RESEARCH CENTRE

Inria Paris Centre

IN PARTNERSHIP WITH:

Collège de France

2023

ACTIVITY REPORT

Project-Team

CAMBIUM

Programming languages: type systems, concurrency, proofs of programs

DOMAIN

Algorithmics, Programming, Software and Architecture

THEME

Proofs and Verification

Contents

Project-Team CAMBIUM

Creation of the Project-Team: 2019 August 01

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

A2.1.3. – 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

A2.4. – Formal method for verification, reliability, certification

A2.4.1. – Analysis

A2.4.3. – Proofs

A2.5.4. – Software Maintenance & Evolution

A7.1.2. – Parallel algorithms

A7.2. – Logic in Computer Science

A7.2.2. – Automated Theorem Proving

A7.2.3. – Interactive Theorem Proving

Other research topics and application domains

B5.2.3. – Aviation

B6.1. – Software industry

B6.6. – Embedded systems

B9.5.1. – Computer science

1 Team members, visitors, external collaborators

Research Scientists

- François Pottier [Team leader, INRIA, Senior Researcher, HDR]
- Damien Doligez [INRIA, Researcher]
- Yannick Forster [INRIA, Researcher, from Dec 2023]
- Jean-Marie Madiot [INRIA, Researcher]
- Luc Maranget [INRIA, Researcher]
- Didier Rémy [INRIA, Senior Researcher, HDR]

Faculty Member

- Xavier Leroy [COLLEGE DE FRANCE, Professor]

PhD Students

- Clément Allain [INRIA]
- Clément Blaudeau [UNIV PARIS]
- Frédéric Bour [TARIDES]
- Paulo De Vilhena [INRIA, until Jan 2023]
- Tiago Lopes Soares [UNIV NOVA LISBONNE, from Oct 2023]
- Alexandre Moine [INRIA]
- Thomas Refis [TARIDES, until Feb 2023]
- Remy Seassau [INRIA, from Oct 2023]

Interns and Apprentices

- Arnaud Daby-Seesaram [ENS PARIS-SACLAY, Intern, from Mar 2023 until Aug 2023]
- Samuel Vivien [ENS PARIS, Intern, from Sep 2023]

Administrative Assistant

- Hélène 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 and the CompCert C compiler 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 [23], 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 [22]. 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 [28] and the Verasco abstract interpretation-based static analyzer [24]. 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 Charguéraud [1] has developed CFML, an implementation of Separation Logic for a subset of OCaml. Armaël Guéneau[2] has extended CFML with the ability to simultaneously verify the correctness and the time complexity of an OCaml component. Glen Mével[3] 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.

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

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

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

[3]A former student in our team.

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. (GPUs 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 been clearly defined, both on paper and in Coq. Also since OCaml 5, the language has been extended with 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.

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.

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 Highlights of the year

5.1 Awards

In March 2023, Sandrine Blazy, Zaynah Dargaye and Xavier Leroy received the Lucas Award from Formal Methods Europe for a highly influential paper published at the conference Formal Methods 2006.

In May 2023, Paulo Emílio de Vilhena received the 2022 best dissertation award from GDR GPL, a network of French research labs in programming languages and software engineering.

In June 2023, Xavier Leroy was elected a member of Académie des Sciences.

In October 2023, the OCaml language and system received one of the eight *prix science ouverte du logiciel libre de la recherche 2023* (open science award for free software) from the French Ministry for Higher Education and Research.

5.2 Recruitment

As of January 1st, 2023, Florian Angeletti is recruited as an Inria engineer on a permanent position. He plays a key role in the development and release cycle of the OCaml compiler and in animating the OCaml developer community.

As of December 1st, 2023, Yannick Forster joins the team as a junior researcher (*chargé de recherche*).

6 New software, platforms, open data

6.1 New software

6.1.1 OCaml

Keywords: Functional programming, Static typing, Compilation

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-03146495, hal-03510931, hal-03145030, hal-01929508, hal-03125031, hal-00772993, hal-00914493, hal-00914560, inria-00074804, hal-01499973, hal-01499946

Contact: Damien Doligez

Participants: Florian Angeletti, Damien Doligez, Xavier Leroy, Luc Maranget, Gabriel Scherer, Alain Frisch, Jacques Garrigue, Marc Shinwell, Jeremy Yallop, Leo White

6.1.2 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

Participants: Xavier Leroy, Sandrine Blazy, Jacques-henri Jourdan, Sylvie Boldo, Guillaume Melquiond

Partner: AbsInt Angewandte Informatik GmbH

6.1.3 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

Participants: Jade Alglave, Luc Maranget

Partner: University College London UK

6.1.4 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

6.1.5 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

Participants: Arthur Charguéraud, Armael Guéneau, François Pottier

6.1.6 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

Participants: Damien Doligez, Stephan Merz, Ioannis Filippidis

Partner: Microsoft

6.1.7 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: Damien Doligez

6.1.8 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/`

Author: Luc Maranget

Contact: Luc Maranget

7 New results

7.1 Long-term software projects

7.1.1 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 [28]. It comprises a back-end, which translates the Cminor intermediate language to PowerPC assembly and which can be reused for source languages other than C [27], and a front-end, which translates the "CompCert C" subset of C to Cminor. The compiler is written mostly within the specification language of the Coq proof assistant, out of which Coq'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 improved the speed and compactness of the generated code through

- improvements to the value analysis, with a better distinction between "any integer value" and "any integer or pointer value";
- better recognition of conditional expressions that can be "if-converted" into a conditional move;
- generation of more compact ARM Thumb2 code, in particular by favoring the use of registers R0–R3.

We also improved ABI compatibility concerning the definition of the type `wchar_t`, and fixed minor issues involving variadic function definitions. These improvements were released in July 2023 as part of CompCert version 3.13.

7.1.2 The OCaml system

Participants: Florian Angeletti, Damien Doligez, Sébastien Hinderer, Xavier Leroy, Luc Maranget, Thomas Refis, David Allsop *(Tarides)*, Enguerrand Decorne *(Tarides)*, Stephen Dolan *(Jane Street LLC)*, 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 Partout)*, Leo White *(Jane Street LLC)*.

This year, we have released one minor version of OCaml, namely OCaml 5.1.0, on September 9, 2023. Some of the highlights of this release are:

- Many runtime performance regression and memory-leak fixes (dynlinking, weak array, weak hash sets, GC with idle domains, GC prefetching).
- Restored support for native code generation on RISC-V and s390x architectures.
- Restored Cygwin port.
- Reduced installation size (a 50% reduction).
- Compressed compilation artefacts (.cmi, .cmt, .cmti, .cmo, .cma files).
- 19 error message improvements.
- 14 standard library functions made tail-recursive thanks to Tail-Modulo-Cons (TMC) optimization.
- 57 new standard library functions.
- More examples in the documentation of the standard library.

- 42 bug fixes.

The initial release of OCaml 5.1 was followed on December 8 by a patch release, OCaml 5.1.1, which fixed significant bugs concerning the type-checker and the performance of the GC.

We also worked on a new port for the POWER architecture and on a new memory compactor that can give unused memory back to the operating system. These results will be made available next year in OCaml 5.2.0.

A significant portion of the development focus for 5.1.0 has been on the maturation and stabilisation of the new OCaml 5 runtime system. There is still much work in progress in this area.

Meanwhile, the OCaml 4.14 branch has been maintained as a stable version of the OCaml 4 compiler. The next release on this branch, OCaml 4.14.2, is in preparation, and will be released in the coming months.

This year, OCaml received a prize for open-science research software from the French ministry of Research and Education (§5.1).

7.1.3 The `diy` tool suite

Participants: Hadrien Renaud *(University College London)*, Artem KhyzhaARM Ltd. , Nikos NikorelisARM 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, Luc Maranget extended the tools mostly by implementing new instructions for various architectures, mostly for ARMv8 including their alternative implementation on top of Hadrien Renaud's interpreter of the instruction definition language. More generally, Luc Maranget acts as the main maintainer and coordinator of the toolbox: he reviews and validates the pull requests submitted by contributors, mostly ARM engineers. Nikos Nikorelis is cited for his significant contribution to the combination of virtual memory and tagged memory, and Artem Khyzha for self-modifying code. These are follow-ups on work initiated by Luc Maranget.

7.1.4 Menhir

Participants: François Pottier.

The parser generator Menhir has been extended with new features that aim to facilitate unparsing, that is, transforming abstract syntax trees into text. This is non-trivial because our method supports non-LR(1) grammars decorated with precedence declarations and guarantees correct unparsing: that is, parentheses or other disambiguation symbols are inserted where necessary. Furthermore, we allow the user to control other aspects of the unparsing process, such as layout. Our aproach is described in a paper that has been accepted for publication at JFLA 2024 [17].

7.1.5 TLAPS

Participants: Rosalie Defourné *(Inria team Veridis)*, Damien Doligez, Igor Konnov *(Informal Systems)*, Markus Kuppe *(Microsoft Research)*, Leslie Lamport *(Microsoft Research)*, Stephan Merz *(Inria team Veridis)*.

This project produces and maintains tools for managing and verifying proofs in the proof language of TLA+ [26].

Despite slowing down due to the end of the Microsoft-INRIA Joint Centre, the project goes on. TLA+ is attracting more and more users, some of whom are interested in the proof language. Consequently, we have switched to an open-source development model for the proof tools, in the hope of attracting external contributors.

7.2 Programming language design and implementation

7.2.1 Formalizing and improving OCaml modules

Participants: Clément Blaudeau, Didier Rémy, Gabriel Radanne *(Inria team Cash)*.

We continued the formalization of modules that started in 2021 by revisiting, improving, and adapting *F-ing modules* [31] to OCaml.

This year, we focused on the treatment of applicative functors, which is more difficult than that of generative functors. First, we introduced *transparent existential types* into F^ω. This simplifies the encoding of applicative functors by allowing Skolemization of abstract types. Then, we developed a new notion of *module identity*, which models OCaml's granularity for applicativity, and we generalized OCaml's *module aliases* by introducing a notion of *transparent ascription*. This also helped us provide a (partial) reverse translation from inferred F^ω signatures back to OCaml-style, path-based signatures. This improves the treatment of the signature avoidance problem in OCaml.

This work has been conditionally accepted (modulo minor revisions) for presentation at OOPSLA 2024. Preliminary investigations for refactoring the OCaml type-checker, based on the ideas developed in the paper, have been conducted on a prototype implementation.

While abstract signatures, a peculiarity of OCaml, have been left out of our encoding, we proposed a restriction of abstract signatures that covers most useful cases of abstract signatures and that can be given a simple and intuitive semantics by elaboration into F^ω. This has been summarized in a blog post.

7.2.2 Designing and formalizing modular implicits

Participants: Samuel Vivien, Didier Rémy.

Modular implicits are modules that are passed as implicit parameters to functions. Our work on modular implicits is based on a proposal and prototype implementation by White, Bour and Yallop [32]. Extending OCaml with modular implicits seems desirable because OCaml's functors are extremely verbose, preventing their use at a small scale. Modular implicits 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.

During a 6-month internship, Samuel Vivien, supervised by Didier Rémy, has conducted experiments with modular implicits on top of a new prototype implementation of modules in F^ω with partial type inference. We aim to explore and better specify the interaction between the core language and the module language. We seek both a predictable, robust specification and an efficient implementation of modular implicits. We started the formalization of the elaboration algorithm.

7.2.3 A declarative approach to LR(1) syntax error messages

Participants: Frédéric Bour, François Pottier.

`LRgrep` is a declarative language, and a supporting tool, whose purpose is to let the author of an LR(1) parser describe the syntax error messages that the parser should produce when the input is syntactically incorrect. The language allows matching the parser's stack against a specialized flavor of regular expressions. The design and implementation of `LRgrep` are the subject of Frédéric Bour's PhD work.

The problem presents different aspects: designing a convenient and expressive language to characterize error situations, efficiently compiling this language, and, finally, providing tools to assist with designing and maintaining error messages.

This year, Frédéric redesigned the language, so as to make it more practical, and formalized its new semantics. He optimized the compilation scheme so as to produce more compact automata, and improved the tools that support the language. He added a sentence enumeration feature, which produces a set of erroneous sentences that offers complete coverage (in a certain sense) of the automaton. This set of sentences allows testing a parser outside the set of correct sentences.

One important future application of `LRgrep`, which Frédéric has started to investigate, is to improve the syntax error messages procuced by the OCaml compiler. Furthermore, Frédéric has validated the approach by applying `LRgrep` to two other languages. He has built an experimental version of the Catala compiler that uses `LRgrep` to produce error messages. He has also written a parser for the Elm language that uses `LRgrep` and tries to mimic the error messages of the Elm compiler.

Frédéric has started writing his dissertation, where the research and design work behind `LRgrep` will be presented. Writing is in progress, and should be finished during the first half of 2024.

7.3 Semantics of shared-memory concurrency

7.3.1 Axiomatic memory models and virtual memory

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

Modern multi-core and multi-processor computers do not follow the intuitive "sequential consistency" model that would define a concurrent execution as the interleaving of the executions of its constituent threads and that would command instantaneous writes to the shared memory. This situation is due both to in-core optimisations such as speculative and out-of-order execution of instructions, and to the presence of sophisticated (and cooperating) caching devices between processors and memory. Jade Alglave and Luc Maranget have been collaborating in this domain for more than a decade.

Jade Alglave and Luc Maranget, with the help of ARM engineers, have completed the design of a significant extension of the ARM `aarch64.cat` memory model, namely virtual memory. The real-world relevance and potential impact of this work are high, as such a specification is necessary in order to write correct parallel operating systems. A submission to ASPLOS 2023 was unfortunately rejected. We are preparing a simplified and improved paper, which will be submitted to ISCA 2024.

In this project, Luc Maranget is specifically in charge of software development and of experiments. The amount of work involved (in particular, experimental work) is much more significant than in similar previous works, due to the numerous features of virtual memory, such as permissions, TLBs, alternative mappings of a single physical page, and so on.

7.3.2 Semantics 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. 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.

ARM develops a dedicated language for specifying the semantics of instructions. This year, Hadrien Renaud achieved significant progress in the definition and implementation of this language and of its type system. The implementation is quite involved, as the core mechanism that handles parallel execution is non-standard and must be integrated with the memory model simulator `herd`.

7.4 Program verification and mechanized mathematics

7.4.1 Revisiting well-founded recursion in Coq

Participants: Xavier Leroy.

Several Coq libraries and extensions support the definition of non-structural recursive functions using well-founded orderings for termination. While developing CompCert, we ran into some drawbacks of these existing approaches: function definitions can be hard to understand and review; proofs about these functions can require axioms such as function extensionality; extraction generates OCaml code that can be difficult to review. We recently realized that recursive functions can be defined quite simply by explicit structural induction on a proof of accessibility of the principal recursive argument. This back-to-the-basics approach works quite well, and we have started to use it liberally in CompCert. A "pearl" paper describing this work has been presented in January 2024 at the CoqPL workshop [19].

7.4.2 Verified extraction from Coq to OCaml

Participants: Yannick Forster, Matthieu Sozeau *(Inria team Gallinette)*, Nicolas Tabareau *(Inria team Gallinette)*.

One of the central claims of fame of the Coq proof assistant is extraction, that is, the ability to obtain efficient programs in industrial programming languages such as OCaml, Haskell, or Scheme from programs written in Coq's expressive dependent type theory. Extraction is of great practical usefulness; it plays a crucial role, for instance, in the CompCert project.

However, for such executables obtained by extraction, the extraction process is part of the trusted code base (TCB), as are Coq's kernel and the compiler used to compile the extracted code. The extraction process contains intricate semantic transformation of programs that rely on subtle operational features of both the source and target language. Its code has also evolved since its theoretical exposition in Pierre Letouzey's PhD thesis.

Furthermore, while the exact correctness statements for the execution of extracted code are described clearly in academic literature, the interoperability with unverified code has never been investigated formally; yet it is used in virtually every project that relies on extraction.

In this line of work, we develop a novel extraction pipeline from Coq to OCaml, implemented and verified in Coq itself, with a clear correctness theorem and guarantees for safe interoperability.

We build on the MetaCoq project, which aims at decreasing the TCB of Coq's kernel by re-implementing it in Coq itself and proving it correct with respect to a formal specification of Coq's type theory in Coq.

The main result of this line of work has been submitted for review [20]. Yannick Forster has also given an invited talk about this work at the conference TYPES 2023.

7.4.3 Synthetic computability theory in Coq

Participants: Yannick Forster, Dominik Kirst *(Ben-Gurion University, Israel)*, Felix Jahn *(Saarland University)*, Gert Smolka *(Saarland University)*, Niklas Mück *(MPI-SWS, Germany)*, Haoyi Zeng *(Saarland University, Germany)*.

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 Coq proof assistant.

Continuing a long line of work on synthetic computability in proof assistants, Yannick Forster has published results on reducibility theory, covering the Myhill Isomorphism theorem and Cantor Bernstein theorem, jointly with Felix Jahn and Gert Smolka, at the conference CPP 2023, and on simple and hypersimple sets, jointly with Felix Jahn, at the conference CSL 2023.

Yannick Forster has advised the Bachelor's thesis of Niklas Mück and is currently advising the Bachelor's thesis of Haoyi Zeng, both at Saarland University and both jointly with Dominik Kirst.

The projects have led to a definition of oracle computability in type theory, published at the conference APLAS 2023, and to a synthetic proof of Post's theorem concerning the arithmetical hierarchy, soon to appear at the conference CSL 2024, jointly with Dominik Kirst and Niklas Mück.

Yannick Forster has given an invited talk about this line at work in the special session of proof assistants at the conference MFPS 2023.

7.5 Program verification in separation logic

7.5.1 Verifying heap space bounds for concurrent programs under garbage collection

Participants: Alexandre Moine, Arthur Charguéraud, François Pottier.

For two decades, Separation Logic has been applied mainly to functional correctness proofs: that is, it has been used to prove that a program cannot crash and must eventually produce a correct result. It has also been extended with *time credits*, which endow it with the ability to reason about the time complexity of a program. In our group, this topic has been studied in previous years by François Pottier, Armaël Guéneau, and Glen Mével; see also §7.5.5.

In 2021, Jean-Marie Madiot and François Pottier presented a Separation Logic with Space Credits [29], which allows establishing verified space complexity bounds under garbage collection for an assembly-like language. In 2022, Alexandre Moine, Arthur Charguéraud and François Pottier scaled it up to a high-level sequential language [10].

This year, Alexandre Moine, Arthur Charguéraud and François Pottier further extended this work to a concurrent setting. The new logic involves new "pointed-by-thread" assertions, which keep track of the existence of stack-to-heap pointers in a reasonably lightweight manner. It also accounts for the subtleties of a stop-the-world garbage collector (GC), similar to the one found in OCaml 5. The programmer explicitly delimits sections of the code where the GC is *not* allowed to run, and the reasoning rules of the logic exploit this information to establish tighter space complexity bounds.

A first version of this work was presented at the Iris workshop in 2023, an international workshop without proceedings. A paper is in preparation and will be submitted to a journal.

7.5.2 DisLog: a separation logic for disentanglement

Participants: Alexandre Moine, Sam Westrick *(Carnegie Mellon University)*, Stephanie Balzer *(Carnegie Mellon University)*.

Disentanglement is a run-time property of parallel programs that facilitates task-local reasoning about the memory footprint of parallel tasks. In particular, it ensures that a task does not access any memory locations allocated by another concurrently executing task. Disentanglement can be exploited, for example, to implement a high-performance parallel memory manager. The MPL (MaPLe) compiler for Parallel ML, developed at Carnegie Mellon University, uses such a memory manager. Prior research on disentanglement has focused on the design of optimizations. Disentanglement, so far, was not enforced: one must either trust the programmer to provide a disentangled program or rely on runtime instrumentation to detect and tolerate entanglement.

This year, Alexandre Moine, Sam Westrick, and Stephanie Balzer developed the first static analysis for disentanglement. They propose DisLog, a concurrent separation logic for disentanglement. DisLog enriches concurrent separation logic with facilities for reasoning about the fork-join structure of parallel programs, allowing a user of the logic to verify that memory accesses are effectively disentangled. A large class of programs, including race-free programs, exhibit memory access patterns that are disentangled "by construction". To reason about these patterns, on top of DisLog, the paper proposes an almost standard concurrent separation logic, DisLog+. In this high-level logic, no specific reasoning about memory accesses is needed: functional correctness proofs entail disentanglement.

This work has been presented at POPL 2024 [11].

7.5.3 Verifying extensible arrays in a concurrent setting

Participants: Clément Allain, Gabriel Scherer *(Partout)*.

This year, Gabriel Scherer proposed an implementation of `Dynarray`, an OCaml 5 livrary for resizeable arrays. This library will be integrated into the OCaml standard library with the release of OCaml 5.2.

Although `Dynarray` is meant to be used in a sequential context, its implementation must ensure memory safety even in the case of improper use, that is, even if concurrent updates take place. OCaml 5 does ensure memory safety for well-typed programs but, for performance reasons, `Dynarray` uses unsafe features that could compromise memory safety and require careful use. To reason about the correctness of this library, Clément Allain has suggested introducing strong invariants (used to reason about sequential uses of the library) and weak invariants (used to reason about concurrent uses of the library). Clément has formalized and verified `Dynarray` using the separation logic Iris.

This work has been presented at JFLA 2024 [15].

7.5.4 Verifying concurrent algorithms in OCaml 5

Participants: Clément Allain.

Following the release of OCaml 5, which introduces shared-memory parallelism into OCaml, new libraries for parallel programming are emerging. These libraries typically involve intricate algorithms, whose verification is particularly challenging. Clément Allain has been working on verifying key parts of several such libraries, including `saturn`, `kcas`, and `eio`, using the separation logic Iris.

Clément Allain has been working, in particular, on the verification of the Chase-Lev work-stealing deque [25]. Work-stealing is a popular scheduling policy, governing how running tasks are divided among processors, in which an idle processor may steal tasks from others, thereby performing load balancing. In a typical work-stealing algorithm, each processor owns a concurrent deque (that is, a concurrent double-ended queue), which stores its ready tasks. This deque features three operations: the owner may *push* and *pop* tasks at one end of the deque, while the thieves may *steal* tasks at the other end. The Chase-Lev work-stealing deque is implemented in the Saturn library and is used in the `domainslib` library.

In May 2023, Clément Allain presented a preliminary result, the verification of a simplified version of the Chase-Lev work-stealing deque, at the Iris Workshop. The main simplification was the replacement of a finite array with an ideal infinite array. More work is required to eliminate this simplification.

7.5.5 Thunks and debits in separation logic

Participants: François Pottier, Armaël Guéneau *(Inria Saclay)*, Jacques-Henri Jourdan *(CNRS)*, Glen Mével.

A thunk is a mutable data structure that offers a simple memoization service: it stores either a suspended computation or the result of this computation. In a famous book (1999), Okasaki presents many data structures that exploit thunks to achieve good amortized time complexity. He analyzes their complexity by associating a *debit* with every thunk. We prove that Okasaki's reasoning rules can be reconstructed in the setting of Iris$^{\$}$, a rich separation logic with time credits. In comparison with our own earlier work on this topic [30], this work adds a key reasoning rule, the consequence rule for thunks, which was previously missing, and whose justification is non-trivial. We demonstrate the expressiveness of our new set of rules by verifying a few operations on streams as well as several of Okasaki's data structures, namely the physicist's queue, implicit queues, and the banker's queue. This paper has been presented at POPL 2024 [13].

7.5.6 Osiris: formal semantics and reasoning rules for OCaml

Participants: Arnaud Daby-Seesaram, François Pottier, Armaël Guéneau *(Inria Saclay)*, Remy Seassau.

The Osiris project aims to develop an instance of the Iris separation logic, inside the Coq 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.

In the spring of 2023, Arnaud Daby-Seesaram began working on this topic as a master's intern (M2), co-advised by Armaël Guéneau and François Pottier. Then, beginning in September 2023, Remy Seassau took up this line of work, as part of his PhD research.

At this point, a number of preliminary results have been obtained. First, a formal semantics of a large fragment of the OCaml programming language has been defined inside Coq. This semantics is original in that it is neither a standard small-step semantics, nor a standard big-step semantics; instead, it is written in the form of a monadic interpreter, where the semantics of the monad itself is defined in a small-step style. This approach is pleasant because a monadic interpreter is easy to read and understand and can (to some extent) be executed by Coq, and because the monad serves as a micro-language whose semantics is very simple. Second, two sets of reasoning rules have been developed. The first set of rules forms a Hoare logic for the pure subset of the language. The second set of rules forms a Separation Logic, based on Iris, for the complete language, including impure expressions (that is, expressions that may have side effects, including divergence, non-determinism, and mutable state). The two logics can interact.

A presentation about this work has been given by Arnaud Daby-Seesaram at the OCaml workshop in September 2023. The project is still at an early stage. Much work is needed to make program verification easier, to support a larger fragment of OCaml, and to carry out case studies.

8 Bilateral contracts and grants with industry

8.1 Bilateral contracts with industry

8.1.1 Tarides

Participants: Frédéric Bour, Thomas Refis, François Pottier, Didier Rémy.

One of our PhD students, Frédéric Bour, is employed by Tarides and carries out a PhD under a CIFRE agreement. Tarides is a small high-tech software company, with a strong expertise in virtualization, distributed systems, and programming languages. Several of their key products, such as MirageOS, are developed using OCaml. Tarides contributes a significant amount of manpower to the OCaml ecosystem. We maintain strong informal ties with this company.

8.2 Bilateral grants with industry

8.2.1 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.

8.2.2 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.

9 Partnerships and cooperations

9.1 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 (§7.5.6) and to connect it with Gospel.

10 Dissemination

Participants: Damien Doligez, Yannick Forster, Xavier Leroy, Jean-Marie Madiot, Luc Maranget, François Pottier, Didier Rémy.

10.1 Promoting scientific activities

Yannick Forster is a member of the operations team of the ACM SIGPLAN Long-Term Mentoring Committee (SIGPLAN-M).

General chair, scientific chair Yannick Forster was the workshop co-chair for the conference ICFP 2023.

Members of conference program committees Yannick Forster was a member of the program commitee for the conference CPP 2023.

Jean-Marie Madiot was a member of the extended review committee for the conference ECOOP 2023.

Luc Maranget was a member of the program committee for the conference JFLA 2024 and for the workshop PLACE 2023.

François Pottier was a member of the program committee for the conference ESOP 2024.

Members of editorial boards Xavier Leroy was area editor for Journal of the ACM, in charge of the Programming Languages area, until August 2023. He is a member of the editorial boards of Journal of Automated Reasoning and of the diamond open access journal TheoretiCS.

Until September 2023, François Pottier was a member of the editorial board of the Journal of Functional Programming.

10.1.1 Research administration

Luc Maranget is an elected member of Inria *Commission d'évaluation* (Evaluation Committee, CE). In particular, he took part in the hiring committee for *chargés de recherche* and was among the CE members who were auditioned during the HCERES evaluation process.

Luc Maranget represents the Cambium team in the *Comité des utilisateurs des moyens informatiques* (computer users committee, CUMI).

François Pottier is the president of Inria Paris' *Comité de Suivi Doctoral.* Since June 2023, he is Inria's delegate in the pedagogical team of MPRI.

Didier Rémy is chair of the steering committee of the Inria-Nomadic Labs partnership. He has been Inria's delegate in the pedagogical team of MPRI until June 2023, and he is the Inria's delegate in the management board of MPRI.

10.2 Teaching - Supervision - Juries

10.2.1 Teaching

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

- Licence (L1): *Initiation aux systèmes d'exploitation*, Clément Allain, 16 HETD, Université Paris Cité, France.
- Licence (L3): *Programmation fonctionelle*, Clément Allain, 24 HETD, Université Paris Cité, France.

- Licence (L3): *Outils logiques*, Clément Blaudeau, 24h TD, Université de Paris Cité, France.
- Master (M2): *Proof assistants*, Yannick Forster, 18 HETD, MPRI, Université Paris Cité, France.
- Open lectures: *Structures de données persistantes*, Xavier Leroy, 16 HETD, Collège de France, France.
- Conference tutorial: *Théorie et pratique des effets en OCaml 5*, Xavier Leroy, 3 HETD, Journées Francophones des Langages Applicatifs (JFLA 2023), France.
- Licence (L3): *Programmation fonctionnelle*, Jean-Marie Madiot, 24 HETD, Université Paris Cité, France.
- Master (M2): *Proofs of Programs*, Jean-Marie Madiot, 18 HETD, MPRI, Université Paris Cité, France.
- Licence (L3): *Principles of programming languages*, Jean-Marie Madiot, 40 HETD, École Polytechnique, France.
- Licence (L3): *Programmation Fonctionelle*, Alexandre Moine, 24 HETD, Université de Paris, France.
- Master (M2): *Functional programming and type systems*, François Pottier, 18 HETD, MPRI, Université Paris Cité, France.

10.2.2 Supervision

François Pottier serves as a co-advisor for the master's research internship of Adrian Dapprich (Saarland University). The main advisor is Derek Dreyer (MPI-SWS).

The following PhD theses are in progress:

- PhD in progress: Clément Allain, *Parallel programming infrastructure for OCaml 5*, Université Paris Cité, since October 2022, advised by François Pottier.
- PhD in progress: Clément Blaudeau, *Formalizing and improving OCaml modules*, Université Paris Cité, since October 2021, advised by Didier Rémy and Gabriel Radanne (Inria team Cash).
- PhD (CIFRE) in progress: Frédéric Bour, *An interactive, modular proof environment for OCaml*, Université de Paris, since August 2020, advised by François Pottier and Thomas Gazagnaire (Tarides).
- PhD in progress: Nathanaëlle Courant, *Towards an efficient, formally-verified proof checker for Coq*, Université Paris Cité, since September 2019, advised by Xavier Leroy.
- PhD in progress: Alexandre Moine, *Formal verification of space bounds*, Université Paris Cité, since October 2021, advised by Arthur Charguéraud and François Pottier.
- 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.

10.2.3 Juries

Damien Doligez was a member of the jury for the PhD thesis of Julie Cailler (Université de Montpellier; defended December 2023).

Yannick Forster served as secretary for the PhD thesis jury of Ana de Almeida Gabriel Vieira Borges (University of Barcelona; defended January 2024).

Xavier Leroy was a reviewer and viva voce examiner for the PhD thesis of Hrutvik Kanabar (University of Kent; defended September 2023). He was a member of the jury for the PhD thesis of Basile Pesin (PSL; defended October 2023).

Jean-Marie Madiot was a member of the jury for the PhD thesis of Wendlasida Ouédraogo (Institut Polytechnique de Paris; defended September 2023).

François Pottier was a reviewer for the PhD theses of Simon Oddershede Gregersen (Aarhus University; defended March 2023), Laurent Prosperi (Sorbonne Université; defended September 2023), Baptiste Pauget (PSL; defended December 2023), and Simon Friis Vindum (Aarhus Université; defended December 2023). He was the president of the jury for the PhD thesis of Xavier Denis (Université Paris-Saclay; defended December 2023).

10.3 Popularization

in November 2023, Cécile Pierrot (INRIA, EPI Caramba) and Xavier Leroy ran a round table discussion about cryptography with students at École de guerre, Paris.

In December 2023, at "Des preuves et des programmes", an event organised by Institut Henri Poincaré and targeting university students, Jean-Marie Madiot gave an accessible introduction to the problem of program verification.

11 Scientific production

11.1 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. URL: https://hal.inria.fr/hal-03470858.

[2] C. Blaudeau, D. Rémy and G. Radanne. 'Retrofitting OCaml modules: Fixing signature avoidance in the generative case'. In: *Journées Francophones des Langages Applicatifs*. JFLA 2023 - 34èmes Journées Francophones des Langages Applicatifs. Praz-sur-Arly, France, 16th Jan. 2023, pp. 59–100. URL: https://inria.hal.science/hal-03936636.

[3] N. Courant and X. Leroy. 'Verified Code Generation for the Polyhedral Model'. In: *Proceedings of the ACM on Programming Languages* 5.POPL (Jan. 2021), 40:1–40:24. DOI: 10.1145/3434321. URL: https://hal.archives-ouvertes.fr/hal-03000244.

[4] 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. URL: https://hal.inria.fr/hal-03049514.

[5] 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. URL: https://hal.archives-ouvertes.fr/hal-03350199.

[6] G. Mével, J.-H. Jourdan and F. Pottier. 'Cosmo: A Concurrent Separation Logic for Multicore OCaml'. In: *ICFP 2020 - 25th ACM SIGPLAN International Conference on Functional Programming*. ICFP 2020. ACM. New-York / Virtual, United States, Aug. 2020. DOI: 10.1145/3408978. URL: https://hal.archives-ouvertes.fr/hal-02929998.

[7] A. Moine, A. Charguéraud and F. Pottier. 'A High-Level Separation Logic for Heap Space under Garbage Collection'. In: *Proceedings of the ACM on Programming Languages*. POPL 7 (2022). DOI: 10.1145/3571218. URL: https://hal.inria.fr/hal-03852060.

[8] 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. URL: https://hal.inria.fr/hal-03426997.

11.2 Publications of the year

International journals

[9] A. W. Appel and X. Leroy. 'Efficient Extensional Binary Tries'. In: *Journal of Automated Reasoning* 67 (12th Jan. 2023), Article number 8. DOI: 10.1007/s10817-022-09655-x. URL: https://inria.hal.science/hal-03372247.

[10] A. Moine, A. Charguéraud and F. Pottier. 'A High-Level Separation Logic for Heap Space under Garbage Collection (Extended Version)'. In: *Proceedings of the ACM on Programming Languages* 7.POPL (2023), pp. 718–747. DOI: 10.1145/3571218. URL: https://inria.hal.science/hal-03823056.

[11] A. Moine, S. Westrick and S. Balzer. 'DisLog: A Separation Logic for Disentanglement'. In: *Proceedings of the ACM on Programming Languages* 8.POPL (5th Jan. 2024). DOI: 10.1145/3632853. URL: https://inria.hal.science/hal-04291381.

[12] P. E. de Vilhena and F. Pottier. 'Verifying an Effect-Handler-Based Define-By-Run Reverse-Mode AD Library'. In: *Logical Methods in Computer Science* 19.4 (23rd Oct. 2023), p. 51. DOI: 10.46298/lmcs-19(4:5)2023. URL: https://inria.hal.science/hal-04292453.

International peer-reviewed conferences

[13] F. Pottier, A. Guéneau, J.-H. Jourdan and G. Mével. 'Thunks and Debits in Separation Logic with Time Credits'. In: *Proceedings of the ACM*. POPL 2024 - 51st ACM SIGPLAN Symposium on Principles of Programming Languages. Vol. 8. POPL. Londres, United Kingdom: ACM, Jan. 2024. URL: https://hal.science/hal-04238691.

[14] P. E. de Vilhena and F. Pottier. 'A Type System for Effect Handlers and Dynamic Labels'. In: *Lecture Notes in Computer Science*. European Symposium on Programming. Vol. 13990. Lecture Notes in Computer Science. Paris, France: Springer Nature Switzerland, 17th Apr. 2023, pp. 225–252. DOI: 10.1007/978-3-031-30044-8_9. URL: https://inria.hal.science/hal-03886668.

National peer-reviewed Conferences

[15] C. Allain and G. Scherer. 'Correct tout seul, sûr à plusieurs'. In: 35es Journées Francophones des Langages Applicatifs (JFLA 2024). Saint-Jacut-de-la-Mer, France, Jan. 2024. URL: https://inria.hal.science/hal-04406412.

[16] C. Blaudeau, D. Rémy and G. Radanne. 'Retrofitting OCaml modules: Fixing signature avoidance in the generative case'. In: *Journées Francophones des Langages Applicatifs*. JFLA 2023 - 34èmes Journées Francophones des Langages Applicatifs. Praz-sur-Arly, France, 16th Jan. 2023, pp. 59–100. URL: https://inria.hal.science/hal-03936636.

[17] F. Pottier. 'Correct, Fast LR(1) Unparsing'. In: 35es Journées Francophones des Langages Applicatifs (JFLA 2024). Saint-Jacut-de-la-Mer, France, Jan. 2024. URL: https://inria.hal.science/hal-04407116.

[18] M. Valnet, N. Courant, G. Bury, P. Chambart and V. Laviron. 'Chamelon : un minimiseur pour et en OCaml'. In: 35es Journées Francophones des Langages Applicatifs (JFLA 2024). Saint-Jacut-de-la-Mer, France, Jan. 2024. URL: https://inria.hal.science/hal-04407119.

Conferences without proceedings

[19] X. Leroy. 'Well-founded recursion done right'. In: CoqPL 2024: The Tenth International Workshop on Coq for Programming Languages. London, United Kingdom, 20th Jan. 2024. URL: https://inria.hal.science/hal-04356563.

Reports & preprints

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

[21] X. Leroy. *The CompCert C verified compiler: Documentation and user's manual.* Inria, 5th June 2023, pp. 1–79. URL: https://inria.hal.science/hal-01091802.

[22] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, K. Sivaramakrishnan and J. Vouillon. *The OCaml system release 5.1: Documentation and user's manual.* Inria, 14th Sept. 2023. URL: https://inria.hal.science/hal-00930213.

11.3 Cited publications

[23] 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.

[24] 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.

[25] D. Chase and Y. Lev. 'Dynamic circular work-stealing deque'. In: *SPAA 2005: Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures, July 18-20, 2005, Las Vegas, Nevada, USA.* Ed. by P. B. Gibbons and P. G. Spirakis. ACM, 2005, pp. 21–28. DOI: 10.1145/1073970.1073974. URL: https://doi.org/10.1145/1073970.1073974.

[26] 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.

[27] X. Leroy. 'A formally verified compiler back-end'. In: *Journal of Automated Reasoning* 43.4 (2009), pp. 363–446. URL: http://dx.doi.org/10.1007/s10817-009-9155-4.

[28] 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.

[29] J.-M. Madiot and F. Pottier. 'A Separation Logic for Heap Space under Garbage Collection'. In: *Proceedings of the ACM on Programming Languages* 6.POPL (Jan. 2022). URL: http://cambium.inria.fr/~fpottier/publis/madiot-pottier-diamonds-2022.pdf.

[30] G. Mével, J.-H. Jourdan and F. Pottier. 'Time Credits and Time Receipts in Iris'. In: *European Symposium on Programming.* Vol. 11423. Lecture Notes in Computer Science. Prague, Czech Republic: Springer, Apr. 2019, pp. 3–29. DOI: 10.1007/978-3-030-17184-1_1. URL: https://hal.science/hal-02183311.

[31] A. Rossberg, C. Russo and D. Dreyer. 'F-ing modules'. In: *Journal of Functional Programming* 24.5 (Sept. 2014), pp. 529–607. URL: https://doi.org/10.1017/S0956796814000264 (visited on 19/11/2020).

[32] 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. URL: https://doi.org/10.4204/EPTCS.198.2.