RESEARCH CENTRE

Inria Paris Center

IN PARTNERSHIP WITH:

Collège de France

2022

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]
- Jean-Marie Madiot [INRIA, Researcher]
- Luc Maranget [INRIA, Researcher]
- Didier Remy [INRIA, Senior Researcher, HDR]

Faculty Member

- Xavier Leroy [COLLEGE DE FRANCE, Professor]

PhD Students

- Clément Allain [INRIA, from Mar 2022]
- Clément Blaudeau [Université Paris Cité]
- Frédéric Bour [TARIDES]
- Basile Clement [ENS PARIS]
- Nathanëlle Courant [Université Paris Cité, until Aug 2022]
- Paulo De Vilhena [INRIA]
- Alexandre Moine [INRIA]
- Glen Mével [INRIA, until Aug 2022]
- Thomas Refis [TARIDES]

Technical Staff

- Florian Angeletti [INRIA, Engineer]
- Sebastien Hinderer [INRIA]

Administrative Assistants

- Helene Bessin Rousseau [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 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 [3], 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 [27]. 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 [34] and the Verasco abstract interpretation-based static analyzer [9]. 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 modern Separation Logics, such as “hidden state” and “monotonic state”. 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 has extended CFML with the ability to simultaneously verify the correctness and the time complexity of an OCaml component. Glen Mével and Paulo de Vilhena are currently investigating the use of Iris, a descendant of Concurrent Separation Logic, to carry out proofs of Multicore OCaml programs.

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

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

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 2014 approximately, the Multicore OCaml project at OCaml Labs (Cambridge, UK) is preparing a version of OCaml where multiple threads execute concurrently and communicate with each other via shared memory.

In principle, it seems desirable for Multicore OCaml to become the standard version of OCaml. Integrating Multicore OCaml into mainstream OCaml, however, is a major undertaking. The runtime system is deeply impacted: in particular, OCaml's current high-performance garbage collector must be replaced with an entirely new concurrent collector. The memory model and operational semantics of the language must be clearly defined. At the programming-language level, several major extensions are proposed, including effect handlers (a generalization of exception handlers, introducing a form of delimited control) and a new type-and-effect-discipline that statically detects and rejects unhandled effects.

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 Multicore OCaml intersects the first two axes, whereas creating verification technology for Multicore 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 integration of Multicore OCaml, 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 Multicore OCaml memory model. This might include proving that the axiomatic and operational presentations of the model agree; testing the Multicore OCaml 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 and in Multicore OCaml 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 June 2022, Xavier Leroy, Sandrine Blazy (U. Rennes 1), Zaynah Dargaye (Nomadic Labs), Jacques-Henri Jourdan (CNRS), Michael Schmidt (AbsInt), Bernhard Schommer (Saarland U. and AbsInt) and Jean-Baptiste Tristan (Boston College) received the 2021 ACM Software System Award "for the development of CompCert, the first practically useful optimizing compiler targeting multiple commercial architectures that has a complete, mechanically checked proof of its correctness".

In September 2022, Xavier Leroy received the 2022 ACM SIGPLAN Programming Languages Achievement Award for "fundamental contributions to both the theory and practice of programming languages on a range of topics, including type system and module system design, efficient compilation of functional programming languages, bytecode verification, verified compilation, and verified static analysis".

5.2 Statement

We must point out that in 2022, the activity of the Cambium team suffered from a number of institutional malfunctions at Inria. The team regrets a **lack of transparency** about the current status and evolution of the Institute, regarding its mission statement, its budget, its recruitment policy and capability, etc. (The absence of the 2021 Rapport Social, at the time of writing in January 2023, is but one symptom of this phenomenon.) The team regrets the lack of emphasis on **research** in the public discourse of the Institute. We believe that the primary mission of the Institute is and should remain research, well before innovation, let alone "soutien aux politiques publiques" (support for public policies) or "souveraineté numérique" (digital independence). The team regrets that the relationship between the head of the Institute (la Direction Générale) and the representative instances of the research personnel (la Commission d'Évaluation), which should be based on mutual trust and respect, has been severely degraded, perhaps even destroyed. The team wishes to thank the Commission d'Évaluation for its outstanding efforts, in 2022 and in previous years, in defending the interests of the research community, keeping us thoroughly informed about topics relevant to the scientific life at Inria, and upholding the moral and intellectual values that we are collectively proud of and that define our Institute.

6 New software and platforms

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: `http://compcert.inria.fr/`

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, Armaël 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.

News of the Year: Besides bug fixes, work on the proof manager in 2021 concentrated on the following items:

- proof methods for reasoning about the ENABLED and action composition operators of TLA+, which allow us to reason about liveness properties of TLA+ specifications,
- support for reasoning about recursively defined operators,
- and support for tuples in binding constructs such as quantifiers and set comprehension.

A new version of the SMT backend is in preparation, and several changes were made to the Isabelle backend. We expect all these new developments to be consolidated for a major release to appear in 2022.

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 Software

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 [34]. 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 [33], 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 compatibility with the ISO C 2011 standard by adding support for:

- `_Generic` expressions, which enable users to define overloaded functions with type-based overloading resolution;
- Unicode string literals and character constants, extending and clarifying CompCert's previous minimal support for extended character sets;
- unstructured `switch` statements (as exemplified by Duff's device). They are implemented via a (not yet verified) translation from unstructured `switch` statements to the structured, Java-style `switch` statements provided by Compcert and formalized earlier, combined with labels and `goto` statements.

Other improvements include:

- a strengthening of the if-conversion optimization, which now recognizes more `if-else` statements that can be turned into conditional move instructions, and a revised correctness proof for this optimization, using a new simulation diagram based on an “eventually” modality;
- better control-flow analysis of calls to `noreturn` functions, resulting in more accurate warnings;
- more careful type-checking of unprototyped function definitions, with additional warnings;
- better cooperation with the linker, including support for mergeable literal sections and for relocatable read-only sections.

We released two versions of CompCert incorporating these improvements: version 3.11 in June 2022 and version 3.12 in November 2022.

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 *(University of Cambridge)*, Jacques Garrigue *(University of Nagoya)*, Sadiq Jaffer *(Tarides)*, Tom Kelly *(Tarides)*, Guillaume Munch-Maccagnoni *(Inria team Gallinette)*, Olivier Nicole *(Tarides)*, Nicolás Ojeda Bär *(Lexifi)*, Sudha Parimala *(Tarides)*, KC Sivaramakrishnan *(IIT Madras)*, Gabriel Scherer *(Inria team Partout)*, Leo White *(Jane Street LLC)*.

This year, on December 16th, we have released the first new major version of OCaml in ten years: OCaml 5.0. We have also released one new normal version, 4.14.0, in March, and a new patch version, 4.14.1, in December.

Nearly all of our work on OCaml this year has been dedicated to the stabilisation of the "Multicore OCaml" prototype project and to its integration into mainstream OCaml, in preparation for the release of OCaml 5.

OCaml 5 is a complete rewrite of the OCaml runtime system. It adds support for shared-memory parallelism and for effect handlers to the OCaml language. Support for shared-memory parallelism means that OCaml 5 can now make full use of operating-system-level concurrency and therefore exploit the full processing power of multicore CPU architectures. Support for effect handlers, a programming construct that allows a computation to be suspended and later resumed, allows library and application programmers to set up more lightweight forms of application-level concurrency. OCaml 5 is the first mainstream programming language with built-in support for effect handlers.

Beyond those two major changes (and the removal of many deprecated functions), OCaml 5 is functionally equivalent to OCaml 4.14: all programs that could be compiled and executed by OCaml 4 can also be compiled and run by OCaml 5, with similar performance. As an exception to this general rule, the performance of ephemerons has been degraded; we are planning to restore it in a future release.

Before this major new version, OCaml 4.14 brought a sizeable set of new features:

- tail-modulo-cons optimization (see also §7.3.2);
- explicit type bindings in signatures;
- a speed-up of the garbage collector, thanks to a prefetching optimization;
- tail call optimization is now guaranteed for functions with up to 64 arguments;
- UTF tools, codecs and validation for the modules `Uchar`, `Bytes` and `String`;
- new standard library modules `In_channel` and `Out_channel`;
- improved error messages when disambiguation could not possibly work;
- improved error messages for mismatched record and variant definitions;
- more detailed error messages, containing full error traces, for module inclusion checks;
- production of metadata that allows tools such as Merlin to trace the definition of a value.

Overall, counting both OCaml 5 and OCaml 4.14, we have published more than 180 usability improvements and more than 56 bug fixes.

7.1.3 The `diy` tool suite

Participants: 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 by improving the support for the virtual memory extension, most noticeably by introducing explicit handlers in tests. He also designed and mostly implemented the new support for self-modifying code. Furthermore, he refined the support for the tagged memory extension, especially regarding test generators and the combination of tagged memory and virtual memory. More generally, Luc Maranget acts as the coordinator of the toolbox: he reviews and validates the pull requests submitted by various contributors, mostly ARM engineers.

7.1.4 TLAPS

Participants: Antoine Defourné *(Inria team Veridis)*, Damien Doligez, Ioannis Filippidis *(Inria team Veridis)*, Igor Konnov *(Informal Systems)*, Markus Kuppe *(Microsoft Research)*, Leslie Lamport *(Microsoft Research)*, Stephan Merz *(Inria team Veridis)*, Martin Riener *(TU Wien)*.

Damien Doligez is head of the “Tools for Proofs” team in the Microsoft-INRIA Joint Centre. The aim of this project is to extend the TLA+ language with a formal language for hierarchical proofs, formalizing Lamport’s ideas [32], and to build tools for writing TLA+ specifications and mechanically checking the proofs.

This year, we have continued the work that was started last year on extending TLAPS to handle liveness proofs, the `ENABLED` operator, and the action composition operators of TLA+. We have also continued our refactoring efforts in order to reduce the technical debt.

7.2 Programming language design and implementation

7.2.1 Formalizing and improving OCaml modules

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

This year, we continued our ongoing work whose aim is to revisit the type system of OCaml modules. This work is inspired by the paper *F-ing modules* [35], which gives a semantics to Standard ML modules by elaboration into F^ω. We adapt this work to the case of OCaml, which differs technically from SML, and we improve upon it. Our first core contribution is to introduce an intermediate system, known as the *canonical* system, which closely resembles the OCaml module system, but uses F^ω-style binders to introduce types. Our second core contribution is to translate F^ω-style canonical signatures back into the source syntax.

This work can be decomposed in two stages: in the first stage, we study the restricted case where only “generative” functors are allowed; in the second stage, we study the general case where both generative and “applicative” functors are allowed.

The formalization of the easier generative case has been done and will be presented at the conference JFLA 2023 [21].

The harder case of applicative functors *à la OCaml* presents specific challenges. Specifically, we developed a new notion of *module identity* that models the *syntactic criterion* for applicativity. This led us to also consider the feature of *module aliases* and the proposed feature of *transparent ascription*, for which we give a comprehensive description via our canonical system. From there, we explored two directions:

- The encoding of applicative functors into F^ω was reworked to allow for skolemization of abstract types (from $\forall\alpha.\tau \to \exists\beta.\sigma[\beta]$ to $\exists\beta'.\forall\alpha.\tau \to \sigma[\beta'(\alpha)]$) in a much lighter way than in Rossberg *et al.*'s paper [35], via the introduction of a new notion of *transparent existentials* in F^ω.
- The translation of signatures from the canonical system (which uses F^ω-style binders) to source signatures (which uses a path-based representation) was extended to partially support higher-order abstract types. Our solution already covers the current implementation of OCaml. We are investigating a better coverage of the remaining signature-avoidance cases.

7.2.2 The Celsius model for safety of object initialization

Participants: Clément Blaudeau, Fengyun Liu *(Oracle Labs)*.

In object-oriented programming, an object *under initialization* does not fulfill its class specification yet and can be unsafe to use, as it may have uninitialized fields. This year, continuing a line of work that was started at EPFL, we finished the formal proof of soundness of the Celsius model, which ensures safe initialization in a small calculus. This led us to rework the meta-theory and our intuitions about the model. This work has been published, along with an artifact containing the Coq proofs, at OOSPLA 2022 [13].

7.2.3 A type system for effect handlers and dynamic labels

Participants: Paulo Emílio de Vilhena, François Pottier.

Type systems are one of the main components of modern programming languages. They provide safety guarantees: for example, they guarantee that well-typed programs do not crash. They also provide machine-checked documentation: a type provides a concise description of the behavior of a program.

This year, Paulo Emílio de Vilhena and François Pottier studied the question of designing a type system with support for *effect handlers*, a new feature of OCaml 5. This requires addressing the *aliasing challenge*: in the presence of dynamic allocation of effect labels, two distinct compile-time effect names may denote the same runtime effect label. This phenomenon makes it difficult to propose a sound and precise type system for effect handlers. Several previous proposals in the literature either were restricted to *lexically scoped handlers* or relied on *effect coercions*, which are complex and do not exist in OCaml 5. De Vilhena and Pottier designed a new type system that addresses the aliasing challenge and does not have either of these limitations: it is not restricted to lexically scoped handlers and does not involve effect coercions. A description of this type system and a proof of its soundness have been accepted for publication and presentation at the conference ESOP 2023 [25].

7.2.4 Improvements to Inferno

Participants: François Pottier, Gabriel Scherer *(Inria team Partout)*.

Inferno is an OCaml library that aims to offer a powerful and reusable engine for type inference based on equality constraints.

This year, François Pottier and Gabriel Scherer made the following changes and improvements:

- support for rigid variables, that is, variables that cannot be unified with other rigid variables or with older flexible variables;

- a simplification and cleanup of the library's public API;
- improved documentation;
- minor performance improvements.

7.2.5 A study of reverse-mode automatic differentiation

Participants: François Pottier, Solal Toulouse *(ENS Rennes)*.

The paper You Only Linearize Once: Tangents Transpose to Gradients, by Radul *et al.*, to appear at POPL 2023, explains how reverse-mode automatic differentiation, a program transformation that is traditionally considered quite complex, can in fact be understood as the composition of three simpler program transformations, namely forward-mode automatic differentiation, unzipping, and transposition. François Pottier supervised Solal Toulouse, a student at ENS Rennes, during a 2-month internship whose aim was to implement these program transformations. As an outcome of this internship, François Pottier designed a programming project, also based on this topic, for the students of the course *Functional programming and type systems* (MPRI 2.4).

7.3 Verified compilation

7.3.1 Verification of tensor compilers

Participants: Basile Clément.

Basile Clément, advised by Xavier Leroy and Albert Cohen (Google), has been working on the translation-validation of tensor compilers, and in particular, of the Halide compiler.

Tensor compilers are used to transform high-level specifications of multi-dimensional computations into low-level code that runs efficiently on the hardware. Such compilers perform complex loop-based code transformations which can drastically change the structure of the code, as well as more traditional simplifications. The problem is to develop methods to formally prove that the code that is generated by the compiler is correct with respect to the specification.

This year, Basile refined his translation-validation approach to the topic and further developed his practical tool for the verification of affine Halide programs to handle more constructs of the Halide language, making it applicable to a larger set of official benchmarks. He presented this work at the conference OOPSLA 2022 [18]. He also completed and defended his PhD dissertation [22]. The dissertation gives the details of the approach and lays the groundwork for an extension of the approach to handle associative-commutative reductions.

7.3.2 Verification of tail-call optimization modulo constructors

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

Tail-call optimization (TCO), a well-known optimization performed by many compilers for functional programming languages, optimizes tail calls by deallocating the stack frame of the caller before jumping to the callee. *Tail-call optimization modulo constructors* (TMC), improves upon TCO by also optimizing the situation where a function call is *not* a tail call because it is followed by the application of a constructor. A canonical example appears in the most natural implementation of the `map` function on lists. In such a situation, the code can be transformed into *destination-passing style*: first, a memory block is allocated and partially initialized; then, a tail call takes place, and the responsibility of completing the initialization of the memory block is delegated to the callee.

TMC has been implemented in the OCaml compiler by Frédéric Bour, Basile Clément and Gabriel Scherer [29]. This year, based on this work, Clément Allain and François Pottier have formalized and verified TMC for an untyped sequential λ-calculus equipped with mutable heap cells. The correctness of the transformation is expressed as a *termination-preserving behavioral program refinement*. The proof relies on a *relational separation logic* that is constructed on top of Iris [31], drawing inspiration from Simuliris [30]. The proof has been sketched on paper; its formalisation in the Coq proof assistant is ongoing work. We hope to submit this work to a conference in 2023.

7.4 Shared-memory concurrency

7.4.1 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, are currently designing a significant extension of the ARM `aarch64.cat` memory model: virtual memory. This work, which is pending approval by ARM before release, describes the interaction of the memory model and of virtual memory. Its real-world relevance and potential impact are high, as such a specification is necessary in order to write correct operating systems.

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.

This work was started last year. Significant advances were made this year concerning the management of faults: in particular, the possibility of writing explicit fault handlers in tests was added. Many technicalities, such as the ability to switch execution levels (back and forth), were also addressed.

7.4.2 Semantics of AArch64 instructions

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

Starting this autumn, Hadrien Renaud is a PhD candidate under the supervision of Jade Alglave. Hadrien'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 and attends the meetings with the PhD candidate.

7.5 Software specification and verification

7.5.1 A high-level Separation Logic for heap space under garbage collection

Participants: Alexandre Moine, Arthur Charguéraud, Jean-Marie Madiot, 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.) It has long been tempting to also extend it with *space credits*, so as to allow establishing verified space complexity bounds. Unfortunately, when verifying programs expressed in a high-level programming language, such as OCaml, a major obstacle arises. Because memory is managed by the garbage collector, memory deallocation is implicit: it is not clear in the code where memory can be freed.

Last year, Jean-Marie Madiot and François Pottier presented a Separation Logic with space credits [15] that allows reasoning about heap space under garbage collection. They introduced "pointed-by" assertions that keep track of the predecessors of every block and can be used to prove that a block is unreachable. However, their work was carried out in the setting of a low-level, assembly-like programming language.

This year, Alexandre Moine, Arthur Charguéraud and François Pottier adapted these ideas to a high-level language, where a central problem is to identify the memory locations that the garbage collector considers as roots. For this purpose, they proposed new "Stackable" assertions, which keep track of the existence of stack-to-heap pointers in a reasonably lightweight manner. In addition, they explain how to reason about closures, which are heap-allocated blocks that can contain pointers to other blocks.

A preliminary version of this work, without support for closures, was presented at the peer-reviewed workshop ASL 2022. A more full-fledged version has been accepted for publication and presentation at the conference POPL 2023 [16].

As a case study, Jean-Marie Madiot used Moine, Charguéraud and Pottier's new program logic to specify and verify a "vector" data structure. A vector is a resizable array that supports "push" and "pop" operations. The array automatically grows and shrinks when necessary. The proof guarantees that the code is functionally correct and has correct amortized time and space complexity. A program that uses this data structure, namely a depth-first search algorithm, has also been verified, together with its time and space complexities.

7.5.2 Proofs of programs with effect handlers

Participants: Paulo Emílio de Vilhena, François Pottier.

This year, Paulo Emílio de Vilhena wrote his dissertation, entitled "Proof of Programs with Effect Handlers" [23]. This thesis addresses the problem of reasoning about programs that modify the heap and alter the control flow through effect handlers, a novel programming construct that provides a relatively simple interface to delimited control. The thesis begins with a description of Hazel, an extension of Separation Logic with support for effect handlers and one-shot continuations. It then explores the application of Hazel to a number of case studies: (1) a program that transforms a *higher-order iteration method* into a *lazy sequence*; (2) a library for *asynchronous computation*; and (3) a library for *reverse-mode automatic differentiation*. The last case study is the subject of a paper that has been submitted for publication in 2021 and entirely revised in 2022. A preprint is available online. The thesis also documents Maze, a Separation Logic for effect handlers with multi-shot continuations. The applicability of Maze is assessed through the verification of a simple SAT solver that uses multi-shot continuations to implement backtracking and through the design of reasoning rules for the control operators `callcc` and `throw`. Finally, the thesis documents the design of a type system for effect handlers: see §7.2.3. Paulo defended his thesis on December 16, 2022.

7.5.3 Towards an efficient, formally-verified proof checker for Coq

Participants: Nathanaëlle Courant.

Nathanaëlle Courant, who is advised by Xavier Leroy, has been working on writing a formally verified and efficient convertibility test for Coq.

One of the key challenges is to perform strong reduction (that is, reduction under binders); computers are usually better suited to weak reduction. With strong reduction, care must be taken to not evaluate too far, and to avoid reducing the body of functions that will not appear in the final term. Two related difficult problems are to compare two reduced terms and to combine the convertibility check with reduction itself.

This year, Nathanaëlle finished a Coq proof of an efficient convertibility checker for a λ-calculus extended with data constructors and pattern matching. In the case of function applications, this algorithm launches parallel attempts to check the convertibility of the function arguments and the convertibility of the whole terms.

Nathanaëlle also wrote a large part of her thesis manuscript, which includes the presentation of a big-step strong call-by-need semantics for the λ-calculus, the convertibility checker discussed above, and the presentation and discussion of the accompanying Coq proofs.

8 Bilateral contracts and grants with industry

8.1 Bilateral contracts with industry

8.1.1 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

In the future, we would like the Caml Consortium to be superseded by the OCaml Software Foundation, discussed below. For the moment, 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.

8.1.2 Tarides

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

Two of our PhD students, Frédéric Bour and Thomas Refis, are employed by Tarides and carry 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.

8.2 Bilateral grants with industry

8.2.1 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.

8.2.2 Nomadic Labs

Nomadic Labs, a Paris-based company, has implemented the Tezos blockchain and cryptocurrency entirely in OCaml. In 2019, Nomadic Labs and Inria have signed a framework agreement ("contrat-cadre") that allows Nomadic Labs to fund multiple research efforts carried out by Inria groups. Within this framework, we have received three 3-year grants:

- "Évolution d'OCaml". This grant is intended to fund a number of improvements to OCaml, including the addition of new features and a possible re-design of the OCaml type-checker. This grant has allowed us to fund Jacques Garrigue's visit (10 months, from September 2019 to June 2020) and to hire Gabriel Radanne on a Starting Research Position (14 months, from October 2019 to November 2020).
- "Maintenance d'OCaml". This grant is intended to fund the day-to-day maintenance of OCaml as well as the considerable work involved in managing the release cycle. This grant has allowed us to hire Florian Angeletti as an engineer for 3.5 years. As of January 1st, 2023, Florian has been recruited on a permanent position as an Inria engineer.
- "Multicore OCaml". This grant is intended to encourage research work on Multicore OCaml within our team. This grant has allowed us to fund Glen Mével's PhD thesis (3 years).

9 Dissemination

Participants: Xavier Leroy, Jean-Marie Madiot, Alexandre Moine, François Pottier, Didier Rémy.

9.1 Promoting scientific activities

9.1.1 Scientific events: selection

Chairs of conference program committees François Pottier was an associate chair of the program committee for the conference POPL 2023.

Member of the conference program committees Xavier Leroy was a member of the program committee for the conference POPL 2023.

9.1.2 Journals

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

François Pottier is a member of the editorial board of the Journal of Functional Programming.

9.1.3 Leadership within the scientific community

François Pottier is the head of the ANR project GOSPEL, which was submitted and approved by ANR in 2022, and which is planned to begin in January 2023.

9.1.4 Research administration

François Pottier is a member of Inria Paris' *Commission de Développement Technologique* and the president of Inria Paris' *Comité de Suivi Doctoral*.

Didier Rémy is a co-chair of the steering committee of the Inria-Nomadic Labs partnership. He is Inria's delegate in the pedagogical team and management board of MPRI.

9.2 Teaching - Supervision - Juries

9.2.1 Teaching

In 2022, the members of our team have taught or assisted in teaching the following courses:

- Open lectures: Xavier Leroy, *Sécurité du logiciel: quel rôle pour les langages de programmation?* 16 HETD, Collège de France, 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): *Introduction à l'informatique*, 40 HETD, École Polytechnique, France.
- Licence (L2): *Proofs of Programs*, Alexandre Moine, 36 HETD, 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 Rémy, 18 HETD, MPRI, Université Paris Cité, France.

9.2.2 Supervision

The following PhD theses are in progress or have been defended in 2022:

- 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: Basile Clement, *Translation validation of tensor compilers*, Université PSL, advised by Xavier Leroy and Albert Cohen (Google), defended on September 9th, 2022 [22].
- 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: Paulo Emílio de Vilhena, *Proof of programs with effect handlers*, Université Paris Cité, advised by François Pottier, defended on December 16, 2022 [23].
- PhD: Glen Mével, *Towards a system for proving the correctness of concurrent Multicore OCaml programs*, Université Paris Cité, advised by Jacques-Henri Jourdan and François Pottier, defended on December 14, 2022.

- PhD in progress: Alexandre Moine, *Formal verification of space bounds*, Université Paris Cité, since October 2021, advised by Arthur Charguéraud and François Pottier.
- PhD (CIFRE) in progress: Thomas Refis, *Designing and formalizing modular implicits*, Université Paris Cité, since February 2021 (paused since September 2022), advised by Didier Rémy and Thomas Gazagnaire (Tarides).

9.2.3 Juries

Xavier Leroy was a reviewer for the PhD defense of Aurèle Barrière (ENS Rennes, December 2022). He was a member of the jury for the PhD defense of Basile Clément (PSL University, September 2022).

Xavier Leroy participated in the jury for the 2022 session of *agrégation externe d'informatique*.

Jean-Marie Madiot was a member of the jury for the PhD defense of Guillaume Ambal (Université de Rennes 1, October 2022).

François Pottier was a reviewer and member of the jury for the master's defense of Tiago Soares (Universidade NOVA de Lisboa, December 2022). He was also a member of the jury for the 2021 EAPLS Best PhD Dissertation Award.

9.3 Popularization

Freek Wiedijk's list of famous formalized theorems serves as an indicator of the performance and popularity of several proof assistants. Jean-Marie Madiot maintains a version of this list that is specialized for the Coq proof assistant. In 2022, he performed a lengthy revision process, so as to evaluate the reproducibility of the proofs, and so as to determine which proofs are constructive.

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

[2] J. Alglave, L. Maranget and M. Tautschnig. 'Herding cats: modelling, simulation, testing, and data-mining for weak memory'. In: *ACM Transactions on Programming Languages and Systems* 36.2 (2014). URL: http://dx.doi.org/10.1145/2627752.

[3] 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.

[4] B. Clément and A. Cohen. 'End-to-End Translation Validation for the Halide Language'. In: OOPSLA 2022 - Conference on Object-Oriented Programming Systems, Languages, and Applications. Vol. 6. Proceedings of the ACM on Programming Languages (PACMPL) No. OOPSLA1, Article 84. Auckland, New Zealand, 5th Dec. 2022. DOI: 10.1145/3527328. URL: https://hal.inria.fr/hal-03653857.

[5] 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.

[6] J. Cretin and D. Rémy. 'System F with Coercion Constraints'. In: *CSL-LICS 2014: Computer Science Logic / Logic In Computer Science*. ACM, 2014. URL: http://dx.doi.org/10.1145/2603088.2603128.

[7] 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.

[8] A. Guéneau, J.-H. Jourdan, A. Charguéraud and F. Pottier. ‘Formal Proof and Analysis of an Incremental Cycle Detection Algorithm’. In: *Interactive Theorem Proving*. Ed. by J. Harrison, J. O’Leary and A. Tolmach. Vol. 141. Leibniz International Proceedings in Informatics. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Sept. 2019. URL: https://hal.inria.fr/hal-02167236.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

10.2 Publications of the year

International journals

[13] C. Blaudeau and F. Liu. ‘A conceptual framework for safe object initialization: a principled and mechanized soundness proof of the Celsius model’. In: *Proceedings of the ACM on Programming Languages* 6.OOPSLA2 (31st Oct. 2022), pp. 729–757. DOI: 10.1145/3563314. URL: https://hal.inria.fr/hal-03902583.

[14] N. Courant, J. Lepiller and G. Scherer. ‘Debootstrapping without Archeology’. In: *The Art, Science, and Engineering of Programming* 6.3 (18th Feb. 2022). DOI: 10.22152/programming-journal.org/2022/6/13. URL: https://hal.archives-ouvertes.fr/hal-03917754.

[15] J.-M. Madiot and F. Pottier. ‘A Separation Logic for Heap Space under Garbage Collection’. In: *Proceedings of the ACM on Programming Languages* (Jan. 2022). DOI: 10.1145/3498672. URL: https://hal.inria.fr/hal-03478162.

[16] 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.

International peer-reviewed conferences

[17] 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. URL: https://hal.inria.fr/hal-03936636.

[18] B. Clément and A. Cohen. ‘End-to-End Translation Validation for the Halide Language’. In: OOPSLA 2022 - Conference on Object-Oriented Programming Systems, Languages, and Applications. Vol. 6. Proceedings of the ACM on Programming Languages (PACMPL) No. OOPSLA1, Article 84. Auckland, New Zealand, 5th Dec. 2022. DOI: 10.1145/3527328. URL: https://hal.inria.fr/hal-03653857.

[19] A. Moine, A. Charguéraud and F. Pottier. ‘Specification and Verification of a Transient Stack’. In: CPP 2022 - 11th ACM SIGPLAN International Conference on Certified Programs and Proofs. Philadelphia, United States, 17th Jan. 2022. DOI: 10.1145/3497775.3503677. URL: https://hal.inria.fr/hal-03472028.

[20] 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.

Scientific books

[21] C. Keller, T. Bourke, S. Blazy, F. Bour, G. Bury, S. Dumbrava, D. Gallois-Wong, A. Guatto, D. Janin, M. Kerjean, L. Pellissier, M. Pereira, A. Trieu and Y. Zakowski. *33èmes Journées Francophones des Langages Applicatifs*. Saint-Médard-d'Excideuil, France, 28th June 2022, pp. 1–292. URL: https://hal.inria.fr/hal-03689075.

Doctoral dissertations and habilitation theses

[22] B. Clément. 'Translation Validation of Tensor Compilers'. École Normale Supérieure (Paris), 9th Sept. 2022. URL: https://theses.hal.science/tel-03903895.

[23] P. E. de Vilhena. 'Proof of Programs with Effect Handlers'. Université Paris Cité, 16th Dec. 2022. URL: https://hal.inria.fr/tel-03891381.

Reports & preprints

[24] A. W. Appel and X. Leroy. *Efficient Extensional Binary Tries*. 21st Sept. 2022. URL: https://hal.inria.fr/hal-03372247.

[25] P. Emílio de Vilhena and F. Pottier. *A Type System for Effect Handlers and Dynamic Labels*. 6th Dec. 2022. URL: https://hal.inria.fr/hal-03886668.

[26] X. Leroy. *The CompCert C verified compiler: Documentation and user's manual*. Inria, 25th Nov. 2022, pp. 1–79. URL: https://hal.inria.fr/hal-01091802.

[27] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, K. Sivaramakrishnan and J. Vouillon. *The OCaml system release 5.0: Documentation and user's manual*. Inria, 16th Dec. 2022, pp. 1–989. URL: https://hal.inria.fr/hal-00930213.

[28] A. Moine, A. Charguéraud and F. Pottier. *A High-Level Separation Logic for Heap Space under Garbage Collection (Extended Version)*. Inria, 2022. URL: https://hal.inria.fr/hal-03823056.

10.3 Cited publications

[29] F. Bour, B. Clément and G. Scherer. 'Tail Modulo Cons'. In: *JFLA 2021 - Journées Francophones des Langages Applicatifs*. Saint Médard d'Excideuil, France, Apr. 2021. URL: https://hal.inria.fr/hal-03146495.

[30] L. Gäher, M. Sammler, S. Spies, R. Jung, H.-H. Dang, R. Krebbers, J. Kang and D. Dreyer. 'Simuliris: a separation logic framework for verifying concurrent program optimizations'. In: *Proc. ACM Program. Lang.* 6.POPL (2022), pp. 1–31. DOI: 10.1145/3498689. URL: https://doi.org/10.1145/3498689.

[31] R. Jung, R. Krebbers, J.-H. Jourdan, A. Bizjak, L. Birkedal and D. Dreyer. 'Iris from the ground up: A modular foundation for higher-order concurrent separation logic'. In: *Journal of Functional Programming* 28.e20 (2018). DOI: 10.1017/S0956796818000151. URL: https://hal.archives-ouvertes.fr/hal-01945446.

[32] 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.

[33] 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.

[34] 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.

[35] 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).