexts'. In: *Integration of AI and OR Techniques in Constraint Programming - 13th International Conference, CPAIOR 2016, Banff, AB, Canada, May 29 - June 1, 2016, Proceedings.* Ed. by C.-G. Quimper. Vol. 9676. Lecture Notes in Computer Science. Springer, 2016, pp. 18–34. DOI: 10.1007/978-3-319-33954-2_2. URL: https://doi.org/10.1007/978-3-319-33954-2_2.

[41]   A. F. Barco, J.-G. Fages, É. Vareilles, M. Aldanondo and P. Gaborit. 'Open Packing for Facade-Layout Synthesis Under a General Purpose Solver'. In: *Principles and Practice of Constraint Programming - 21st International Conference, CP 2015, Cork, Ireland, August 31 - September 4, 2015, Proceedings.* Ed. by G. Pesant. Vol. 9255. Lecture Notes in Computer Science. Springer, 2015, pp. 508–523. DOI: 10.1007/978-3-319-23219-5_36. URL: http://dx.doi.org/10.1007/978-3-319-23219-5%5C_36.

[42]   N. Beldiceanu, M. Carlsson, R. Douence and H. Simonis. 'Using finite transducers for describing and synthesising structural time-series constraints'. In: *Constraints* 21.1 (2016), pp. 22–40. DOI: 10.1007/s10601-015-9200-3. URL: http://dx.doi.org/10.1007/s10601-015-9200-3.

[43]   S. Berardi, M. Bezem and T. Coquand. 'On the computational content of the axiom of choice'. In: *The Journal of Symbolic Logic* 63.02 (1998), pp. 600–622.

[44]   M. Bezem, T. Coquand and S. Huber. 'A model of type theory in cubical sets'. In: *Preprint, September* (2013).

[45]   S. Boulier, P.-M. Pédrot and N. Tabareau. 'The next 700 syntactical models of type theory'. In: *Certified Programs and Proofs (CPP 2017).* Paris, France, Jan. 2017, pp. 182–194. DOI: 10.1145/3018610.3018620. URL: https://hal.inria.fr/hal-01445835.

[46]   E. Brady. 'Idris, a general-purpose dependently typed programming language: Design and implementation'. In: *J. Funct. Program.* 23.5 (2013), pp. 552–593. DOI: 10.1017/S095679681300018X. URL: https://doi.org/10.1017/S095679681300018X.

[47]   F. Chyzak, A. Mahboubi, T. Sibut-Pinote and E. Tassi. 'A Computer-Algebra-Based Formal Proof of the Irrationality of $\zeta(3)$'. In: *Interactive Theorem Proving.* Ed. by R. G. Gerwin Klein. Vol. 8558. Lecture Notes in Computer Science. Springer, 2014.

[48]   C. Cohen, T. Coquand, S. Huber and A. Mörtberg. *Cubical Type Theory: a constructive interpretation of the univalence axiom.* To appear in post-proceedings of Types for Proofs and Programs (TYPES 2015). 2016.

[49]   J. Cohen, R. Douence and A. Ajouli. 'Invertible Program Restructurings for Continuing Modular Maintenance'. In: *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on.* Mar. 2012, pp. 347–352. DOI: 10.1109/CSMR.2012.42. URL: http://dx.doi.org/10.1109/CSMR.2012.42.

[50]   J. Cohen. 'Renaming Global Variables in C Mechanically Proved Correct'. In: *Proceedings of the Fourth International Workshop on Verification and Program Transformation, Eindhoven, The Netherlands, 2nd April 2016.* Ed. by G. Hamilton, A. Lisitsa and A. P. Nemytykh. Vol. 216. Electronic Proceedings in Theoretical Computer Science. Open Publishing Association, 2016, pp. 50–64. DOI: 10.4204/EPTCS.216.3. URL: http://dx.doi.org/10.4204/EPTCS.216.3.

[51]   J. Cohen and A. Ajouli. 'Practical Use of Static Composition of Refactoring Operations'. In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing.* SAC '13. Coimbra, Portugal: ACM, 2013, pp. 1700–1705. DOI: 10.1145/2480362.2480684. URL: http://dx.doi.org/10.1145/2480362.2480684.

[52]   P. Cohen and M. Davis. *Set theory and the continuum hypothesis.* WA Benjamin New York, 1966.

[53] W. R. Cook. 'On understanding data abstraction, revisited'. In: *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*. Ed. by S. Arora and G. T. Leavens. ACM, 2009, pp. 557–572. DOI: 10.1145/1640089.1640133. URL: http://doi.acm.org/10.1145/16 40089.1640133.

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

[55] P.-L. Curien, M. Fiore and G. Munch-Maccagnoni. 'A Theory of Effects and Resources: Adjunction Models and Polarised Calculi'. In: *Proc. POPL*. 2016. DOI: 10.1145/2837614.2837652. URL: http://dx.doi.org/10.1145/2837614.2837652.

[56] P.-L. Curien and H. Herbelin. 'The duality of computation'. In: *ACM SIGPLAN Notices* 35 (2000), pp. 233–243.

[57] D. Delahaye and M. Mayero. 'Dealing with algebraic expressions over a field in Coq using Maple'. In: *J. Symbolic Comput.* 39.5 (2005). Special issue on the integration of automated reasoning and computer algebra systems, pp. 569–592. DOI: 10.1016/j.jsc.2004.12.004. URL: http://dx .doi.org/10.1016/j.jsc.2004.12.004.

[58] É. Tanter and N. Tabareau. 'Gradual Certified Programming in Coq'. In: *Proceedings of the 11th ACM Dynamic Languages Symposium (DLS 2015)*. Pittsburgh, PA, USA: ACM Press, Oct. 2015, pp. 26–40.

[59] R. Douence, X. Lorca and N. Loriant. 'Lazy Composition of Representations in Java'. In: *Software Composition, 8th International Conference, SC 2009, Zurich, Switzerland, July 2-3, 2009. Proceedings*. Ed. by A. Bergel and J. Fabry. Vol. 5634. Lecture Notes in Computer Science. Springer, 2009, pp. 55–71. DOI: 10.1007/978-3-642-02655-3_6. URL: https://doi.org/10.1007/978-3-642-02655-3_6.

[60] T. Ehrhard. 'Call-by-push-value from a linear logic point of view'. In: *European Symposium on Programming Languages and Systems*. Springer. 2016, pp. 202–228.

[61] A. Frisch, G. Castagna and V. Benzaken. 'Semantic Subtyping: Dealing Set-theoretically with Function, Union, Intersection, and Negation Types'. In: *J. ACM* 55.4 (Sept. 2008), 19:1–19:64.

[62] C. Führmann. 'Direct Models for the Computational Lambda Calculus'. In: *Electr. Notes Theor. Comput. Sci.* 20 (1999), pp. 245–292.

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

[64] J.-Y. Girard, A. Scedrov and P. J. Scott. 'Normal Forms and Cut-Free Proofs as Natural Transformations'. In: *in : Logic From Computer Science, Mathematical Science Research Institute Publications 21*. Springer-Verlag, 1992, pp. 217–241.

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

[66] G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. Roux, A. Mahboubi, R. O'Connor, S. Ould Biha, I. Pasca, L. Rideau, A. Solovyev, E. Tassi and L. Théry. 'A Machine-Checked Proof of the Odd Order Theorem'. In: *Interactive Theorem Proving*. Ed. by S. Blazy, C. Paulin-Mohring and D. Pichardie. Vol. 7998. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 163–179. DOI: 10.1007/978-3-642-39634-2_14. URL: http://dx.doi.org/10.1007/9 78-3-642-39634-2_14.

[67] G. Gonthier, A. Mahboubi and E. Tassi. *A Small Scale Reflection Extension for the Coq system*. Research Report RR-6455. *The Reference Manual of the Ssreflect extension to the Coq tactic language, available at http://hal.inria.fr/inria-00258384*. INRIA, 2008.

[68] T. G. Griffin. 'A Formulae-as-Types Notion of Control'. In: *Seventeenth Annual ACM Symposium on Principles of Programming Languages*. ACM Press, 1990, pp. 47–58.

[69] Y.-G. Guéhéneuc, R. Douence and N. Jussien. 'No Java without Caffeine: A Tool for Dynamic Analysis of Java Programs'. In: *17th IEEE International Conference on Automated Software Engineering (ASE 2002), 23-27 September 2002, Edinburgh, Scotland, UK*. IEEE Computer Society, 2002, p. 117. DOI: 10.1109/ASE.2002.1115000. URL: https://doi.org/10.1109/ASE.2002.1115000.

[70]   T. C. Hales, M. Adams, G. Bauer, D. T. Dang, J. Harrison, T. L. Hoang, C. Kaliszyk, V. Magron, S. McLaughlin, T. T. Nguyen, T. Q. Nguyen, T. Nipkow, S. Obua, J. Pleso, J. Rute, A. Solovyev, A. H. T. Ta, T. N. Tran, D. T. Trieu, J. Urban, K. K. Vu and R. Zumkeller. 'A formal proof of the Kepler conjecture'. In: *CoRR* abs/1501.02155 (2015). URL: http://arxiv.org/abs/1501.02155.

[71]   H. A. Helfgott. 'The ternary Goldbach conjecture is true'. In: *ArXiv e-prints* (Dec. 2013). arXiv: 1312.7748 [math.NT].

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

[73]   H. Herbelin and É. Miquey. 'Toward dependent choice: a classical sequent calculus with dependent types'. In: *TYPES 2015*. 2015.

[74]   T. Hirschowitz. 'Cartesian closed 2-categories and permutation equivalence in higher-order rewriting'. In: *Logical Methods in Computer Science* 9.3 (2013). 19 pages, p. 10. DOI: 10.2168/LMCS-9(3:10)2013. URL: https://hal.archives-ouvertes.fr/hal-00540205.

[75]   F. Immler. 'Verified Reachability Analysis of Continuous Systems'. In: *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. Ed. by C. Baier and C. Tinelli. Vol. 9035. Lecture Notes in Computer Science. Springer, 2015, pp. 37–51.

[76]   G. Jaber, G. Lewertowski, P.-M. Pédrot, M. Sozeau and N. Tabareau. 'The Definitional Side of the Forcing'. In: *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*. 2016, pp. 367–376.

[77]   G. Jaber, N. Tabareau and M. Sozeau. 'Extending type theory with forcing'. In: *Logic in Computer Science (LICS), 2012*. IEEE. 2012, pp. 395–404.

[78]   C. B. Jay and N. Ghani. 'The Virtues of Eta-Expansion'. In: *J. Funct. Program.* 5.2 (1995), pp. 135–154.

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

[80]   J.-L. Krivine. 'Realizability algebras II : new models of ZF + DC'. In: *Logical Methods in Computer Science* 8.1 (2012).

[81]   J. Lambek and P. J. Scott. *Introduction to higher order categorical logic*. New York, NY, USA: Cambridge University Press, 1986.

[82]   S. M. Lane and I. Moerdijk. *Sheaves in Geometry and Logic*. Springer-Verlag, 1992.

[83]   R. Lepigre. 'A classical realizability model for a semantic value restriction'. In: *European Symposium on Programming Languages and Systems*. Springer. 2016, pp. 476–502.

[84]   X. Leroy. 'Formal certification of a compiler back-end or: programming a compiler with a proof assistant'. In: *ACM SIGPLAN Notices* 41.1 (2006), pp. 42–54.

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

[86]   P. B. Levy. 'Contextual isomorphisms'. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM. 2017, pp. 400–414.

[87]   C. Liang and D. Miller. 'Focusing and polarization in linear, intuitionistic, and classical logics'. In: *Theor. Comput. Sci.* 410.46 (2009), pp. 4747–4768.

[88]   Z. Luo, S. Soloviev and T. Xue. 'Coercive subtyping: Theory and implementation'. In: *Inf. Comput.* 223 (2013), pp. 18–42.

[89]   J. Lurie. *Higher topos theory*. Annals of mathematics studies. Princeton, N.J., Oxford: Princeton University Press, 2009.

[90]   S. Mac Lane. 'Natural associativity and commutativity'. In: *Selected Papers* (1979), pp. 415–433.

[91]   P. Martin-Löf. 'An intuitionistic theory of types: predicative part'. In: *Logic Colloquium '73* Studies in Logic and the Foundations of Mathematics.80 (1975), pp. 73–118.

[92]   P.-A. Melliès. 'Asynchronous Games 3 An Innocent Model of Linear Logic'. In: *Electr. Notes Theor. Comput. Sci.* 122 (2005), pp. 171–192.

[93]   P.-A. Melliès. 'Asynchronous Games 4: A Fully Complete Model of Propositional Linear Logic'. In: *LICS.* 2005, pp. 386–395.

[94]   P.-A. Melliès and N. Tabareau. 'Resource modalities in tensor logic'. In: *Ann. Pure Appl. Logic* 161.5 (2010), pp. 632–653.

[95]   É. Miquey. 'A classical sequent calculus with dependent types'. In: *European Symposium on Programming.* Springer. 2017, pp. 777–803.

[96]   E. Moggi. 'Computational lambda-calculus and monads'. In: *Proceedings of the Fourth Annual IEEE Symposium on Logic in Computer Science (LICS 1989).* Pacific Grove, CA, USA: IEEE Computer Society Press, June 1989, pp. 14–23.

[97]   E. Moggi. 'Notions of computation and monads'. In: *Inf. Comput.* 93.1 (July 1991), pp. 55–92. DOI: 10.1016/0890-5401(91)90052-4. URL: http://dx.doi.org/10.1016/0890-5401(91)900 52-4.

[98]   G. Munch-Maccagnoni. 'Formulae-as-Types for an Involutive Negation'. In: *Proceedings of the joint meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (CSL-LICS).* 2014.

[99]   G. Munch-Maccagnoni. 'Models of a Non-Associative Composition'. In: *Proc. FoSSaCS.* Ed. by A. Muscholl. Vol. 8412. LNCS. Springer, 2014, pp. 397–412.

[100]   G. Munch-Maccagnoni. 'Note on Curry's style for Linear Call-by-Push-Value'. Manuscript. 3rd May 2017. URL: https://hal.inria.fr/hal-01528857.

[101]   A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau and L. Birkedal. *Ynot: Reasoning with the awkward squad.* 2008.

[102]   C. Okasaki. *Purely functional data structures.* Cambridge University Press, 1999.

[103]   P.-M. Pédrot and N. Tabareau. 'An Effectful Way to Eliminate Addiction to Dependence'. Jan. 2017. URL: https://hal.inria.fr/hal-01441829.

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

[105]   C. Prud'homme, X. Lorca, R. Douence and N. Jussien. 'Propagation engine prototyping with a domain specific language'. In: *Constraints* 19.1 (2014), pp. 57–76. DOI: 10.1007/s10601-013-91 51-5. URL: https://doi.org/10.1007/s10601-013-9151-5.

[106]   K. Quirin and N. Tabareau. 'Lawvere-Tierney sheafification in Homotopy Type Theory'. In: *Journal of Formalized Reasoning* 9.2 (2016). DOI: 10.6092/issn.1972-5787/6232. URL: https://hal .inria.fr/hal-01451710.

[107]   F. van Raamsdonk. 'Higher-order Rewriting'. In: *Proc. Rewrit. Tech. App.* Vol. 1631. LNCS. Springer, 1999, pp. 220–239.

[108]   J. C. Reynolds. 'Types, Abstraction and Parametric Polymorphism'. In: *IFIP Congress.* 1983, pp. 513–523.

[109]   P. Selinger. 'Control Categories and Duality: On the Categorical Semantics of the Lambda-Mu Calculus'. In: *Math. Struct in Comp. Sci.* 11.2 (2001), pp. 207–260.

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

[111]   K. Støvring. 'Extending the Extensional Lambda Calculus with Surjective Pairing is Conservative'. In: *Logical Methods in Computer Science* 2.2 (2006).

[112]   N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoue and S. Zanella-Béguelin. 'Dependent Types and Multi-Monadic Effects in F*'. In: *43nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, Jan. 2016, pp. 256–270. URL: https://www.fstar-lang.org/papers/mumon/.

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

[114]   M. Vákár. 'A Framework for Dependent Types and Effects'. In: *arXiv preprint arXiv:1512.08009* (2015).

[115]   B. Ziliani, D. Dreyer, N. R. Krishnaswami, A. Nanevski and V. Vafeiadis. 'Mtac: A monad for typed tactic programming in Coq'. In: *Journal of Functional Programming* 25 (2015). DOI: 10.1017/S0956796815000118. URL: http://dx.doi.org/10.1017/S0956796815000118.