

2025 Activity Report

RESEARCH CENTRE: Inria Paris Centre

IN PARTNERSHIP WITH: Collège de France

Project-Team

CAMBIUM

Programming languages: type systems, concurrency,
proofs of programs



Project-Team CAMBIUM

Creation of the Project-Team: 2019 August 01

Each year, Inria research teams publish an Activity Report presenting their work and results over the reporting period. These reports follow a common structure, with some optional sections depending on the specific team. They typically begin by outlining the overall objectives and research programme, including the main research themes, goals, and methodological approaches. They also describe the application domains targeted by the team, highlighting the scientific or societal contexts in which their work is situated. The reports then present the highlights of the year, covering major scientific achievements, software developments, or teaching contributions. When relevant, they include sections on software, platforms, and open data, detailing the tools developed and how they are shared. A substantial part is dedicated to new results, where scientific contributions are described in detail, often with subsections specifying participants and associated keywords. Finally, the Activity Report addresses funding, contracts, partnerships, and collaborations at various levels, from industrial agreements to international cooperations. It also covers dissemination and teaching activities, such as participation in scientific events, outreach, and supervision. The document concludes with a presentation of scientific production, including major publications and those produced during the year.

Keywords

Computer sciences and digital sciences

- A1.1.1. – Multicore, Manycore
- A1.1.3. – Memory models
- A2.1. – Programming Languages
 - A2.1.1. – Semantics of programming languages
 - A2.1.2. – Imperative programming
 - A2.1.3. – Object-oriented programming
 - A2.1.4. – Functional programming
 - A2.1.6. – Concurrent programming
 - A2.1.11. – Proof languages
- A2.2. – Compilation
 - A2.2.1. – Static analysis
 - A2.2.2. – Memory models
 - A2.2.4. – Parallel architectures
 - A2.2.5. – Run-time systems
- A2.5.4. – Software Maintenance & Evolution
- A4.5. – Formal method for verification, reliability, certification
 - A4.5.1. – Static analysis
 - A4.5.3. – Program proof
 - A4.5.4. – Proof of Theorems
- A7.2. – Logic in Computer Science
 - A7.2.2. – Automated Theorem Proving
 - A7.2.3. – Interactive Theorem Proving
 - A7.2.4. – Mechanized Formalization of Mathematics
- A7.3. – Calculability and computability
 - A7.3.1. – Computational models and calculability
 - A7.3.2. – Computability

Other research topics and application domains

- B5.2.3. – Aviation
- B6.1. – Software industry
- B6.6. – Embedded systems
- B9.5.1. – Computer science

Contents

| | |
|--|-----------|
| Project-Team CAMBIUM | 1 |
| 1 Team members, visitors, external collaborators | 5 |
| 2 Overall objectives | 6 |
| 2.1 Software reliability and reusability | 6 |
| 2.2 Qualities of a programming language | 7 |
| 2.3 Design, implementation, and evolution of OCaml | 7 |
| 2.4 Software verification | 8 |
| 2.5 Shared-memory concurrency | 9 |
| 3 Research program | 10 |
| 4 Application domains | 10 |
| 4.1 Formal methods | 10 |
| 4.2 High-assurance software | 11 |
| 4.3 Design and test of microprocessors | 11 |
| 4.4 Teaching programming | 11 |
| 5 Latest software developments, platforms, open data | 11 |
| 5.1 Latest software developments | 11 |
| 5.1.1 OCaml | 11 |
| 5.1.2 CompCert | 12 |
| 5.1.3 Diy | 12 |
| 5.1.4 Menhir | 12 |
| 5.1.5 CFML | 13 |
| 5.1.6 TLAPS | 13 |
| 5.1.7 ZENON | 13 |
| 5.1.8 hevea | 14 |
| 6 New results | 14 |
| 6.1 Long-term software projects | 14 |
| 6.1.1 The CompCert formally-verified compiler | 14 |
| 6.1.2 The OCaml system | 14 |
| 6.1.3 The diy tool suite | 15 |
| 6.1.4 TLA+ | 16 |
| 6.1.5 Menhir | 16 |
| 6.2 Programming language design and implementation | 16 |
| 6.2.1 Implementing and formalizing Modular Explicit | 16 |
| 6.2.2 Designing and formalizing Modular Implicit | 16 |
| 6.2.3 Omnidirectional type inference | 17 |
| 6.2.4 Formalization of recursive modules | 17 |
| 6.2.5 Tail Modulo Async-Await | 17 |
| 6.2.6 Mapping and explaining syntax errors with LRgrep | 18 |
| 6.3 Semantics | 18 |
| 6.3.1 Choice Trees: reasoning about nondeterministic, recursive, impure programs in Rocq | 18 |
| 6.3.2 Executable semantics for a concurrent subset of LLVM IR | 18 |
| 6.3.3 Layers of confluence for actors | 18 |
| 6.3.4 An abstract, certified account of operational game semantics | 19 |
| 6.3.5 Structural temporal logic for mechanized program verification | 19 |
| 6.3.6 MTrees: mixing monadic effects and labeled transition systems in Rocq | 19 |
| 6.3.7 Axiomatic memory models and virtual memory | 19 |
| 6.3.8 Semantics and sound implementation of AArch64 instructions | 20 |
| 6.4 Mechanized mathematics and program verification in Rocq | 20 |

| | | |
|-----------|--|-----------|
| 6.4.1 | Constructive reverse mathematics | 20 |
| 6.4.2 | Synthetic computability theory | 20 |
| 6.4.3 | A lazy, concurrent convertibility checker | 21 |
| 6.4.4 | Meta-programming in Rocq | 21 |
| 6.4.5 | The MetaRocq project | 21 |
| 6.4.6 | Code generation from proof assistants | 22 |
| 6.4.7 | Implementing and verifying Kaplan and Tarjan's catenable dequeues | 22 |
| 6.5 | Program verification in separation logic | 23 |
| 6.5.1 | Implementing and verifying Kaplan, Okasaki, and Tarjan's simple catenable dequeues | 23 |
| 6.5.2 | Verification of concurrent OCaml 5 programs in separation logic | 23 |
| 6.5.3 | A verified parallel scheduler for OCaml 5 | 23 |
| 6.5.4 | Osiris: formal semantics and reasoning rules for OCaml | 24 |
| 6.5.5 | Gospel: a formal specification language for OCaml | 24 |
| 6.5.6 | Verification of parallel real-time programs | 24 |
| 6.5.7 | Separation logic for one-shot undelimited continuations | 24 |
| 7 | Bilateral contracts and grants with industry | 25 |
| 7.1 | Bilateral grants with industry | 25 |
| 7.1.1 | The Caml Consortium | 25 |
| 7.1.2 | The OCaml Software Foundation | 25 |
| 8 | Partnerships and cooperations | 26 |
| 8.1 | National initiatives | 26 |
| 9 | Dissemination | 26 |
| 9.1 | Promotion of scientific activities | 26 |
| 9.1.1 | Organisation of scientific events | 26 |
| 9.1.2 | Participation in program committees | 26 |
| 9.1.3 | Participation in editorial boards | 27 |
| 9.1.4 | Invited talks | 27 |
| 9.1.5 | Research administration | 27 |
| 9.2 | Teaching - Supervision - Juries | 27 |
| 9.2.1 | Teaching | 27 |
| 9.2.2 | Supervision | 28 |
| 9.2.3 | Juries | 29 |
| 9.3 | Popularization | 29 |
| 10 | Scientific production | 29 |
| 10.1 | Major publications | 29 |
| 10.2 | Publications of the year | 30 |
| 10.3 | Cited publications | 32 |

1 Team members, visitors, external collaborators

Research Scientists

- François Pottier [Team leader, INRIA, Senior Researcher, HDR]
- Damien Doligez [INRIA, Researcher]
- Yannick Forster [INRIA, Researcher]
- Jean-Marie Madiot [INRIA, Researcher]
- Luc Maranget [INRIA, Researcher]
- Didier Remy [INRIA, Senior Researcher, HDR]
- Yannick Zakowski [INRIA, Researcher, from Apr 2025]

Faculty Member

- Xavier Leroy [COLLEGE DE FRANCE, Professor]

Post-Doctoral Fellow

- Euisun Yoon [INRIA, Post-Doctoral Fellow, until Jan 2025]

PhD Students

- Clement Allain [INRIA]
- Mathis Bouverot-Dupuis [INRIA, from Sep 2025]
- Timothee Huneau [INRIA, from Oct 2025]
- Tiago Lopes Soares [UNIV NOVA LISBONNE]
- Remy Seassau [INRIA]
- Samuel Vivien [PSL]
- Poyraz Yilan [AIRBUS, CIFRE, from Nov 2025]

Technical Staff

- Litao Zhou [INRIA, Engineer, from Oct 2025]

Interns and Apprentices

- Mathis Bouverot-Dupuis [ENS PARIS, Intern, until Feb 2025]
- Jean Caspar [ENS PARIS, Intern, from Oct 2025]
- Anton Danilkin [ENS Paris, Intern, from Jul 2025 until Aug 2025]
- Simon Dima [ENS PARIS, Intern, from Mar 2025 until Aug 2025]
- Zhicheng Hui [INRIA, Intern, from Mar 2025 until Aug 2025]
- Yvan Parent [UNIV PARIS SACLAY, Intern, until Apr 2025]
- Clement Pinard [INRIA, Intern, from May 2025 until Jul 2025]
- Maxime Ponsonnet [ENS DE LYON, Intern, from Jun 2025 until Jul 2025]
- Hugo Segoufin-Chollet [ENS PARIS, Intern, from Jun 2025 until Jul 2025]

Administrative Assistants

- Marina Kovacic [INRIA]
- Helene Milome [INRIA]

2 Overall objectives

The research conducted in the Cambium team aims at improving the safety, reliability and security of software through advances in programming languages and in formal program verification. Our work is centered on the design, formalization, and implementation of programming languages, with particular emphasis on type systems and type inference, formal program verification, shared-memory concurrency and weak memory models. We are equally interested in theoretical foundations and in applications to real-world problems. The OCaml programming language, the CompCert C compiler, and the Coq proof assistant embody many of our research results.

2.1 Software reliability and reusability

Software nowadays plays a pervasive role in our environment: it runs not only on general-purpose computers, as found in homes, offices, and data centers, but also on mobile phones, credit cards, inside transportation systems, factories, and so on. Furthermore, whereas building a single isolated software system was once rightly considered a daunting task, today, tens of millions of developers throughout the world collaborate to develop software components that have complex interdependencies. Does this mean that the “software crisis” of the early 1970s, which Dijkstra described as follows, is over?

By now it is generally recognized that the design of any large sophisticated system is going to be a very difficult job, and whenever one meets people responsible for such undertakings, one finds them very much concerned about the reliability issue, and rightly so. – Edsger W. Dijkstra

To some extent, the crisis is indeed over. In the past five decades, strong emphasis has been put on **modularity** and **reusability**. It is by now well-understood how to build reusable software components, thus avoiding repeated programming effort and reducing costs. The availability of hundreds of thousands of such components, hosted in collaborative repositories, has allowed the software industry to bloom in a manner that was unimaginable a few decades ago.

As pointed out by Dijkstra, however, the problem is not just to build software, but to ensure that it works. Today, the **reliability** of most software leaves a lot to be desired. Consumer-grade software, including desktop, Web, and mobile phone applications, often crashes or exhibits unexpected behavior. This results in loss of time, loss of data, and also can often be exploited for malicious purposes by attackers. Reliability includes **safety**—exhibiting appropriate behavior under normal usage conditions—and **security**—resisting abuse in the hands of an attacker.

Today, achieving very high levels of reliability is possible, albeit at a tremendous cost in time and money. In the aerospace industry, for instance, high reliability is obtained via meticulous development processes, extensive testing efforts, and external reviewing by independent certification authorities. There and elsewhere, **formal verification** is also used, instead of or in addition to the above methods. In the hardware industry, model-checking is used to verify microprocessor components. In the critical software industry, deductive program verification has been used to verify operating system kernels, file systems, compilers, and so on. Unfortunately, these methods are difficult to apply in industries that have strong cost and time-to-market constraints, such as the automotive industry, let alone the general software industry.

Today, thus, we arguably still are experiencing a “reliable-software crisis”. Although we have become pretty good at producing and evolving software, we still have difficulty producing cheap reliable software.

How to resolve this crisis remains, to a large extent, an open question. Modularity and reusability seem needed now more than ever, not only in order to avoid repeated programming effort and reduce the likelihood of errors, but also and foremost to avoid repeated specification and verification effort. Still, apparently, the languages that we use to write software are not expressive enough, and the logics and tools that we use to verify software are not mature enough, for this crisis to be behind us.

2.2 Qualities of a programming language

A programming language is the medium through which an intent (software design) is expressed (program development), acted upon (program execution), and reasoned about (verification). It would be a mistake to argue that, with sufficient dedication, effort, time and cleverness, good software can be written in any programming language. Although this may be true in principle, in reality, the choice of an adequate programming language can be the deciding factor between software that works and software that does not, or even cannot be developed at all.

We believe, in particular, that it is crucial for a programming language to be **safe**, **expressive**, to encourage **modularity**, and to have a simple, well-defined **semantics**.

- **Safety.** The execution of a program must not ever be allowed to go wrong in an unpredictable way. Examples of behaviors that must be forbidden include reading or writing data outside of the memory area assigned by the operating system to the process and executing arbitrary data as if it were code. A programming language is safe if every safety violation is gracefully detected either at compile time or at runtime.
- **Expressiveness.** The programming language should allow programmers to think in terms of concise, high-level abstractions—including the concepts and entities of the application domain—as opposed to verbose, low-level representations or encodings of these concepts.
- **Modularity.** The programming language should make it easy to develop a software component in isolation, to describe how it is intended to be composed with other components, and to check at composition time that this intent is respected.
- **Semantics.** The programming language should come with a mathematical definition of the meaning of programs, as opposed to an informal, natural-language description. This definition should ideally be formal, that is, amenable to processing by a machine. A well-defined semantics is a prerequisite for proving that the language is safe (in the above sense) and for proving that a specific program is correct (via model-checking, deductive program verification, or other formal methods).

The safety of a programming language is usually achieved via a combination of design decisions, compile-time type-checking, and runtime checking. As an example design decision, memory deallocation, a dangerous operation, can be placed outside of the programmer’s control. As an example of compile-time type-checking, attempting to use an integer as if it were a pointer can be considered a type error; a program that attempts to do this is then rejected by the compiler before it is executed. Finally, as an example of runtime checking, attempting to access an array outside of its bounds can be considered a runtime error: if a program attempts to do this, then its execution is aborted.

Type-checking can be viewed as an automated means of establishing certain correctness properties of programs. Thus, type-checking is a form of “lightweight formal methods” that provides weak guarantees but whose burden seems acceptable to most programmers. However, type-checking is more than just a program analysis that detects a class of programming errors at compile time. Indeed, types offer a language in which the interaction between one program component and the rest of the program can be formally described. Thus, they can be used to express a high-level description of the service provided by this component (i.e., its API), independently of its implementation. At the same time, they protect this component against misuse by other components. In short, “type structure is a syntactic discipline for enforcing levels of abstraction”. In other words, types offer basic support for expressiveness and modularity, as described above.

For this reason, types play a central role in programming language design. They have been and remain a fundamental research topic in our group. More generally, the design of new programming languages and new type systems and the proof of their safety has been and remains an important theme. The continued evolution of OCaml, as well as the design and formalization of Mezzo [40], are examples.

2.3 Design, implementation, and evolution of OCaml

Our group’s expertise in programming language design, formalization and implementation has traditionally been focused mainly on the programming language OCaml [35]. OCaml can be described as a high-level statically-typed general-purpose programming language. Its main features include first-class functions,

algebraic data structures and pattern matching, automatic memory management, support for traditional imperative programming (mutable state, exceptions), and support for modularity and encapsulation (abstract types; modules and functors; objects and classes).

OCaml meets most of the key criteria that we have put forth above. Thanks to its static type discipline, which rejects unsafe programs, it is **safe**. Because its type system is equipped with powerful features, such as polymorphism, abstract types, and type inference, it is **expressive, modular**, and concise. Although OCaml as a whole does not have a **formal semantics**, many fragments of it have been formally studied in isolation. As a result, we believe that OCaml is a good language in which to develop complex software components and software systems and (possibly) to verify that they are correct.

OCaml has long served a dual role as a vehicle for our programming language research and as a mature real-world programming language. This remains true today, and we wish to preserve this dual role. On the research side, there are many directions in which the language could be extended. On the applied side, OCaml is used within academia (for research and for teaching) and in the industry. It is maintained by a community of active contributors, which extends beyond our team at Inria. It comes with a package manager, **opam**, a rich ecosystem of libraries, and a set of programming tools, including an IDE (Merlin), support for debugging and performance profiling, etc.

OCaml has been used to develop many complex systems, such as proof assistants (Coq, HOL Light), automated theorem provers (Alt-Ergo, Zenon), program verification tools (Why3), static analysis engines (Astrée, Frama-C, Infer, Flow), programming languages and compilers (SCADE, Reason, Hack), Web servers (Ocsigen), operating systems (MirageOS, Docker), financial systems (at companies such as Jane Street, LexiFi, Nomadic Labs), and so on.

2.4 Software verification

We have already mentioned the importance of formal verification to achieve the highest levels of software quality. One of our major contributions to this field has been the verification of programming tools, namely the CompCert optimizing compiler for the C language [49] and the Verasco abstract interpretation-based static analyzer [41]. Technically, this is deductive verification of purely functional programs, using the Coq proof assistant both as the prover and the programming language. Scientifically, CompCert and Verasco are milestones in the area of program proof, due to the complexity and realism of the code generation, optimization, and static analysis techniques that are verified. Practically, these formally-verified tools strengthen the guarantees that can be obtained by formal verification of critical software and reduce the need for other verification activities, attracting the interest of Airbus and other companies that develop critical embedded software.

CompCert is implemented almost entirely in Gallina, the purely functional programming language that lies at the heart of Coq. Extraction, a whole-program translation from Gallina to OCaml, allows Gallina programs to be compiled to native code and efficiently executed. Unfortunately, Gallina is a very restrictive language: it rules out all side effects, including nontermination, mutable state, exceptions, delimited control, nondeterminism, input/output, and concurrency. In comparison, most industrial programming languages, including OCaml, are vastly more expressive and convenient. Thus, there is a clear need for us to also be able to verify software components that are written in OCaml and exploit side effects.

To reason about the behavior of effectful programs, one typically uses a “program logic”, that is, a system of deduction rules that are tailor-made for this purpose, and can be built into a verification tool. Since the late 1960s, program logics for imperative programming languages with global mutable state have been in wide use. A key advance was made in the 2000s with the appearance of Separation Logic, which emphasizes local reasoning and thereby allows reasoning about a callee independently of its caller, about one heap fragment independently of the rest of the heap, about one thread independently of all other threads, and so on. Today, this field is extremely active: the development of powerful program logics for rich effectful programming languages, such as OCaml or Multicore OCaml, is a thriving and challenging research area.

Our team has expertise in this field. For several years, François Pottier has been investigating the theoretical foundations and applications of several features of Separation Logics, such as “hidden state” and “monotonic state”, and has developed expertise in Iris, a modern Separation Logic that is jointly developed by several European research teams. Jean-Marie Madiot has contributed to the Verified Software Toolchain,

which includes a version of Concurrent Separation Logic for a subset of C. Arthur Charguéraud¹ has developed CFML, an implementation of Separation Logic for a subset of OCaml. Armaël Guéneau² has extended CFML with the ability to simultaneously verify the correctness and the time complexity of an OCaml component. Glen Mével³ has extended Iris with support for the weak memory model of OCaml 5, while Paulo de Vilhena has extended it with support for effect handlers. Alexandre Moine, in collaboration with Madiot, Pottier, and Charguéraud, has extended Iris with the ability to verify the space complexity of an OCaml component. Clément Allain has carried out a proof of compiler correctness using a relational variant of Iris and has verified several lock-free concurrent data structures using Iris.

We envision several ways of using OCaml components that have been verified using a program logic. In the simplest scenario, some key OCaml components, such as the standard library, are verified, and are distributed for use in unverified applications. This increases the general trustworthiness of the OCaml system, but does not yield strong guarantees of correctness. In a second scenario, a fully verified application is built out of verified OCaml components, therefore it comes with an end-to-end correctness guarantee. In a third scenario, while some components are written and verified directly at the level of OCaml, others are first written and verified in Gallina, then translated down to verified OCaml components by an improved version of Coq’s extraction mechanism. In this scenario, it is possible to fully verify an application that combines effectful OCaml code and side-effect-free Gallina code. This scenario represents an improvement over the current state of the art. Today, CompCert includes several OCaml components, which cannot be verified in Coq. As a result, the data produced by these components must be validated by verified checkers.

2.5 Shared-memory concurrency

Concurrent shared-memory programming seems required in order to extract maximum performance out of the multicore general-purpose processors that have been in wide use for more than a decade. (GPU’s and other special-purpose processors offer even greater raw computing power, but are not easily exploited in the symbolic computing applications that we are usually interested in.) Unfortunately, concurrent programming is notoriously more difficult than sequential programming. This can be attributed to a “state-space explosion problem”: the number of permitted program executions grows exponentially with the number of concurrent agents involved. Shared memory introduces an additional, less notorious, difficulty: on a modern multicore processor, execution does *not* follow the strong model where the instructions of one thread are interleaved with the instructions of other threads, and where reads and writes to memory instantaneously take effect. To properly understand and analyze a program, one must first formally define the semantics of the programming language, or of the device that is used to execute the program. The aspect of the semantics that governs the interaction of threads through memory is known as a **memory model**. Most modern memory models are **weak** in the sense that they offer fewer guarantees than the strong model sketched above.

Describing a memory model in precise mathematical language, in a manner that is at the same time faithful with respect to real-world machines and exploitable as a basis for reasoning about programs, is a challenging problem and a domain of active research, where thorough testing and verification are required.

Luc Maranget and Jean-Marie Madiot have acquired an expertise in the domain of weak memory models, including so-called axiomatic models and event-structure-based models. Moreover, Luc Maranget develops **diy-herd-litmus**, a unique software suite for defining, simulating and testing memory models. In short, **diy** generates so-called *litmus tests* from concise specifications; **herd** simulates litmus tests with respect to memory models expressed in the domain-specific language CAT; **litmus** executes litmus tests on real hardware. These tools have been instrumental in finding bugs in the deployed processors IBM Power5 and ARM Cortex-A9. Moreover, within industry, some models are now written in CAT, either for internal use, such as the AArch64 model by Will Deacon (ARM), or for publication, such as the RISC-V model by Luc Maranget and the HSA model by Jade Alglave and Luc Maranget.

For a long time, the OCaml language and runtime system have been restricted to sequential execution, that is, execution of a single computation thread on a single processor core. Yet, since OCaml 5, it is possible to execute multiple threads in parallel. The runtime system has been deeply impacted: in particular, OCaml’s garbage collector has been replaced with an entirely new concurrent collector. The memory model has been clearly defined, both on paper and in Coq. Also since OCaml 5, the language has been extended with

¹Formerly a PhD student in our team, today a researcher at Inria Nancy Grand-Est, team Camus.

²Also a former student in our team, today a research at Inria Saclay, team Toccata.

³A former student in our team.

effect handlers, a generalization of exception handlers. Effect handlers are a form of delimited control: they allow suspending a computation, storing it in memory, and resuming it at a later time.

3 Research program

Our research proposal is organized along three main axes, namely **programming language design and implementation**, **concurrency**, and **program verification**. These three areas have strong connections. For instance, the definition and implementation of OCaml intersects the first two axes, whereas creating verification technology for OCaml programs intersects the last two.

In short, the “programming language design and implementation” axis includes:

- The search for richer type disciplines, in an effort to make our programming languages safer and more expressive. Two domains, namely modules and effects, appear of particular interest. In addition, we view type inference as an important cross-cutting concern.
- The continued evolution of OCaml. The major evolutions that we envision in the medium term are the possible addition of a strong type-and-effect system, the addition of modular implicits, and a redesign of the type-checker.
- Research on refactoring and program transformations.
- Research on meta-programming techniques, in particular for proof assistants.

The “concurrency” axis includes:

- Research on weak memory models, including axiomatic models, operational models, and event-structure models.
- Research on the OCaml 5 memory model. This might include proving that the axiomatic and operational presentations of the model agree; testing the OCaml 5 implementation to ensure that it conforms to the model; and extending the model with new features, should the need arise.

The “program verification” axis includes:

- The continued evolution of CompCert.
- Building new verified tools, such as verified compilers for domain-specific languages, verified components for the Coq type-checker, and so on.
- Verifying algorithms and data structures implemented in OCaml, including concurrent data structures, and enriching Separation Logic with new features, if needed, to better support this activity.
- The continued development of tools for TLA+.

4 Application domains

4.1 Formal methods

We develop techniques and tools for the formal verification of critical software:

- program logics based on CFML and Iris for the deductive verification of software, including concurrency and algorithmic complexity aspects;
- verified development tools such as the CompCert verified C compiler, which extends properties established by formal verification at the source level all the way to the final executable code.
- improvements to the trustworthiness of different components of the Coq proof assistant, which is in turn used in other projects.

Some of these techniques have already been used in the nuclear industry (MTU Friedrichshafen uses CompCert to develop emergency diesel generators) and are under evaluation in the aerospace industry.

4.2 High-assurance software

Software that is not critical enough to undergo formal verification can still benefit greatly, in terms of reliability and security, from a functional, statically-typed programming language. The OCaml type system offers several advanced tools (generalized algebraic data types, abstract types, extensible variant and object types) to express many data structure invariants and safety properties and have them automatically enforced by the type-checker. This makes OCaml a popular language to develop high-assurance software, in particular in the financial industry. OCaml is the implementation language for the Tezos blockchain and cryptocurrency. It is also used for automated trading at Jane Street and for modeling and pricing of financial contracts at Bloomberg, Lexifi and Simcorp. OCaml is also widely used to implement code verification and generation tools at Facebook, Microsoft, CEA, Esterel Technologies, and many academic research groups, at Inria and elsewhere.

4.3 Design and test of microprocessors

The **diy** tool suite and the underlying methodology is in use at ARM Ltd to design and test the memory model of ARM architectures. In particular, the internal reference memory model of the ARMv8 (or AArch64) architecture has been written “in house” in Cat, our domain-specific language for specifying and simulating memory models. Moreover, our test generators and runtime infrastructure are used routinely at ARM to test various implementations of their architectures.

4.4 Teaching programming

Our work on the OCaml language family has an impact on the teaching of programming. OCaml is one of the programming languages selected by the French Ministry of Education for teaching Computer Science in classes préparatoires scientifiques. OCaml is also widely used for teaching advanced programming in engineering schools, colleges and universities in France, the USA, and Japan. The MOOC “Introduction to Functional Programming in OCaml”, developed at University Paris Diderot, is available on the France Université Numérique platform and comes with an extensive platform for self-training and automatic grading of exercises, developed in OCaml itself.

5 Latest software developments, platforms, open data

5.1 Latest software developments

5.1.1 OCaml

Keywords: Programming language, Functional programming, Compilers

Functional Description: The OCaml language is a functional programming language that combines safety with expressiveness through the use of a precise and flexible type system with automatic type inference. The OCaml system is a comprehensive implementation of this language, featuring two compilers (a bytecode compiler, for fast prototyping and interactive use, and a native-code compiler producing efficient machine code for x86, ARM, PowerPC, RISC-V and System Z), a debugger, and a documentation generator. Many other tools and libraries are contributed by the user community and organized around the OPAM package manager.

URL: <https://ocaml.org/>

Publications: [hal-04884634](#), [hal-04681703](#), [hal-04794404](#), [hal-03917754](#), [hal-03947986](#), [hal-04407119](#), [hal-03146495](#), [hal-03510931](#), [hal-03145030](#), [hal-01929508](#), [hal-03125031](#), [hal-00772993](#), [hal-00914493](#), [hal-00914560](#), [inria-00074804](#), [hal-01499973](#), [hal-01499946](#)

Contact: Florian Angeletti

Participant: 13 anonymous participants

5.1.2 Compcert

Name: The CompCert formally-verified C compiler

Keywords: Compilers, Formal methods, Deductive program verification, C, Coq

Functional Description: CompCert is a compiler for the C programming language. Its intended use is the compilation of life-critical and mission-critical software written in C and meeting high levels of assurance. It accepts most of the ISO C 99 language, with some exceptions and a few extensions. It produces machine code for the ARM, PowerPC, RISC-V, and x86 architectures. What sets CompCert C apart from any other production compiler, is that it is formally verified to be exempt from miscompilation issues, using machine-assisted mathematical proofs (the Coq proof assistant). In other words, the executable code it produces is proved to behave exactly as specified by the semantics of the source C program. This level of confidence in the correctness of the compilation process is unprecedented and contributes to meeting the highest levels of software assurance. In particular, using the CompCert C compiler is a natural complement to applying formal verification techniques (static analysis, program proof, model checking) at the source code level: the correctness proof of CompCert C guarantees that all safety properties verified on the source code automatically hold as well for the generated executable.

URL: <https://compcert.org/>

Contact: Xavier Leroy

Participant: 5 anonymous participants

Partner: AbsInt Angewandte Informatik GmbH

5.1.3 Diy

Name: Do It Yourself

Keyword: Parallelism

Functional Description: The diy suite provides a set of tools for testing shared memory models: the litmus tool for running tests on hardware, various generators for producing tests from concise specifications, and herd, a memory model simulator. Tests are small programs written in x86, Power or ARM assembler that can thus be generated from concise specification, run on hardware, or simulated on top of memory models. Test results can be handled and compared using additional tools.

URL: <http://diy.inria.fr/>

Contact: Luc Maranget

Participant: 2 anonymous participants

Partner: University College London UK

5.1.4 Menhir

Keywords: Compilation, Context-free grammars, Parsing

Functional Description: Menhir is a LR(1) parser generator for the OCaml programming language. That is, Menhir compiles LR(1) grammar specifications down to OCaml code. Menhir was designed and implemented by François Pottier and Yann Régis-Gianas.

Publications: [hal-03478172](#), [hal-01633123](#), [hal-01417004](#)

Contact: François Pottier

5.1.5 CFML

Name: Interactive program verification using characteristic formulae

Keywords: Coq, Software Verification, Deductive program verification, Separation Logic

Functional Description: The CFML tool supports the verification of OCaml programs through interactive Coq proofs. CFML proofs establish the full functional correctness of the code with respect to a specification. They may also be used to formally establish bounds on the asymptotic complexity of the code. The tool is made of two parts: on the one hand, a characteristic formula generator implemented as an OCaml program that parses OCaml code and produces Coq formulae, and, on the other hand, a Coq library that provides notations and tactics for manipulating characteristic formulae interactively in Coq.

URL: <http://www.chargueraud.org/softs/cfml/>

Contact: Arthur Charguéraud

Participant: 3 anonymous participants

5.1.6 TLAPS

Name: TLA+ proof system

Keyword: Proof assistant

Functional Description: TLAPS is a platform for developing and mechanically verifying proofs about specifications written in the TLA+ language. The TLA+ proof language is hierarchical and explicit, allowing a user to decompose the overall proof into proof steps that can be checked independently. TLAPS consists of a proof manager that interprets the proof language and generates a collection of proof obligations that are sent to backend verifiers. The current backends include the tableau-based prover Zenon for first-order logic, Isabelle/TLA+, an encoding of TLA+ set theory as an object logic in the logical framework Isabelle, an SMT backend designed for use with any SMT-lib compatible solver, and an interface to a decision procedure for propositional temporal logic.

URL: <https://tla.msr-inria.inria.fr/tlaps/content/Home.html>

Contact: Stephan Merz

Participant: 2 anonymous participants

Partner: Microsoft

5.1.7 ZENON

Name: The Zenon automatic theorem prover

Keywords: Automated theorem proving, First-order logic

Functional Description: Zenon is an automatic theorem prover based on the tableaux method. Given a first-order statement as input, it outputs a fully formal proof in the form of a Coq or Isabelle proof script. It has special rules for efficient handling of equality and arbitrary transitive relations. Although still in the prototype stage, it already gives satisfying results on standard automatic-proving benchmarks.

Zenon is designed to be easy to interface with front-end tools (for example integration in an interactive proof assistant) and also to retarget to output scripts for different frameworks (for example Dedukti).

URL: <http://zenon-prover.org/>

Publications: [inria-00338299v1](#), [hal-02305831v1](#), [inria-00315920v1](#), [hal-00909784v1](#), [tel-01420460v2](#), [hal-00909688v1](#), [hal-01204701v2](#), [hal-01171360v1](#), [hal-01100512v1](#), [hal-01099338v1](#), [hal-01243593v1](#), [hal-01420638v1](#), [hal-01342849v1](#)

Contact: Damien Doligez

Participant: an anonymous participant

5.1.8 hevea

Name: hevea is a fast latex to html translator.

Keywords: LaTeX, Web

Functional Description: HEVEA is a LATEX to html translator. The input language is a fairly complete subset of LATEX 2 (old LATEX style is also accepted) and the output language is html that is (hopefully) correct with respect to version 5. HEVEA understands LATEX macro definitions. Simple user style files are understood with little or no modifications. Furthermore, HEVEA customisation is done by writing LATEX code.

HEVEA is written in Objective Caml, as many lexers. It is quite fast and flexible. Using HEVEA it is possible to translate large documents such as manuals, books, etc. very quickly. All documents are translated as one single html file. Then, the output file can be cut into smaller files, using the companion program HACHA. HEVEA can also be instructed to output plain text or info files.

Information on HEVEA is available at <http://hevea.inria.fr/>.

URL: <http://hevea.inria.fr/>

Contact: Luc Maranget

6 New results

6.1 Long-term software projects

6.1.1 The CompCert formally-verified compiler

Participants: Xavier Leroy, , Michael Schmidt (*AbsInt GmbH*), , Bernhard Schommer (*Saarland University and AbsInt GmbH*). .

Since 2005, in the context of our work on compiler verification, we have been developing and formally verifying CompCert, a moderately-optimizing compiler for a large subset of the C programming language. CompCert generates assembly code for the ARM, PowerPC, RISC-V, and x86 architectures [49]. The compiler is written mostly within the specification language of the Rocq proof assistant, out of which Rocq’s extraction facility generates executable OCaml code. The compiler comes with a 100000-line machine-checked proof of semantic preservation, establishing that the generated assembly code executes exactly as prescribed by the semantics of the source C program.

This year, we extended CompCert towards the generation of position-independent code (PIC) and position-independent executables (PIE). This allows CompCert to produce shared libraries and improves code security.

We also improved the neededness analysis of 64-bit integer operations, which allows for the removal of more useless integer computations. We also extended common subexpression elimination (CSE) and dead code elimination to optimize calls to known, pure built-in functions and arithmetic helper functions.

These improvements were released as part of CompCert version 3.16 in September 2025.

6.1.2 The OCaml system

Participants: Florian Angeletti, Damien Doligez, Sébastien Hinderer, Xavier Leroy, Luc Maranget, Clément Allain, Samuel Vivien, David Allsop (*Cambridge University*), Nick Barnes (*Tarides*), Stephen Dolan (*Cambridge University*), Jacques Garrigue (*University of Nagoya*), Sadiq Jaffer (*Tarides*), Guillaume Munch-Maccagnoni (*Inria team Gallinette*), Olivier Nicole (*Tarides*), Nicolás Ojeda Bär (*Lexifi*), KC Sivaramakrishnan (*IIT Madras*), Gabriel Scherer (*Inria team Picube*).

This year we released one new minor version of OCaml, OCaml 5.4.0. Some of the highlights of this new version are:

- Labeled tuples
- Immutable arrays
- Array literal syntax (through type-directed disambiguation) for immutable arrays and arrays of floating-point numbers
- Atomic record fields
- Four new standard library modules: `Pair`, `Pqueue`, `Repr`, and `Tarray`
- Restored “memory cleanup upon exit” mode
- A new chapter in the manual on profiling OCaml programs on Linux and macOS

As usual, this release also contains many incremental changes:

- Many runtime and code generation improvements
- More than thirty new standard library functions
- Nearly a dozen improved error messages
- Around fifty bug fixes

6.1.3 The `diy` tool suite

Participants: Hadrien Renaud (*University College London*), Artem KhyzhaARM Ltd. , Nikos NikorelisARM Ltd. , Vladimir Murzin (*ARM Ltd.*), Jade Alglave (*ARM Ltd. and University College London*), Luc Maranget.

The `diy` suite provides a set of tools for testing shared memory models: the `litmus` tool for running tests on hardware, various generators for producing tests from concise specifications, and `herd`, a memory model simulator. Tests are small programs written in x86, Power, ARM, generic (LISA) assembler, or a subset of the C language that can thus be generated from concise specifications, run on hardware, or simulated on top of memory models. Test results can be handled and compared using additional tools. One distinctive feature of our system is `Cat`, a domain-specific language for memory models.

This year saw many enhancements and maintenance work. As a developer, Luc Maranget contributed to the global coherence of the tools, adding specific features such as “instructions as data” for all architectures, ensuring that newly implemented instructions are available in all tools, extending test suites, etc.

Luc Maranget acts as the main maintainer and coordinator of the toolbox: with others, he reviews and validates the pull requests submitted by contributors. `diy` now has about 10 frequent contributors, most of whom are employed by ARM as engineers or interns. Nikos Nikorelis, Artem Khyzha and Vladimir Murzin are cited for their regular and significant contributions and reviews.

6.1.4 TLA+

Participants: Damien Doligez, , Andrew Helwer (*Disjunctive Consulting*), Igor Konnov (*Informal Systems*), , Markus Kuppe (*Oracle*), , Stephan Merz (*Inria Nancy*). , Karolis Petrauskas (*Individual contributor*).

This project produces and maintains tools for managing and verifying proofs in the proof language of TLA+ [48].

We have been working on the maintenance and development of the TLAPM software, which takes place on GitHub, with the help of external contributors.

6.1.5 Menhir

Participants: François Pottier.

From September to December 2025, François Pottier spent significant effort cleaning up and restructuring the code of the parser generator **Menhir**. This effort should make the code easier to modify and reuse. As immediate benefits, several stand-alone libraries have been isolated: **intPQueue**, which offers fast priority queues; **patricia**, which offers immutable integer maps; and **bitsets**, which offers immutable bit sets and mutable vectors. Furthermore, Menhir has been extended with a new GLR back-end, which allows non-deterministic parsing, that is, parsing with an ambiguous context-free grammar. No paper about this work has not been published yet.

6.2 Programming language design and implementation

6.2.1 Implementing and formalizing Modular Explicit

Participants: Samuel Vivien, Didier Rémy, Gabriel Scherer (*Inria team Picube*).

*Modular explicit*s are functions that take modules as arguments. This language feature is a prerequisite for the elaboration of *modular implicit*s (§6.2.2).

In 2025, our work on modular explicit has been continued. On the implementation side, their integration in the OCaml compiler has been approved; they will be included in the upcoming version 5.5. On the formalization side, we detailed type inference for module-dependent functions and described their elaboration into F^ω , following the framework introduced by Blaudeau and Rémy for OCaml modules [42]. This work has been presented at the conference JFLA 2026 [25].

6.2.2 Designing and formalizing Modular Implicit

Participants: Samuel Vivien, Didier Rémy, Gabriel Scherer (*Inria team Picube*).

*Modular implicit*s are modules that are passed as implicit parameters to functions. Our work on modular implicit is based on a proposal and prototype implementation by White, Bour and Yallop [52]. Extending OCaml with modular implicit seems desirable because OCaml's functors are extremely verbose, preventing their use at a small scale. Modular implicit would give us some of the flexibility and ease of use of Haskell's type classes. To this end, we have been working on defining a clear specification of implicit argument synthesis and on optimizing the synthesis algorithm so that it scales up to real-world applications.

In 2025, we separated our work in two steps, namely module inference and interaction with the core language. We worked mainly on module inference, by implementing a prototype for the OCaml compiler

and by formalizing module inference. A presentation about this work has been given at the workshop ML 2025 [29].

6.2.3 Omnidirectional type inference

Participants: Didier Rémy, Gabriel Scherer (*Inria team Picube*), Alistair O’Brien (*Cambridge University*).

The Damas-Hindley-Milner (ML) type system owes its success to *principality*, the property that every well-typed expression has a unique most general type. This makes inference predictable and efficient. Yet, principality is *fragile*: many extensions of ML—GADTs, higher-rank polymorphism, and static overloading—break it by introducing *fragile* constructs that resist principal inference. Existing approaches recover principality through *directional* inference algorithms, which propagate *known* type information in a fixed order. However, the rigidity of a *static* inference order often causes otherwise well-typed programs to be rejected.

We propose *omnidirectional* type inference, where type information flows in a *dynamic* order, suspending typing constraints when progress requires known type information and resuming once such information becomes available. This approach is straightforward for simply-typed systems, but extending it to ML is challenging due to *let-generalization*. We introduce *incremental instantiation*, allowing partially solved type schemes containing suspended constraints to be instantiated, with a mechanism to incrementally update instances as the scheme is refined. Omnidirectionality provides a *general framework* for restoring principality in the presence of fragile features. We demonstrate its versatility on two fundamentally different features of OCaml: static overloading of record labels and datatype constructors and semi-explicit first-class polymorphism. This work has been presented at the workshop WITS 2026 and has been submitted for journal publication [28, 37].

As a side topic, Didier Rémy revisited and improved semi-unification-based type inference for ML, using scopes to track polymorphism. He implemented this type inference technique in a small experimental prototype, which he extended with omnidirectional inference applied to static overloading. This work has not yet been published.

6.2.4 Formalization of recursive modules

Participants: Didier Rémy, Litao Zhou.

Litao Zhou has been a visiting intern at Cambium from October 2025 until March 2026, on leave from Hong-Kong University. He and Didier Rémy have worked on the formalization of recursive modules. Following a previous approach by Clément Blaudeau and Didier Rémy [42], where a module language named M^ω was elaborated into F^ω , they extended M^ω with recursive modules in a way that allows the so-called *double vision*, as proposed by Derek Dreyer. This work has not yet been published.

6.2.5 Tail Modulo Async-Await

Participants: Emma Nardino (*ENS Lyon*), Ludovic Henrio (*CNRS*), Gabriel Radanne (*Inria team Cash*), Yannick Zakowski.

We extend the tail-call optimization by extending it to asynchronous calls. We first introduce Tail-Modulo-Await, a novel code transformation for asynchronous tail recursive functions that prevents the creation of unnecessary tasks. We then show how to combine Tail-Modulo-Await with the existing Tail-Modulo-Cons optimization; we obtain an optimization that can turn a recursive function with multiple tail calls under constructors into a parallel version of the function, also optimized in space.

This year, we have completed a paper formalization of the transformation, as well as proof-of-concept implementation for OCaml. A draft paper about this work has been written [36]; we expect to submit it for publication in 2026.

6.2.6 Mapping and explaining syntax errors with LRgrep

Participants: Frédéric Bour, François Pottier.

LR parsers and generated parsers are often criticized for their perceived inability to produce good syntax error messages. Yet, we believe, this inability is not inherent to LR parsing or to the use of a parser generator. Instead, we claim, this perception is caused mainly by the lack of languages and tools that allow the problem of producing good syntax error messages to be addressed at a suitable level of abstraction. To remedy this problem, during his PhD thesis, defended in December 2024, Frédéric Bour developed the LRgrep language and its compiler. In 2025, Frédéric Bour and François Pottier wrote a short tutorial paper on LRgrep. The paper demonstrates how to use LRgrep to construct good syntax error explanations for an LR parser. This paper has been presented at the conference JFLA 2026 [22].

6.3 Semantics

6.3.1 Choice Trees: reasoning about nondeterministic, recursive, impure programs in Rocq

Participants: Nicolas Chappe (*ENS Lyon*), Paul He (*University of Pennsylvania*), Ludovic Henrio (*CNRS*), Eleftherios Ioannidis (*University of Pennsylvania*), Yannick Zakowski, Steve Zdancewic (*University of Pennsylvania*).

Choice Trees are a monad for modeling nondeterministic, recursive, and impure programs in Rocq. They extend Interaction Trees to support non-determinism, providing notably bisimulations and simulations to reason about them. In addition to a monadic interpretation of effects, they support the refinement of non-deterministic choices.

Choice trees have initially been introduced at POPL 2023. This year, we have published an extended version in the Journal of Functional Programming [45] describing the current state of the library, and in particular a novel characterization of the core bisimulation supporting more expressive proof rules.

6.3.2 Executable semantics for a concurrent subset of LLVM IR

Participants: Nicolas Chappe (*ENS Lyon*), Ludovic Henrio (*CNRS*), Yannick Zakowski.

The Vellvm project formalizes the semantics of LLVM IR in a denotational style, as a monadic interpreter built on Interaction Trees. In this work, we explore the perspective of extending Vellvm to support threads and a weakly consistent memory model. We demonstrate that Choice Trees (§6.3.1) are a viable structure to this end by instantiating the approach over a minimal concurrent subset of LLVM IR. This work has been presented at CPP 2025 [46].

6.3.3 Layers of confluence for actors

Participants: Åsmund Aqissaq Arild Kløvstad (*University of Oslo*), Einar Broch Johnsen (*University of Oslo*), Ludovic Henrio (*CNRS*), Violet Ka I Pun (*Western Norway University of Applied Sciences*), Yannick Zakowski.

We introduce a novel proof method to prove the confluence, i.e., the observational determinacy, of programs written in an actor-like programming paradigm. The approach, based on a result by de Bruijn that allows stratifying a rewrite system and its proof of confluence in levels, is formalized in the Rocq proof assistant. This year, we have finished the formalization of our case study and written a paper that has been presented at the conference CPP 2026 [23].

6.3.4 An abstract, certified account of operational game semantics

Participants: Peio Borthelle (*Laboratoire LAMA*), Tom Hirschowitz (*Laboratoire LAMA*), Guilhème Jaber (*Laboratoire LS2N*), Yannick Zakowski.

Operational game semantics (OGS) is a method for interpreting programs as labeled transition systems over suitable games. Such an interpretation is considered sound if weak bisimilarity of associated strategies entails contextual equivalence. In this line of work, we contribute to the unification and mechanization of OGS by proposing an abstract notion of language with evaluator, for which we construct a generic OGS interpretation, which we prove sound.

This year, this major development has been completed, leading to a publication at ESOP 2025 [44] and to Peio Borthelle’s successful PhD defense [43].

6.3.5 Structural temporal logic for mechanized program verification

Participants: Eleftherios Ioannidis (*University of Pennsylvania*), Yannick Zakowski, Steve Zdancewic (*University of Pennsylvania*), Sebastian Angel (*University of Pennsylvania*).

Interaction Trees provide a language embedded in Rocq to conveniently model programming languages as expanded labeled transition systems, with support for monadic implementations of these labels. We consider here the verification of temporal properties of computations described as such infinite trees. To do so, we formalize an LTL logic through a mixed inductive/coinductive characterization, and establish a proof system to interactively establish such properties. This work has been presented at the conference OOPSLA 2025 [16].

6.3.6 MTrees: mixing monadic effects and labeled transition systems in Rocq

Participants: Peio Borthelle (*Laboratoire LAMA*), Cyril Cohen (*Cash*), Ludovic Henrio (*CNRS*), Théa Hervier (*ENS Lyon*), Yannick Zakowski.

Interaction Trees introduced a mechanization of the initial iterative monad over a given set of free operations. Additional effects are then realized by monadic interpretation of these operations. In this work, we explore a different route: mechanizing the initial iterative monad supporting both a given set of free operations and a given polynomial monad. This new approach leads to the study of highly generic notions of weak bisimilarity, contributes to better understand and unify works such as Choice Trees, and suggest new semantic domains for programming languages.

This year, Théa Hervier has covered concrete instances during her master’s internship [32]. We have then mechanized the general case; Peio Borthelle introduced a novel notion of weak bisimilarity with respect to which we established the initiality. In the upcoming months, we expect to formalize a case study and submit our results to POPL 2027.

6.3.7 Axiomatic memory models and virtual memory

Participants: Jade Alglave (*ARM Ltd. and University College London*), Richard Grisenthwaite (*ARM Ltd.*), Artem Khyzha (*ARM Ltd.*), Luc Maranget, Nikos Nikoleris (*ARM Ltd.*).

6.3.8 Semantics and sound implementation of AArch64 instructions

Participants: Hadrien Renaud (*University College London*), Jade Alglave (*ARM Ltd. and University College London*), Luc Maranget.

Hadrien Renaud is a PhD candidate under the supervision of Jade Alglave and Luc Maranget. Hadrien Renaud’s thesis aims at automating the production of intra-instruction dependencies, based on their definitions in pseudo-code. The task requires not only significant implementation work but also an in-depth study of the semantics of instructions and of the nature of various intra-instruction dependencies. Luc Maranget acts as a co-advisor, focusing on language definition and implementation. We hold a weekly meeting.

This year, Hadrien Renaud achieved significant progress toward the objective of extending our memory model simulator. More specifically, the *herd* simulator can now combine ARM Virtual Memory System Architecture (VMSA) and instruction semantics as specified in ARM Architecture Specification Language. It is important to notice that VMSA possesses an official specification in CAT, our language for memory models and that ASL now has a parallel implementation that generates intra-instruction dependencies. As a result we now provide an implementation that is based upon official specifications.

6.4 Mechanized mathematics and program verification in Rocq

6.4.1 Constructive reverse mathematics

Participants: Yannick Forster, Dominik Kirst (*Inria team Picube*), Timothée Huneau, Sam van Gool (*ENS Paris-Saclay*), Jean Caspar, Martin Baillon (*Inria team Picube*), Pierre-Marie Pédrot (*Inria team Gallinette*), Assia Mahboubi (*Inria team Gallinette*).

Constructive mathematics is based on logically economical foundations for mathematics, which exclude some logical axioms that are usually assumed in classical foundations of mathematics. In particular, constructive mathematics enforces constructions: for example, proofs of existence must construct explicit witnesses.

Some theorems of classical mathematics are not provable in constructive mathematics because they rely on axioms of classical mathematics. To study such theorems and understand their exact status, constructive *reverse* mathematics proves that axioms imply a theorem, and that, conversely, the theorem implies the axioms, thus establishing equivalences, also known as exact characterizations.

Continuing a long line of past work, Yannick Forster and Dominik Kirst have supervised Timothée Huneau’s internship. Timothée studied the exact status of the Löwenheim-Skolem theorem in model theory of first-order logic. Timothée has now started a PhD on extending this work into a general study of the reverse status of results in model theory.

Independently, Yannick Forster has worked on understanding constructive subtleties when defining the notion of continuous total functions in constructive mathematics. This resulted in a paper at FSCD 2025 [21]. This work is continued in an internship by Jean Caspar, whose aim is to generalize the notions of continuity so that they apply to partial functions.

6.4.2 Synthetic computability theory

Participants: Yannick Forster, Dominik Kirst (*Inria team Picube*), Haoyi Zeng (*Saarland University*), Takako Nemoto (*Tohoku University*), Martin Trucchi (*Inria team Picube*), Sara Rousta (*Inria team Picube*).

Proofs in traditional computability theory are notoriously hard to formalize, due to their reliance on models of computation such as Turing machines. In synthetic computability, one abstracts away from models of computation, which is possible thanks to a formal foundation for mathematics where functions and propositions are strictly separated. This is the case in many constructive foundations for mathematics, including the one that underlies the Rocq proof assistant.

Continuing a long line of work on synthetic computability in proof assistants, Yannick Forster, Dominik Kirst, Haoyi Zeng, and Takako Nemoto have written a paper on the status of Post’s problem in synthetic computability theory [39].

Yannick Forster and Dominik Kirst co-supervise the internships of Martin Trucchi and Sara Rosta in the Picube team. Martin studies synthetic notions of randomness, such as Kolmogorov and Martin-Löf randomness. Sara studies how to define realisability semantics for arithmetic using synthetic axioms.

Yannick Forster has given an invited talk about this line of work at the 33rd EACSL Annual Conference on Computer Science Logic (CSL 2025) and at the Simons Institute for the Theory of Computing and SLMATH Joint Workshop: AI for Mathematics and Theoretical Computer Science in April 2025.

6.4.3 A lazy, concurrent convertibility checker

Participants: Nathanaëlle Courant (*OCamlPro*), Xavier Leroy.

During her PhD (defended in 2024), Nathanaëlle Courant developed a novel algorithm to decide convertibility, that is, to test whether two terms (such as types or logical propositions) are equal up to some computation steps. Convertibility plays a crucial role in the implementation of proof assistants such as the Rocq prover.

This year, Nathanaëlle Courant and Xavier Leroy reformulated this algorithm using a simple process calculus, and extended it to handle η -conversion, that is, the equivalence between $\lambda x. M x$ and M . Convertibility up to η is used in Agda, Lean, and Rocq.

A paper about this work has been presented at the conference POPL 2026 [14].

6.4.4 Meta-programming in Rocq

Participants: Yannick Forster, Matthieu Sozeau (*Inria team Gallinette*), Weituo Dai, Thomas Lamiaux (*Inria team Gallinette*), Mathis Bouverot-Dupuis.

The state of meta-programming in the Rocq Prover (previously known as the Coq proof assistant) is diverse and has grown organically over the years. Various meta-programming languages are used to implement the kernel, tactics, and plugins.

In September 2025, Mathis Bouverot-Dupuis started a PhD, supervised by Yannick Forster and François Pottier. He aims to develop a robust meta-programming framework based on the lessons learned through 40 years of organic development of such frameworks in Rocq.

Mathis gave a talk about his 2024 internship on investigating the needs for such a framework at the conference TYPES 2025. A paper in the form of an experience report has been accepted for presentation at the conference ESOP 2026 [31].

6.4.5 The MetaRocq project

Participants: Yannick Forster, Matthieu Sozeau (*Inria team Gallinette*), Nicolas Tabareau (*Inria team Gallinette*), Yee-Jian Tan, Thomas Lamiaux (*Inria team Gallinette*).

The MetaRocq project (previously known as MetaCoq) is an umbrella project that contains several subprojects. These include a formal definition of Rocq’s type theory and a verified type- and proof-checker for Rocq.

Several crucial parts of Rocq’s type theory have not yet been formally studied. These include the guard checker, which ensures that recursive functions terminate, and the strict positivity condition, which ensures that inductive type definitions are consistent. The proof assistant Lean, a younger sibling to Rocq, largely shares the same type theory.

In 2024, Yee-Jian Tan carried out an internship, supervised by Yannick Forster, on understanding the current implementation of Rocq’s guard checker. This year, he presented this work at the conference TYPES 2025.

Together with Matthieu Sozeau and Nicolas Tabareau, Yannick Forster co-supervises Thomas Lamiaux’s PhD thesis in the Gallinette team in Nantes. They have submitted a paper which clarifies the strict positivity condition, focusing in particular on nested inductive types, both in Rocq and in Lean [33].

6.4.6 Code generation from proof assistants

Participants: Yannick Forster, Matthieu Sozeau (*Inria team Gallinette*), Nicolas Tabareau (*Inria team Gallinette*), Hugo Segoufin, Simon Dima, Zoe Paraskevopoulou (*National Technical University of Athens*), Bas Spitters (*University of Aarhus*).

Based on the extraction pipeline from Rocq to OCaml developed by Yannick Forster, Matthieu Sozeau, and Nicolas Tabareau, Yannick Forster carries on a collaboration whose aim is to extend the code generation abilities of proof assistants, with a concern of verifying the correctness of the generation algorithms.

Yannick Forster has given an invited talk about this line of work at the Workshop on Logical Frameworks and Meta Languages: Theory and Practice (LFMTP 2025).

6.4.7 Implementing and verifying Kaplan and Tarjan’s catenable dequeues

Participants: Jules Viennot, Arthur Wendling (*Tarides*), François Pottier, Armaël Guéneau (*Inria team Toccata*).

In 1999, Kaplan and Tarjan introduced *purely functional, real-time catenable dequeues*, a data structure that supports five fundamental operations, namely insertion and extraction of one element at either end and concatenation. All operations are performed with a worst-case time complexity of $O(1)$. Furthermore, this data structure is persistent: no operation modifies or destroys its argument.

Kaplan and Tarjan’s paper provides an English description of this data structure, without code. This data structure is in fact extremely difficult to implement: in the 25 years that have elapsed since the paper was published, no implementation appeared.

In 2024, Arthur Wendling was able to implement this data structure in OCaml. His implementation makes sophisticated use of OCaml’s generalized algebraic data types (GADTs). Then, Jules Viennot, supervised by Armaël Guéneau and François Pottier, ported Wendling’s code from OCaml to Rocq and verified its correctness. This work stresses the limits of Rocq’s expressiveness. It is the first published or verified implementation of Kaplan and Tarjan’s catenable dequeues.

A paper on this topic has been submitted to the journal *Logical Methods in Computer Science* in May 2025. The reviewers requested additional clarifications and a performance evaluation. The paper has been revised and submitted again in January 2026.

6.5 Program verification in separation logic

6.5.1 Implementing and verifying Kaplan, Okasaki, and Tarjan’s simple catenable deques

Participants: Juliette Ponsonnet (*ENS Lyon*), François Pottier.

In 2000, Kaplan, Okasaki, and Tarjan proposed a “simple” persistent catenable deque data structure. This data structure has mutable internal state and relies on a restricted form of mutation. It supports insertion and extraction at either end, as well as concatenation, with worst-case *amortized* time complexity $O(1)$. Compared with Kaplan and Tarjan’s catenable deques (§6.4.7), this data structure is simpler, possibly faster in practice, but offers a weaker time complexity guarantee.

In 2025, during a six-week internship, Juliette Ponsonnet, advised by François Pottier, implemented Kaplan, Okasaki, and Tarjan’s simple catenable deques in OCaml. Furthermore, she carried out *two* program verification efforts about this data structure, and obtained the following two results. First, using Iris, she proved that the code is correct, in sequential and concurrent usage scenarios. Second, using Iris with time credits, she verified that, provided concurrent accesses are forbidden, every operation has amortized time complexity $O(1)$. This required repairing a possible mistake in Kaplan, Okasaki, and Tarjan’s description.

A paper on these results has been accepted for publication and presentation at the conference JFLA 2026 [24].

6.5.2 Verification of concurrent OCaml 5 programs in separation logic

Participants: Clément Allain, Gabriel Scherer (*Inria team Picube*).

The release of OCaml 5, which introduced parallelism into the language, created a need for safe and efficient concurrent data structures. Several new libraries, such as Saturn [47], aim at addressing this need. From the perspective of formal verification, this is an opportunity to apply and further state-of-the-art techniques to provide stronger guarantees.

Clément Allain designed and implemented Zoo, a framework for verifying fine-grained concurrent OCaml 5 algorithms. Following a pragmatic approach, he defined ZooLang, a limited fragment of the OCaml language, embedded inside Rocq, which is sufficient to faithfully express these algorithms. He formalized its semantics via a deep embedding in the Rocq proof assistant, uncovering several subtle aspects of physical equality in the process. He provided a tool that translates OCaml programs into ZooLang abstract syntax. The programs can then be specified and verified using the concurrent separation logic Iris. To illustrate the applicability of Zoo, he verified a subset of the OCaml standard library and a collection of fine-grained concurrent data structures from the libraries Saturn [47] and Eio [50].

This work has been presented at the conference POPL 2026 [12] and in Clément Allain’s PhD thesis, which he defended in December 2025 [30].

6.5.3 A verified parallel scheduler for OCaml 5

Participants: Clément Allain, Gabriel Scherer (*Inria team Picube*).

Clément Allain implemented a realistic parallel scheduler for OCaml 5 and verified it using the Zoo framework (§6.5.2). This scheduler relies on a work-stealing strategy to perform load balancing but also supports other scheduling strategies thanks to its flexible interface. Several basic benchmarks demonstrate that its performance is on par with other schedulers from the OCaml ecosystem.

As part of this effort, Clément Allain verified the Chase-Lev work-stealing deque, as implemented in the Saturn library [47]. He showed that this data structure has a subtle external and future-dependent linearization point. To deal with it, we introduce new abstractions that help exploit prophecy variables in Iris.

This work has been submitted for publication at the conference PLDI 2026.

6.5.4 Osiris: formal semantics and reasoning rules for OCaml

Participants: Remy Seassau, François Pottier, Irene Yoon, Jean-Marie Madiot.

The Osiris project aims to develop an instance of the Iris separation logic, inside the Rocq proof assistant, and to customize it for the OCaml programming language, so as to provide end users with a powerful, state-of-the-art, ready-to-use program verification environment for OCaml.

As part of this long-running project, a first paper about Osiris has been published and presented at the conference ICFP 2025 [19]. This paper presents a mechanized dynamic semantics for a fragment of OCaml, structured as a monadic interpreter, running on top of a monad whose behavior is described by a small-step operational semantics. For this dynamic semantics, we construct two program logics, Horus and Osiris. Horus, a Hoare logic, allows reasoning about side-effect-free expressions only; Osiris, an instance of Iris, allows reasoning about all kinds of expressions.

After this paper was published, we have added support for concurrency in the dynamic semantics and in the program logic.

6.5.5 Gospel: a formal specification language for OCaml

Participants: Tiago Soares, François Pottier, Mário Pereira (*NOVA LINCS*).

Gospel is a formal specification language for OCaml. It allows a library's interface files to contain not only types, but also logical specifications, which describe the library's behavior. Gospel can be viewed as a pleasant surface syntax for Separation Logic specifications.

In 2025, we added *lenses* to the language; they are a lightweight notation for the Separation Logic assertions that relate a runtime data structure and its logical model. Furthermore, we defined an intermediate language, GospelSL, which can be translated both into CFML and into Iris, and re-implemented our translation of Gospel into CFML and Iris. In the process, we had to more clearly define certain features of Gospel, such as the way in which program values can be used in logical formulae. This update to the Gospel ecosystem has been presented at the conference JFLA 2026 [51].

6.5.6 Verification of parallel real-time programs

Participants: Jean-Marie Madiot, Dumitru Potop-Butucaru (*Inria team AT-PRO*), Poyraz Yilan.

In some settings, real-time programs can be transformed to parallel and concurrent C programs that satisfy a strong notion of confluence. As a first step towards the formal verification of the soundness of these transformations, while taking into account the relaxed memory model of the C language, Jean-Marie Madiot, Dumitru Potop-Butucaru, and Poyraz Yilan (then an intern at Cambium) developed a framework for a class of such programs.

This framework offers a set of reasoning rules whose correctness is proved based on Iris and on another, intricate, logic for the C11 release/acquire synchronization discipline. These new rules, which are tailored for this class of programs, establish program safety and are much simpler to use than the general C11 rules.

Poyraz is starting a CIFRE PhD thesis with Airbus on this subject.

6.5.7 Separation logic for one-shot undelimited continuations

Participants: Xavier Leroy, Paulo de Vilhena (*Imperial College*).

For his forthcoming book on control structures for programming languages, Xavier Leroy conducted a survey of program logics for reasoning about control operators. Several logics have been proposed for the `call/cc` operator, which captures undelimited, multi-shot (reusable) continuations. However, these logics are rather *ad hoc* and difficult to use. In contrast, the PhD work of Paulo de Vilhena, supervised by François Pottier, develops elegant separation logic rules for the user-defined effects and handlers of OCaml 5, which implement delimited, one-shot (affine) continuations.

What about undelimited, one-shot continuations, as captured by the `call/1cc` operator of Bruggeman, Waddell and Dybvig? Xavier Leroy and Paulo de Vilhena found that `call/1cc` admits a simple reasoning rule in separation logic. Furthermore, this rule is similar to the rule for `goto` jumps in Hoare logic. Xavier Leroy and Paulo de Vilhena proved the soundness of this `call/1cc` rule in two different ways, one using the Iris framework and the other using a basic separation logic built directly on a continuation-based transition semantics. This work has not yet been published.

7 Bilateral contracts and grants with industry

7.1 Bilateral grants with industry

7.1.1 The Caml Consortium

Participants: Damien Doligez.

The Caml Consortium is a formal structure where industrial and academic users of OCaml can support the development of the language and associated tools, express their specific needs, and contribute to the long-term stability of OCaml. Membership fees are used to fund specific developments targeted towards industrial users. Members of the Consortium automatically benefit from very liberal licensing conditions on the OCaml system, allowing for instance the OCaml compiler to be embedded within proprietary applications.

Damien Doligez chairs the Caml Consortium.

The Consortium currently has 5 member companies:

- Esterel / ANSYS
- Facebook
- Jane Street
- LexiFi
- SimCorp

One might think that the Caml Consortium could be superseded by the OCaml Software Foundation, discussed below. However, the Caml Consortium remains alive, because it is able to offer special licensing conditions. The above companies still need the Consortium's license. Most of them are also sponsors of the OCaml Foundation.

7.1.2 The OCaml Software Foundation

Participants: Damien Doligez, Xavier Leroy.

The **OCaml Software Foundation**, established in 2018 under the umbrella of the Inria Foundation, aims to promote, protect, and advance the OCaml programming language and its ecosystem, and to support and facilitate the growth of a diverse and international community of OCaml users.

Damien Doligez and Xavier Leroy serve as advisors on the foundation's Executive Committee.

We receive substantial basic funding from the OCaml Software Foundation in order to support research activity related to OCaml.

8 Partnerships and cooperations

8.1 National initiatives

Participants: Xavier Leroy, François Pottier, Jean-Marie Madiot, Remy Seassau, Tiago Soares.

The ANR GOSPEL project (2023–2026) involves four main participants, namely Inria Paris (team Cambium), LMF (Saclay), Tarides, Nomadic Labs, and two invited researchers, namely Arthur Charguéraud (Inria Strasbourg) and Mário Pereira (NOVA LINCS University, Lisbon). François Pottier is the head of the project. The total budget of the project is approximately 480K€, out of which the Cambium team receives 157K€.

The aim of this project is define Gospel, a standard specification language for OCaml, and to develop tools that can read Gospel specifications and connect them with formal program verification environments, such as Osiris. At Cambium, this project funds Remy Seassau’s PhD thesis, whose aim is to develop Osiris (§6.5.4) and to connect it with Gospel. Tiago Soares, who is funded by a Portuguese grant, also works on Gospel.

9 Dissemination

Participants: Yannick Forster, François Pottier, Yannick Zakowski, Jean-Marie Madiot, Xavier Leroy, Luc Maranget, Didier Remy, Remy Seassau, Tiago Lopes Soares, Samuel Vivien.

9.1 Promotion of scientific activities

9.1.1 Organisation of scientific events

Participants: Yannick Forster, François Pottier, Yannick Zakowski.

Yannick Forster was the senior co-chair of the Programming Languages Mentoring Workshop (PLMW) co-located with POPL 2025 in Denver. Yannick Forster is an ex-officio member of the steering committee of ITP.

Along with Philippe Aghion, Nalini Anantharaman, Patrick Boucheron and Stéphane Mallat, Xavier Leroy co-organized the October 2025 Colloquium of Collège de France, whose theme was *Formes de l’intelligence*.

François Pottier was the local organizer and the program chair of the workshop Iris 2025, which took place at Inria Paris in June 2025.

Yannick Zakowski served as webchair for POPL 2026, general chair for CPP 2026, vice president for JFLA 2026.

9.1.2 Participation in program committees

Participants: Yannick Forster, Jean-Marie Madiot, François Pottier, Yannick Zakowski.

Yannick Forster was the co-chair of the program committee for the International Conference on Interactive Theorem Proving (ITP 2025). Yannick Forster was also a member for the program committees for the conferences PLDI 2025 and CICM 2025.

Jean-Marie Madiot was a member of the program committee for the conference PLDI 2025.

François Pottier was a member of the program committee for the conference JFLA 2026 and for the workshop RocqPL 2026.

Yannick Zakowski was a member of the program committees for the conferences JFLA 2025, OOPSLA 2025, and PLDI 2026. Yannick Zakowski served as Artifact Evaluation Chair for the conferences ECOOP 2025 and 2026.

9.1.3 Participation in editorial boards

Participants: Xavier Leroy.

Xavier Leroy is a member of the editorial board of the Journal of Automated Reasoning. He is also a member of the Conseil Scientifique (scientific advisory board) of the Arte TV channel.

9.1.4 Invited talks

Participants: Yannick Forster.

Yannick Forster has given an invited talk about synthetic computability theory at the 33rd EACSL Annual Conference on Computer Science Logic (CSL 2025) and at the Workshop on Logical Frameworks and Meta Languages: Theory and Practice (LFMTP 2025).

9.1.5 Research administration

Participants: François Pottier, Luc Maranget, Didier Remy.

Luc Maranget is an elected member of Inria's *Commission d'évaluation* (CE). In particular, he took part in two hiring committees for *chargé de recherche* positions. Luc Maranget represents the Cambium team in the *Comité des utilisateurs des moyens informatiques* (CUMI).

François Pottier is the president of Inria Paris's *Comité de Suivi Doctoral* (CSD).

François Pottier is Inria's delegate in the pedagogical board of **MPRI**.

Didier Rémy is Inria's delegate in the management board of **MPRI**.

9.2 Teaching - Supervision - Juries

9.2.1 Teaching

Participants: Clement Allain, Yannick Forster, Xavier Leroy, Jean-Marie Madiot, François Pottier, Didier Remy, Tiago Lopes Soares, Samuel Vivien, Remy Seassau.

This year, the members of our team have taught or assisted in teaching the following courses:

- Licence (L1): *Programmation fonctionnelle pour le Web*, Clément Allain, 24 HETD, Université Paris Cité, France.
- Master (M2): *Proof assistants*, Yannick Forster, 18 HETD, MPRI, Université Paris Cité, France.
- Open lectures: *Le calcul sécurisé: calculer sur des données chiffrées ou privées*, Xavier Leroy, 16 HETD, Collège de France, France.

- Licence (L3): *Principles of programming languages*, Jean-Marie Madiot, 40 HETD, École Polytechnique, France.
- Master (M2): *Proofs of Programs* (course teacher and coordinator), Jean-Marie Madiot, 18 HETD, MPRI, Université Paris Cité, France.
- Master (M2): *Functional programming and type systems*, François Pottier, 18 HETD, MPRI, Université Paris Cité, France.
- Master (M2): *Functional programming and type systems*, Didier Remy, 3 HETD, MPRI, Université Paris Cité, France.
- Licence (L1): *Séminaire CAML*, Remy Seassau, 48 HETD, EPITA, France.
- Licence (L1): *Introduction to Programming*, Tiago Soares, 72 HETD, NOVA School of Science and Technology - UNL, Portugal.
- Licence (L1): *Initiation à la programmation 2*, Samuel Vivien, 24 HETD, Université Paris Cité, France.
- Licence (L3): *Langages de programmation et compilation*, Samuel Vivien, 40 HETD, ENS-PSL, France.
- License (L3): *Séminaire du département d'informatique de l'ENS de Lyon*, Yannick Zakowski, 32 HETD, ENS de Lyon France.
- Master (M1): *Compilation and Program Analysis*, Yannick Zakowski (with Gabriel Radanne), 18 HETD, ENS de Lyon France.
- Master (M2): *Functional programming and type systems*, Yannick Zakowski, 15 HETD, MPRI, Université Paris Cité, France.

9.2.2 Supervision

Participants: François Pottier, Yannick Forster, Yannick Zakowski, Didier Remy, Jean-Marie Madiot.

The following PhD theses are in progress or have been defended in 2025:

- PhD (completed): Clément Allain, *Parallel programming infrastructure for OCaml 5*, Université Paris Cité, since October 2022, advised by Gabriel Scherer and François Pottier; defended on December 16, 2025 [30].
- PhD (in progress): Mathis Bouverot-Dupuis, *Meta-programming with Scope Guarantees*, ENS Paris, since September 2025, advised by Yannick Forster and François Pottier.
- PhD (in progress): Timothée Huneau, *Unified Classical and Constructive Reverse Mathematics for Model Theory*, ENS Paris-Saclay, since October 2025, advised by Yannick Forster, Dominik Kirst, and Sam van Gool.
- PhD (in progress): Emma Nardino, *Formal Verification of Program Optimizations for the Async/Await Programming Model*, ENS Lyon, since September 2024, advised by Ludovic Henrio and Yannick Zakowski.
- PhD (in progress): Remy Seassau, *Developing a Specification Language and a Program Verification Framework for OCaml*, since October 2023, advised by François Pottier.
- PhD (in progress): Tiago Soares, *Verifying OCaml Programs With Exceptions and Control Effects*, since September 2023, advised by Mário Pereira (NOVA University, Lisboa) and François Pottier.

- PhD (in progress): Samuel Vivien, *Theory and practice of modular implicits: design, formalisation, implementation, and applications*, since October 2025, advised by Didier Rémy and Gabriel Scherer.
- PhD (in progress): Poyraz Yilan, *Certified compilation of critical parallel real-time programs*, Sorbonne Université, CIFRE with Airbus, since December 2025, advised by Dumitru Potop-Butucaru and Jean-Marie Madiot.

9.2.3 Juries

Participants: Yannick Forster, Xavier Leroy, François Pottier, Didier Remy, Yannick Zakowski.

Yannick Forster was a member of the jury for the PhD thesis of Zhuo Chen (University of Melbourne; October 2025).

Xavier Leroy was a member of the jury for the PhD thesis of Josué Moreau (Université Paris Saclay; November 2025).

François Pottier was a reviewer for the PhD theses and habilitation theses of Ike Mulder (Radboud University; February 2025), Simon Spies (Saarland University; May 2025), Long Pham (Carnegie-Mellon University; July 2025), and Matthieu Lemerre (HDR; Université Paris-Saclay; November 2025).

Didier Rémy was a reviewer for the PhD thesis of Guillaume Duboc (Université Paris Cité; January 2025).

Yannick Zakowski was a reviewer for the PhD thesis of Sergei Stepanenko (Aarhus University; October 2025).

9.3 Popularization

Participants: Xavier Leroy.

Xavier Leroy gave an overview talk on deductive software verification at the CS Colloquium of Chalmers university (Göteborg, Sweden, October 2025).

10 Scientific production

10.1 Major publications

- [1] J. Alglave, W. Deacon, R. Grisenthwaite, A. Hacquard and L. Maranget. ‘Armed Cats: formal concurrency modelling at Arm’. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 43 (July 2021), pp. 1–54. DOI: [10.1145/3458926](https://doi.org/10.1145/3458926). URL: <https://hal.inria.fr/hal-03470858>.
- [2] C. Allain and G. Scherer. ‘Zoo: A Framework for the Verification of Concurrent OCaml 5 Programs using Separation Logic’. In: *Proceedings of the ACM on Programming Languages* 10.POPL (8th Jan. 2026), pp. 1702–1729. DOI: [10.1145/3776701](https://doi.org/10.1145/3776701). URL: <https://inria.hal.science/hal-05467809>.
- [3] M. Baillon, Y. Forster, A. Mahboubi, P.-M. Pédrot and M. Piquerez. ‘A Zoo of Continuity Properties in Constructive Type Theory’. In: 10th International Conference on Formal Structures for Computation and Deduction (FSCD 2025). Birmingham, France: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. DOI: [10.4230/LIPIcs.FSCD.2025.9](https://doi.org/10.4230/LIPIcs.FSCD.2025.9). URL: <https://inria.hal.science/hal-04908282>.
- [4] C. Blaudeau, D. Rémy and G. Radanne. ‘Avoiding Signature Avoidance in ML Modules with Zippers’. In: *Proceedings of the ACM on Programming Languages* POPL.9 (22nd Jan. 2025). DOI: [10.1145/3704902](https://doi.org/10.1145/3704902). URL: <https://inria.hal.science/hal-04801582>.

- [5] N. Courant and X. Leroy. ‘A Lazy, Concurrent Convertibility Checker’. In: *Proceedings of the ACM on Programming Languages* 10.POPL (8th Jan. 2026), 53:1–53:27. DOI: [10.1145/3776695](https://doi.org/10.1145/3776695). URL: <https://hal.science/hal-05322705>.
- [6] P. Emílio De Vilhena and F. Pottier. ‘A Separation Logic for Effect Handlers’. In: *Proceedings of the ACM on Programming Languages* (Jan. 2021). DOI: [10.1145/3434314](https://doi.org/10.1145/3434314). URL: <https://hal.inria.fr/hal-03049514>.
- [7] J.-M. Madiot, D. Pous and D. Sangiorgi. ‘Modular coinduction up-to for higher-order languages via first-order transition systems’. In: *Logical Methods in Computer Science* Volume 17, Issue 3 (17th Sept. 2021). DOI: [10.46298/lmcs-17\(3:25\)2021](https://doi.org/10.46298/lmcs-17(3:25)2021). URL: <https://hal.archives-ouvertes.fr/hal-03350199>.
- [8] A. Moine, A. Charguéraud and F. Pottier. ‘Will it Fit? Verifying Heap Space Bounds of Concurrent Programs under Garbage Collection’. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* (10th Feb. 2025). DOI: [10.1145/3716312](https://doi.org/10.1145/3716312). URL: <https://inria.hal.science/hal-04946590>.
- [9] A. Raad, L. Maranget and V. Vafeiadis. ‘Extending Intel-x86 Consistency and Persistency: Formalising the Semantics of Intel-x86 Memory Types and Non-Temporal Stores’. In: *POPL 2022 - Symposium on Principles of Programming Languages*. Philadelphia, United States, 16th Jan. 2022. DOI: [10.1145/3498683](https://doi.org/10.1145/3498683). URL: <https://hal.inria.fr/hal-03426997>.
- [10] R. Seassau, I. Yoon, J.-M. Madiot and F. Pottier. ‘Formal Semantics and Program Logics for a Fragment of OCaml’. In: *Proceedings of the ACM on Programming Languages* 9.ICFP (5th Aug. 2025), pp. 128–159. DOI: [10.1145/3747509](https://doi.org/10.1145/3747509). URL: <https://inria.hal.science/hal-05423292>.

10.2 Publications of the year

International journals

- [11] C. Allain, F. Bour, B. Clément, F. Pottier and G. Scherer. ‘Tail Modulo Cons, OCaml, and Relational Separation Logic’. In: *Proceedings of the ACM on Programming Languages* 9.POPL (9th Jan. 2025), pp. 2337–2363. DOI: [10.1145/3704915](https://doi.org/10.1145/3704915). URL: <https://inria.hal.science/hal-04884634>.
- [12] C. Allain and G. Scherer. ‘Zoo: A Framework for the Verification of Concurrent OCaml 5 Programs using Separation Logic’. In: *Proceedings of the ACM on Programming Languages* 10.POPL (8th Jan. 2026), pp. 1702–1729. DOI: [10.1145/3776701](https://doi.org/10.1145/3776701). URL: <https://inria.hal.science/hal-05467809> (cit. on p. 23).
- [13] C. Blaudeau, D. Rémy and G. Radanne. ‘Avoiding Signature Avoidance in ML Modules with Zippers’. In: *Proceedings of the ACM on Programming Languages* POPL.9 (22nd Jan. 2025). DOI: [10.1145/3704902](https://doi.org/10.1145/3704902). URL: <https://inria.hal.science/hal-04801582>.
- [14] N. Courant and X. Leroy. ‘A Lazy, Concurrent Convertibility Checker’. In: *Proceedings of the ACM on Programming Languages* 10.POPL (8th Jan. 2026), 53:1–53:27. DOI: [10.1145/3776695](https://doi.org/10.1145/3776695). URL: <https://hal.science/hal-05322705> (cit. on p. 21).
- [15] A. L. Georges, B. Peters, L. Elbeheiry, L. White, S. Dolan, R. A. Eisenberg, C. Casinhino, F. Pottier and D. Dreyer. ‘Data Race Freedom à la Mode’. In: *Proceedings of the ACM on Programming Languages* 9.POPL (9th Jan. 2025), pp. 656–686. DOI: [10.1145/3704859](https://doi.org/10.1145/3704859). URL: <https://inria.hal.science/hal-04884654>.
- [16] E. Ioannidis, Y. Zakowski, S. Zdancewic and S. Angel. ‘Structural Temporal Logic for Mechanized Program Verification’. In: *Proceedings of the ACM on Programming Languages* 9.OOPSLA2 (9th Oct. 2025), pp. 1148–1175. DOI: [10.1145/3763091](https://doi.org/10.1145/3763091). URL: <https://hal.science/hal-05308634> (cit. on p. 19).
- [17] X. Leroy. ‘Sciences du logiciel: [résumé des cours et travaux : 2021-2022]’. In: *L’Annuaire du Collège de France. Résumés des cours et travaux* 122 (2026), pp. 21–29. DOI: [10.4000/15e09](https://doi.org/10.4000/15e09). URL: <https://college-de-france.hal.science/hal-05455792>.

- [18] A. Moine, A. Charguéraud and F. Pottier. ‘Will it Fit? Verifying Heap Space Bounds of Concurrent Programs under Garbage Collection’. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* (10th Feb. 2025). DOI: [10.1145/3716312](https://doi.org/10.1145/3716312). URL: <https://inria.hal.science/hal-04946590>.
- [19] R. Seassau, I. Yoon, J.-M. Madiot and F. Pottier. ‘Formal Semantics and Program Logics for a Fragment of OCaml’. In: *Proceedings of the ACM on Programming Languages 9.ICFP* (5th Aug. 2025), pp. 128–159. DOI: [10.1145/3747509](https://doi.org/10.1145/3747509). URL: <https://inria.hal.science/hal-05423292> (cit. on p. 24).

International peer-reviewed conferences

- [20] C. Allain. ‘Zoo: A framework for the verification of concurrent OCaml 5 programs using separation logic’. In: 36es Journées Francophones des Langages Applicatifs (JFLA 2025). Roiffé, France, 28th Jan. 2025. URL: <https://inria.hal.science/hal-04868573>.
- [21] M. Baillon, Y. Forster, A. Mahboubi, P.-M. Pédrot and M. Piquerez. ‘A Zoo of Continuity Properties in Constructive Type Theory’. In: 10th International Conference on Formal Structures for Computation and Deduction (FSCD 2025). Birmingham, France: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. DOI: [10.4230/LIPIcs.FSCD.2025.9](https://doi.org/10.4230/LIPIcs.FSCD.2025.9). URL: <https://inria.hal.science/hal-04908282> (cit. on p. 20).
- [22] F. Bour and F. Pottier. ‘Mapping and explaining syntax errors with LRgrep’. In: JFLA 2026 – 37es Journées Francophones des Langages Applicatifs. Vol. JFLA 2026 – 37es Journées Francophones des Langages Applicatifs. Oberbronn, France, 27th Jan. 2026. URL: <https://hal.science/hal-05428164> (cit. on p. 18).
- [23] L. Henrio, E. Broch Johnsen, Å. A. A. Kløvstad, V. Ka I Pun and Y. Zakowski. ‘Layers of Confluence for Actors’. In: CPP 2026 - Certified Programs and Proofs. Rennes, France, 12th Jan. 2026. DOI: [10.1145/3779031.3779104](https://doi.org/10.1145/3779031.3779104). URL: <https://hal.science/hal-05424264> (cit. on p. 19).
- [24] J. Ponsonnet and F. Pottier. ‘Verified Persistent Catenable Deques’. In: JFLA 2026 – 37es Journées Francophones des Langages Applicatifs. Vol. JFLA 2026 – 37es Journées Francophones des Langages Applicatifs. Oberbronn, France, 27th Jan. 2026. URL: <https://hal.science/hal-05427954> (cit. on p. 23).
- [25] S. Vivien, D. Rémy and G. Scherer. ‘On the design and implementation of Modular Explicits’. In: JFLA 2026 – 37es Journées Francophones des Langages Applicatifs. Vol. JFLA 2026 – 37es Journées Francophones des Langages Applicatifs. Oberbronn, France, 27th Jan. 2026. URL: <https://hal.science/hal-05428136> (cit. on p. 16).

Conferences without proceedings

- [26] M. Bouverot-Dupuis, T. Winterhalter, K. Stark and K. Maillard. ‘Sulfur: a Reflective Tactic for Substitution Simplification’. In: RocqPL 2026 - Rocq for Programming Languages. Rennes, France, 2026. URL: <https://hal.science/hal-05482084>.
- [27] J. Caspar and G. Munch-Maccagnoni. ‘S4 modal sequent calculus as intermediate logic and intermediate language’. In: PEPM 2026 - ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation. Rennes, France: ACM, 2026. URL: <https://inria.hal.science/hal-05430766>.
- [28] A. O’Brien, D. Rémy and G. Scherer. ‘Omnidirectional type inference for ML: 5th Workshop on the Implementation of Type Systems’. In: WITS 2026. 5th Workshop on the Implementation of Type Systems. Rennes, France, 17th Jan. 2026. URL: <https://inria.hal.science/hal-05438513> (cit. on p. 17).
- [29] S. Vivien, D. Rémy and G. Scherer. ‘Implicit modules, a middle step towards modular implicits’. In: Higher-order, Typed, Inferred, Strict: ML Family Workshop 2025. Singapore, Singapore, 16th Oct. 2025. URL: <https://inria.hal.science/hal-05444317> (cit. on p. 17).

Doctoral dissertations and habilitation theses

- [30] C. Allain. ‘Verification of fine-grained concurrent OCaml 5 algorithms using separation logic’. Université Paris Cité, 17th Dec. 2025. URL: <https://inria.hal.science/tel-05467850> (cit. on pp. 23, 28).

Reports & preprints

- [31] M. Bouverot-Dupuis and Y. Forster. *Code Generation via Meta-programming in Dependently Typed Proof Assistants*. 20th Jan. 2026. URL: <https://hal.science/hal-05024207> (cit. on p. 21).
- [32] T. Hervier. *MTrees : Mixing monadic effects and Labelled Transition Systems in Rocq*. ENS de Lyon, 26th June 2025. URL: <https://hal.science/hal-05140642> (cit. on p. 19).
- [33] T. Lamiaux, Y. Forster, M. Sozeau and N. Tabareau. *Nested Inductive Types: Justified and Usable Nested Inductive Types in Lean and Rocq*. 2025. URL: <https://hal.science/hal-05366368> (cit. on p. 22).
- [34] X. Leroy. *The CompCert C verified compiler: Documentation and user’s manual: Version 3.16*. Inria, 1st Sept. 2025, pp. 1–79. URL: <https://inria.hal.science/hal-01091802>.
- [35] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, K. Sivaramakrishnan and J. Vouillon. *The OCaml system release 5.4: Documentation and user’s manual*. Inria, 9th Oct. 2025, p. 1107. URL: <https://inria.hal.science/hal-00930213> (cit. on p. 7).
- [36] E. Nardino, L. Henrio, G. Radanne and Y. Zakowski. *Tail Modulo Async-Await*. 2025. URL: <https://hal.science/hal-05006570> (cit. on p. 18).
- [37] A. O’Brien, D. Rémy and G. Scherer. *Omnidirectional type inference for ML: principality any way*. University of Cambridge, INRIA, IRIF, Université Paris Cité, 2025. DOI: [10.48550/arXiv.2511.10343](https://doi.org/10.48550/arXiv.2511.10343). URL: <https://inria.hal.science/hal-05438544> (cit. on p. 17).
- [38] Y.-J. Tan. *Towards Formalising the Guard Checker of Coq*. Ecole polytechnique; Inria - Paris, 9th Mar. 2025. URL: <https://inria.hal.science/hal-04983786>.
- [39] H. Zeng, Y. Forster, D. Kirst and T. Nemoto. *Post’s Problem in Constructive Mathematics*. 31st Jan. 2025. URL: <https://inria.hal.science/hal-04923365> (cit. on p. 21).

10.3 Cited publications

- [40] T. Balabonski, F. Pottier and J. Protzenko. ‘The design and formalization of Mezzo, a permission-based programming language’. In: *ACM Transactions on Programming Languages and Systems* 38.4 (2016), 14:1–14:94. URL: <http://doi.acm.org/10.1145/2837022> (cit. on p. 7).
- [41] J.-H. Jourdan, V. Laporte, S. Blazy, X. Leroy and D. Pichardie. ‘A Formally-Verified C Static Analyzer’. In: *POPL’15: 42nd ACM Symposium on Principles of Programming Languages*. ACM Press, Jan. 2015, pp. 247–259. URL: <http://dx.doi.org/10.1145/2676726.2676966> (cit. on p. 8).
- [42] C. Blaudeau, D. Rémy and G. Radanne. ‘Fulfilling OCaml Modules with Transparency’. In: *Proceedings of the ACM on Programming Languages* 8.OOPSLA1 (Apr. 2024), pp. 194–222. DOI: [10.1145/3649818](https://doi.org/10.1145/3649818). URL: <https://inria.hal.science/hal-04794404> (cit. on pp. 16, 17).
- [43] P. Borthelle. ‘Operational Game Semantics in Type Theory’. Theses. Université Savoie Mont Blanc, Mar. 2025. URL: <https://theses.hal.science/tel-05294140> (cit. on p. 19).
- [44] P. Borthelle, T. Hirschowitz, G. Jaber and Y. Zakowski. ‘An abstract, certified account of operational game semantics’. In: *European Symposium on Programming*. Vol. 15694. Hamilton, Ontario, Canada: Springer, 2025, pp. 172–199. DOI: [10.1007/978-3-031-91118-7_7](https://doi.org/10.1007/978-3-031-91118-7_7). URL: <https://hal.science/hal-04583895> (cit. on p. 19).
- [45] N. Chappe, P. He, L. Henrio, E. Ioannidis, Y. Zakowski and S. Zdancewic. ‘Choice Trees: Representing and Reasoning About Nondeterministic, Recursive, and Impure Programs in Rocq’. In: *Journal of Functional Programming* (Sept. 2025). URL: <https://hal.science/hal-05154458> (cit. on p. 18).

- [46] N. Chappe, L. Henrio and Y. Zakowski. ‘Monadic Interpreters for Concurrent Memory Models: Executable Semantics of a Concurrent Subset of LLVM IR’. In: *Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs*. Denver CO USA, France: ACM, Jan. 2025, pp. 283–298. DOI: [10.1145/3703595.3705890](https://doi.org/10.1145/3703595.3705890). URL: <https://hal.science/hal-04594073> (cit. on p. 18).
- [47] [SW] V. Karvonen and C. Morel, *Saturn* 2024. URL: <https://github.com/ocaml-multicore/saturn> (cit. on p. 23).
- [48] L. Lamport. ‘How to write a 21st century proof’. In: *Journal of Fixed Point Theory and Applications* 11 (1 2012), pp. 43–63. URL: <http://dx.doi.org/10.1007/s11784-012-0071-6> (cit. on p. 16).
- [49] X. Leroy. ‘Formal verification of a realistic compiler’. In: *Communications of the ACM* 52.7 (2009), pp. 107–115. URL: <http://doi.acm.org/10.1145/1538788.1538814> (cit. on pp. 8, 14).
- [50] [SW] A. Madhavapeddy and T. Leonard, *Eio* 2025. URL: <https://github.com/ocaml-multicore/eio> (cit. on p. 23).
- [51] T. L. Soares. ‘Keep Singing the Gospel’. In: *JFLA 2026 – 37es Journées Francophones des Langages Applicatifs*. Vol. JFLA 2026 – 37es Journées Francophones des Langages Applicatifs. Marie Kerjean and Yannick Zakowski. Oberbronn, France, Jan. 2026. URL: <https://hal.science/hal-05428147> (cit. on p. 24).
- [52] L. White, F. Bour and J. Yallop. ‘Modular implicits’. In: *Proceedings ML Family/OCaml Users and Developers workshops, ML/OCaml 2014, Gothenburg, Sweden, September 4-5, 2014*. Ed. by O. Kiselyov and J. Garrigue. Vol. 198. EPTCS. 2014, pp. 22–63. DOI: [10.4204/EPTCS.198.2](https://doi.org/10.4204/EPTCS.198.2). URL: <https://doi.org/10.4204/EPTCS.198.2> (cit. on p. 16).