IN PARTNERSHIP WITH:
Collège de France

Activity Report 2019

Project-Team CAMBIUM

Programming languages: type systems, concurrency, proofs of programs

RESEARCH CENTER
Paris

THEME
Proofs and Verification

Table of contents

Project-Team CAMBIUM

Creation of the Project-Team: 2019 August 01

Keywords:

Computer Science and Digital Science:

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, Visitors, External Collaborators

Research Scientists

François Pottier [Team leader, Inria, Senior Researcher, HDR]
Damien Doligez [Inria, Researcher]
Ioannis Filippidis [Inria, Starting Research Position]
Xavier Leroy [Collège de France, Professor]
Jean-Marie Madiot [Inria, Researcher]
Luc Maranget [Inria, Researcher]
Gabriel Radanne [Inria, Starting Research Position, from Oct 2019]
Didier Rémy [Inria, Senior Researcher, HDR]

PhD Students
Frédéric Bour [Tarides, PhD Student, from Oct 2019, CIFRE]
Basile Clément [Inria, PhD Student, from Dec 2019]
Nathanaël Courant [École Normale Supérieure de Paris, PhD Student, from Sep 2019]
Paulo Emílio de Vilhena [Inria, PhD Student, from Sep 2019]
Armaël Guéneau [Inria, PhD Student]
Quentin Ladeveze [Inria, PhD Student, from Oct 2019]
Glen Mével [Inria, PhD Student]

Technical staff
Florian Angeletti [Inria, Engineer, from Sep 2019]
Sébastien Hinderer [Inria, Engineer]

Interns and Apprentices
Paulo Emílio de Vilhena [Inria, from Apr 2019 to Aug 2019]
Carine Morel [Inria, from Apr 2019 to Aug 2019]
Antoine Hacquard [Inria, from Sep 2019 to Jan 2020]

Administrative Assistant
Hélène Milome [Inria, Administrative Assistant]

Visiting Scientist
Jacques Garrigue [Nagoya University, from Sep 2019]

2. Overall Objectives

2.1. 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.

2.2. 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 Dijsktra 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.

2.3. 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 [2], are examples.

2.4. 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.

2.5. 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 [8] and the Verasco abstract interpretation-based static analyzer [6]. 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 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.

2.6. 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

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

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

3.1. Research Directions

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. New Software and Platforms

5.1. 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 and System Z), a debugger, a documentation generator, a compilation manager, a package manager, and many libraries contributed by the user community.

- Participants: Damien Doligez, Xavier Leroy, Fabrice Le Fessant, Luc Maranget, Gabriel Scherer, Alain Frisch, Jacques Garrigue, Marc Shinwell, Jeremy Yallop and Leo White
- Contact: Damien Doligez
- URL: https://ocaml.org/

5.2. Compcert

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.

RELEASE FUNCTIONAL DESCRIPTION: Novelties include a formally-verified type checker for CompCert C, a more careful modeling of pointer comparisons against the null pointer, algorithmic improvements in the handling of deeply nested struct and union types, much better ABI compatibility for passing composite values, support for GCC-style extended inline asm, and more complete generation of DWARF debugging information (contributed by AbsInt).

- Participants: Xavier Leroy, Sandrine Blazy, Jacques-Henri Jourdan, Sylvie Boldo and Guillaume Melquiond
- Partner: AbsInt Angewandte Informatik GmbH
- Contact: Xavier Leroy
- URL: http://compcert.inria.fr/

5.3. Diy

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.

- Participants: Jade Alglave and Luc Maranget
- Partner: University College London UK
- Contact: Luc Maranget
- URL: http://diy.inria.fr/

5.4. 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.

- Contact: François Pottier
- Publications: A Simple, Possibly Correct LR Parser for C11 - Reachability and Error Diagnosis in LR(1) Parsers

5.5. CFML

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.

- Participants: Arthur Charguéraud, Armaël Guéneau and François Pottier
- Contact: Arthur Charguéraud
- URL: http://www.chargueraud.org/softs/cfml/

5.6. TLAPS

TLA+ proof system

KEYWORD: Proof assistant

SCIENTIFIC DESCRIPTION: TLAPS is a platform for developing and mechanically verifying proofs about TLA+ specifications. 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.

FUNCTIONAL DESCRIPTION: TLAPS is a proof assistant for the TLA+ specification language.

NEWS OF THE YEAR: Work in 2019 focused on providing support for reasoning about TLA+'s ENABLED and action composition constructs. We also prepared a minor release, fixing some issues and switching to Z3 as the default SMT back-end solver.

- Participants: Damien Doligez, Stephan Merz and Ioannis Filippidis
- Contact: Stephan Merz
- URL: https://tla.msr-inria.inria.fr/tlaps/content/Home.html

5.7. ZENON

KEYWORD: Automated theorem proving

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 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 be retargeted to output scripts for different frameworks (for example, Isabelle and Dedukti).

- Author: Damien Doligez
- Contact: Damien Doligez
- URL: http://zenon-prover.org/

5.8. hevea

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/.

- Author: Luc Maranget
- Contact: Luc Maranget
- URL: http://hevea.inria.fr/

6. New Results

6.1. Programming language design and implementation

6.1.1. The OCaml system

Participants: Damien Doligez, Armaël Guéneau, Xavier Leroy, Luc Maranget, David Allsop [Cambridge University], Florian Angeletti, Frédéric Bour [Facebook, until Sep 2019], Stephen Dolan [Cambridge University], Alain Frisch [Lexifi], Jacques Garrigue [Nagoya University], Sébastien Hinderer [SED], Nicolás Ojeda Bär [Lexifi], Gabriel Radanne, Thomas Refis [Jane Street], Gabriel Scherer [Inria team Parsifal], Mark Shinwell [Jane Street], Leo White [Jane Street], Jeremy Yallop [Cambridge University].

This year, we released four versions of the OCaml system: versions 4.08.0, 4.08.1, 4.09.0, and 4.09.1. Versions 4.08.1 and 4.09.1 are minor releases that respectively fix 6 and 5 issues. Versions 4.08.0 and 4.09.0 are major releases that introduce new language features, improve performance and usability, and fix about 50 issues. The main novelties are:

- User-defined binding operators are now supported, with syntax similar to `let*`, `let+`, `and*`. These operators make it much easier to write OCaml code in monadic style or using applicative structures.
- The `open` construct now applies to arbitrary module expressions in structures and to applicative paths in signatures.
- A new notion of user-defined “alerts” generalizes the “deprecated” warning.
- New modules were added to the standard library: `Fun`, `Bool`, `Int`, `Option`, `Result`.
- Many floating-point functions were added, including fused multiply-add, as well as a new `Float.Array` submodule.
- Many error messages were improved, as well as error and warning reporting mechanisms.
- Pattern-matching constructs that correspond to affine functions are now optimized into arithmetic computations.

6.1.2. Evolution of the OCaml type system

Participants: Florian Angeletti, Jacques Garrigue, Thomas Refis [Jane Street], Didier Rémy, Gabriel Radanne, Gabriel Scherer [Inria team Parsifal], Leo White [Jane Street].

In addition to the work done on the above releases, efforts have been done to improve the type system and its implementation. Those include:

- Formalizing the typing of the pattern-matching of generalized algebraic data types (GADTs).
- Fixing some issues related to the incompleteness of the treatment of GADTs.
- Proposing extensions of the type system to reduce this incompleteness in concrete cases, by refining the information on abstract types.
- Exploring practical ways to obtain more polymorphism for functions whose soundness does not rely on the value restriction.
- Improving the readability of the type-checker code.
- Making the module layer of the type-checker more incremental, in order to improve efficiency and to facilitate integration with documentation tools.

6.1.3. Refactoring with ornaments in ML

Participants: Didier Rémy, Thomas Williams [Google Paris].

Thomas Williams, Lucas Baudin, and Didier Rémy have been working on refactoring and other transformations of ML programs based on mixed ornamentation and disornamentation. Ornaments have been introduced as a way of describing changes in data type definitions that can reorganize or add pieces of data. After a new data structure has been described as an ornament of an older one, the functions that operate on the bare structure can be partially or sometimes totally lifted into functions that operate on the ornamented structure.

This year, Williams and Rémy improved the formalization of the lifting framework. In particular, they introduced an intermediate language, in which nonexpansive expressions can be marked on source terms and traced during reduction. This allows to treat the nonexpansive part of expansive expressions as nonexpansive and use equational reasoning on nonexpansive parts of terms that appear in types. This approach significantly simplifies the metatheory of ornaments. This calculus could also have some interest in itself, beyond ornaments, to study languages with side effects.

6.1.4. A better treatment of type abbreviations during type inference

Participants: Didier Rémy, Carine Morel.

During her M2 internship under the supervision of Didier Rémy, Carine Morel revisited the treatment of type abbreviations in type inference for ML-like type systems, using a modern approach based on typing constraints [24]. Instead of expanding type abbreviations prior to unification, both the original abbreviated view and all expanded views are kept during unification, so as to avoid unnecessary expansions and use the least-expanded view whenever possible in the result of unification.

6.2. Software specification and verification

6.2.1. The CompCert formally-verified compiler

Participants: Xavier Leroy, Jacques-Henri Jourdan [CNRS], Michael Schmidt [AbsInt GmbH], Bernhard Schommer [AbsInt GmbH].

In the context of our work on compiler verification, since 2005, we have been developing and formally verifying a moderately-optimizing compiler for a large subset of the C programming language, generating assembly code for the ARM, PowerPC, RISC-V and x86 architectures [8]. This compiler comprises a back-end part, which translates the Cminor intermediate language to PowerPC assembly and which is reusable for source languages other than C [7], and a front-end, which translates the CompCert C subset of C to Cminor. The compiler is mostly written within the specification language of the Coq proof assistant, from which Coq's extraction facility generates executable OCaml code. The compiler comes with a 100000-line machine-checked Coq 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 added a new optimization to CompCert: "if-conversion", that is, the replacement of conditional statements and expressions by conditional move operations and similar branchless instruction sequences. As a consequence, fewer conditional branch instructions are generated. This replacement usually improves worst-case execution time (WCET), because mispredicted conditional branches tremendously increase execution time. This replacement is also interesting for cryptographic code and other programs that manipulate secret data: conditional branches over secret data take time that depends on the data, leaking some information, while conditional move instructions are constant-time and do not leak. The new if-conversion optimization plays a role in the ongoing work of Inria team Celtique on compilation that preserves constant-time properties. Its proof of semantic preservation is nontrivial and prompted the development of a new kind of simulation diagram.

Other recent improvements to the CompCert C compiler include:

- a new code generator targeting the AArch64 instruction set, that is, the 64-bit mode of the ARMv8 architecture;
- the ability to specify the semantics of certain built-in functions, making them amenable to optimizations such as constant propagation and common subexpression elimination;
- improvements to the verified C parser generated by Menhir, including fewer run-time checks, faster validation, and the removal of all axioms from the proof.

We released two versions of CompCert incorporating these improvements: version 3.5 in February 2019 and version 3.6 in September 2019.

6.2.2. Time credits and time receipts in Iris

Participants: Glen Mével, François Pottier, Jacques-Henri Jourdan [CNRS].

From March to August 2018, Glen Mével did an M2 internship at Gallium, where he was co-advised by Jacques-Henri Jourdan (CNRS) and François Pottier. Glen extended the program logic Iris with time credits and time receipts.

Time credits are a well-understood concept, and have been used in several papers already by Armaël Guéneau, Arthur Charguéraud, and François Pottier. However, because Iris is implemented and proved sound inside Coq, extending Iris with time credits requires a nontrivial proof, which Glen carried out, based on a program transformation which inserts “tick” instructions into the code. As an application of time credits, Glen verified inside Iris the correctness of Okasaki’s notion of “debits”, which allows reasoning about the time complexity of programs that use thunks.

Time receipts are a new concept, which allows proving that certain undesirable events, such as integer overflows, cannot occur until a very long time has elapsed. Glen extended Iris with time receipts and proved the soundness of this extension. As an application of time credits and receipts together, Jacques-Henri Jourdan updated Charguéraud and Pottier’s earlier verification of the Union-Find data structure [12] and proved that integer ranks cannot realistically overflow, even if they are stored using only $\log W$ bits, where W is the number of bits in a machine word.

This work carried out in 2018 has been published at ESOP 2019 [16].

6.2.3. A program logic for Multicore Ocaml

Participants: Glen Mével, François Pottier, Jacques-Henri Jourdan [CNRS].

Glen Mével, who is co-advised by Jacques-Henri Jourdan and François Pottier, has been working on designing a mechanized program logic for Multicore OCaml.

One of the key challenges is to enable deductive reasoning under a weak memory model. In such a model, the behaviors of a program are no longer described by a naive interleaving semantics. Thus, the operational semantics that describes a weak memory model often feels unnatural to the programmer, and is difficult to reason about.

This year, Glen designed and implemented a proof system on top of Iris, a modular separation logic framework whose implementation and soundness proof are both expressed in Coq. This system allows mechanized program verification for a fragment of the Multicore OCaml language. It provides a certain degree of abstraction over the low-level operational semantics, in the hope of simplifying reasoning. This abstraction includes an abstract concept of “local view” of the shared memory; views are exchanged between threads via atomic locations.

A few simple concurrent data structures have been proven correct using the system. They include several variants of locks and mutual exclusion algorithms.

Glen presented preliminary results at the Iris Workshop in October 2019.

6.2.4. Verifying a generic local solver in Iris

Participants: Paulo Emílio de Vilhena, Jacques-Henri Jourdan [CNRS], François Pottier.

From March to August 2019, Paulo Emílio de Vilhena did an M2 internship in our team, where he was advised by François Pottier, with precious help from Jacques-Henri Jourdan (CNRS).

Paulo verified a short but particularly subtle piece of code, namely a "local generic solver", that is, an on-demand, incremental, memoizing least fixed point computation algorithm. This algorithm is a slightly simplified version of `Fix`[2], an OCaml library published by François Pottier in 2009.

The specification of this algorithm is simple: the solver computes the optimal least fixed point of a system of monotone equations. Although the solver relies on mutable internal state for memoization and for “spying”, a form of dynamic dependency discovery, no side effects are mentioned in the specification. The challenge is precisely to formally justify why it is permitted to hide these side effects from the user.

The verification is carried out in Iris, a modern breed of concurrent separation logic. Iris is embedded in Coq, so the proof is machine-checked. The proof makes crucial use of prophecy variables, a novel feature of Iris. Auxiliary contributions include a restricted infinitary conjunction rule for Iris and a specification and proof of Longley’s “modulus” function, an archetypical example of spying.

[2] https://gitlab.inria.fr/fpottier/fix

This paper [13] has been accepted for presentation at the conference POPL 2020, which will take place in New Orleans in January 2020.

6.2.5. Formal reasoning about asymptotic complexity

Participants: Armaël Guéneau, Arthur Charguéraud [Inria team Camus], François Pottier, Jacques-Henri Jourdan [CNRS].

For several years, Armaël Guéneau, Arthur Charguéraud, François Pottier have been investigating the use of Separation Logic, extended with Time Credits, as an approach to the formal verification of the time complexity of OCaml programs. In 2018 and 2019, in collaboration with Jacques-Henri Jourdan, Armaël has worked on a more ambitious case study, namely a state-of-the-art incremental cycle detection algorithm, whose amortized complexity analysis is nontrivial. Armaël has proposed an improved and simplified algorithm and has carried out a machine-checked proof of its complexity. Furthermore, the verified algorithm has been released and is now used in production inside the Dune build system for OCaml. A paper has been published and presented at the International Conference on Interactive Theorem Proving (ITP 2019) [15]. A more detailed version of these results appears in Armaël Guéneau's dissertation [11], which was defended on December 16, 2019.

6.2.6. TLA+

Participants: Damien Doligez, Leslie Lamport [Microsoft Research], Ioannis Filippidis, Stephan Merz [Inria team VeriDis].

Damien Doligez is the 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 [25], and to build tools for writing TLA+ specifications and mechanically checking the proofs.

We have made a bug-fix release of TLAPS (version 1.4.4). In parallel, we are working on adding features for dealing with temporal properties, that is, fairness and liveness. We have implemented support for the ENABLED operator and the action composition operator in TLA+ proofs. This support is still experimental, but we hope to release a new version of TLAPS next year with these features.

6.3. Shared-memory concurrency

6.3.1. Instruction fetch in the ARMv8 architecture

Participants: Luc Maranget, Peter Sewell [University of Cambridge], Ben Simmer [University of Cambridge].

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. Luc Maranget is taking part in an international research effort to define the semantics of the computers of the multi-core era, and more generally of shared-memory parallel devices or languages, with a clear initial focus on devices.

Luc Maranget participates in project REMS, for *Rigorous Engineering for Mainstream Systems*, an EPSRC project led by Peter Sewell. This year Luc Maranget took part in a research effort that resulted in a paper entitled *ARMv8-A system semantics: instruction fetch in relaxed architectures*. This paper has been accepted for presentation at ESOP 2020. This paper introduces a robust model of instruction fetch and cache maintenance, a central aspect of a processor system's semantics, for ARMv8-A. Luc Maranget specifically extended the **litmus** and **diy** test generators so as to account for self-modifying code. He also performed part of the experiments that support the instruction fetch model.

6.3.2. An ARMv8 mixed-size memory model

Participants: Luc Maranget, Jade Alglave [ARM Ltd & University College London].

Jade Alglave and Luc Maranget have completed their work on a mixed-size version of the ARMv8 memory model. This model builds on the `aarch64.cat` model authored by Will Deacon (ARM Ltd). The model is now ready, and a paper has been written. They hope to work around certain intellectual property restrictions and to submit this paper for publication next year.

6.3.3. Work on diy

Participants: Luc Maranget, Jade Alglave [ARM Ltd & University College London], Antoine Hacquard.

The **diy** suite (for "Do It Yourself") 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. On distinctive feature of our system is Cat, a domain-specific language for memory models.

This year, new synchronisation primitives and instructions were added to various models. Some sizable developments occurred that facilitate the integration of mixed-size models into **herd**: a default definition of the same-instruction relation, which allows using mixed-size models on all tests; an automatic adjustment of the machine's elementary granularity, which facilitates massive testing; and the addition of equivalence classes and relations on them as basic values, which extends the expressiveness of Cat to some abstract mixed-size models.

During a 3-month internship, Antoine Hacquard (an EPITA second-year student) extended the complete tool suite to handle a new target, namely X86_64. The addition of this new target significantly enhances the **diy** tool suite, as X86_64 is a very popular architecture. Moreover, Antoine Hacquard implemented all memory access instructions for all sizes (from byte to quadword), which enabled us to design a mixed-size TSO model for this very popular architecture.

6.3.4. Unifying axiomatic and operational weak memory models

Participants: Quentin Ladeveze, Jean-Marie Madiot, Jade Alglave [ARM Ltd & University College London], Simon Castellan [Imperial College London].

Modern multi-processors optimize the running speed of programs using a variety of techniques, including caching, instruction reordering, and branch speculation. While those techniques are perfectly invisible to sequential programs, such is not the case for concurrent programs that execute several threads and share memory: threads do not share at every point in time a single consistent view of memory. A *weak memory model* offers only weak consistency guarantees when reasoning about the permitted behaviors of a program. Until now, there have been two kinds of such models, based on different mathematical foundations: axiomatic models and operational models.

Axiomatic models explicitly represent the dependencies between the program and memory actions. These models are convenient for causal reasoning about programs. They are also well-suited to the simulation and testing of *hardware* microprocessors.

Operational models represent program states directly, thus can be used to reason on programs: program logics become applicable, and the reasoning behind nondeterministic behavior is much clearer. This makes them preferable for reasoning about *software*.

Jean-Marie Madiot has been collaborating with weak memory model expert Jade Alglave and concurrent game semantics researcher Simon Castellan in order to unify these styles, in a way that attempts to combine the best of both approaches. The first results are a formalisation of TSO-style architectures using partial-order techniques similar to the ones used in game semantics, and a proof of a stronger-than-state-of-art "data-race freedom" theorem: well-synchronised programs can assume a strong memory model.

Since October 2019, Luc Maranget and Jean-Marie Madiot are advising a PhD candidate, Quentin Ladeveze. His goal is to further generalize and formalize weak memory models. This involves reasoning about linearizations of interdependent acyclic relations.

This is a first step towards tractable verification of concurrent programs, combining software verification using concurrent program logics, in the top layer, and hardware testing using weak memory models, in the bottom layer. Our hope is to leave no unverified gap between software and hardware, even (and especially) in the presence of concurrency.

7. Bilateral Contracts and Grants with Industry

7.1. Bilateral Contracts with Industry

7.1.1. The Caml Consortium

Participant: 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 9 member companies:

- Aesthetic Integration
- Citrix
- Docker
- Esterel Technologies
- Facebook
- Jane Street
- LexiFi
- Microsoft
- SimCorp

The Caml Consortium is being gradually phased out. In the future, we would like to replace it entirely with the OCaml Software Foundation, discussed below.

7.2. Bilateral Grants with Industry

7.2.1. The OCaml Software Foundation

Participants: Damien Doligez, Xavier Leroy.

The OCaml Software Foundation (OCSF),[3] 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.

[3] http://ocaml-sf.org/

7.2.2. Funding from Nomadic Labs

Nomadic Labs, a Paris-based company, has implemented the Tezos blockchain and cryptocurrency entirely in OCaml. This year, 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) and to hire Gabriel Radanne on a Starting Research Position (3 years).
- "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 years.
- "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).

7.2.3. Funding from the Microsoft-Inria joint lab

Funding from the Microsoft-Inria joint lab has allowed us to hire Ioannis Filippidis on a Starting Research Position (until March 2020) to work on the TLAPS system.

8. Partnerships and Cooperations

8.1. National Initiatives

8.1.1. ANR projects

8.1.1.1. Vocal

Participants: Armaël Guéneau, Xavier Leroy, François Pottier.

The "Vocal" project (2015–2020) aims at developing the first mechanically verified library of efficient general-purpose data structures and algorithms. It is funded by *Agence Nationale de la Recherche* under its "appel à projets générique 2015".

A first release of the library has been published in December 2018. It contains a small number of verified data structures, including resizable vectors, hash tables, priority queues, and Union-Find.

In 2019, progress was made on the definition of Gospel, a standard language for annotating OCaml programs with logical specifications, which could be understood and processed by several verification tools, including Why3 and CFML.

8.2. International Research Visitors

8.2.1. Visits of International Scientists

Jacques Garrigue (Nagoya University) is staying with our team in Paris from September 2019 to June 2020. He has long been one of the key designers and implementors of the OCaml type system. We are collaborating on the design of new language features and on a possible re-design of the type-checker implementation.

9. Dissemination

9.1. Promoting Scientific Activities

9.1.1. Scientific Events: Selection

9.1.1.1. Chair of Conference Program Committees

François Pottier was the program chair of the International Conference on Functional Programming (ICFP 2019) which took place in Berlin, Germany in August 2019.

9.1.1.2. Member of the Conference Program Committees

Xavier Leroy was on the program committee of PERR 2019, the 3rd Workshop on Program Equivalence and Relational Reasoning, part of the ETAPS 2019 joint conferences.

Xavier Leroy was on the program committee of FOSSACS 2020, the 23rd International Conference on Foundations of Software Science and Computation Structures, part of the ETAPS 2020 joint conferences.

Didier Rémy was a member of the program committee for FLOPS 2020, the 15th International Symposium on Functional and Logic Programming.

9.1.2. Journal

9.1.2.1. Member of the 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 board of Journal of Automated Reasoning.

François Pottier is a member of the ICFP steering committee and a member of the editorial boards of the Journal of Functional Programming and the Proceedings of the ACM on Programming Languages.

Until September 2019, Didier Rémy was a member of the ML Family workshop steering committee.

9.1.3. Scientific Expertise

Didier Rémy co-authored the Inria white book on Cybersecurity [17], [18] and a companion document of *Recommendations for Inria's Management*.

9.1.4. Research Administration

Damien Doligez chairs the Caml Consortium.

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 Inria's delegate in the MPRI's pedagogical and management board of the *Master Parisien de Recherche en Informatique* (MPRI).

Didier Rémy set up the Inria-Nomadic Labs partnership and is currently the co-chair of its steering committee.

9.2. Teaching - Supervision - Juries

9.2.1. Teaching

Master (M2): "Proofs of Programs", Jean-Marie Madiot, 18 HETD, MPRI, Université Paris Diderot, France.

Master (M2): "Programming shared memory multicore machines", Luc Maranget, 18 HETD, MPRI, Université Paris Diderot, France. Starting December 2019, Luc Maranget is in charge of this course.

Master (M2): "Functional programming and type systems", François Pottier, 18 HETD, MPRI, Université Paris Diderot, France.

Master (M2): "Functional programming and type systems", Didier Rémy, 18 HETD, MPRI, Université Paris Diderot, France. Didier Rémy is in charge of this course.

Licence (L3): Jean-Marie Madiot, "Introduction à l'informatique", 40 HETD, École Polytechnique, France.

Open lectures: Xavier Leroy, *Sémantiques mécanisées: quand la machine raisonne sur ses langages*, 19 HETD, Collège de France, France.

Summer school: Xavier Leroy, *Proving the correctness of a compiler*, 6 HETD, 2019 EUtypes summer school on Types for Programming and Verification, North Macedonia.

9.2.2. Supervision

PhD in progress: Frédéric Bour, “An interactive, modular proof environment for OCaml”, Université Paris Diderot, since October 2019, advised by François Pottier and Thomas Gazagnaire (Taridès).

PhD in progress: Basile Clément, “Domain-specific language and machine learning compiler for the automatic synthesis of high-performance numerical libraries”, École Normale Supérieure, since September 2018, advised by Xavier Leroy since October 2019.

PhD in progress: Nathanaël Courant, “Towards an efficient, formally-verified proof checker for Coq”, Université Paris Diderot, since September 2019, advised by Xavier Leroy.

PhD: Armaël Guéneau, “Mechanized Verification of the Correctness and Asymptotic Complexity of Programs”, Université Paris Diderot, defended on December 16, 2019 [11], advised by Arthur Charguéraud and François Pottier.

PhD in progress: Quentin Ladeveze, “Generic conditions for DRF-SC in axiomatic memory models”, Université Paris Diderot, since October 2019, advised by Luc Maranget and Jean-Marie Madiot.

PhD in progress: Glen Mével, “Towards a system for proving the correctness of concurrent Multicore OCaml programs”, Université Paris Diderot, since November 2018, advised by Jacques-Henri Jourdan and François Pottier.

PhD in progress: Thomas Williams, “Putting Ornaments into practice”, Université Paris Diderot, since September 2014, advised by Didier Rémy.

9.2.3. Juries

Xavier Leroy participated in the hiring committee for a professor position at the UFR d’Informatique of U. Paris Diderot.

Xavier Leroy chaired the jury for the Habilitation defense of Yann Régis-Gianas (U. Paris Diderot, November 2019).

Xavier Leroy was external reviewer for the PhD of Andrea Condoluci (U. Bologna, to be defended in 2020).

Jean-Marie Madiot served as an examiner at the computer science oral examination for the “second concours” of École Normale Supérieure de Lyon.

Didier Rémy participated in the hiring committee for an Inria Centrale-Supélec Chair in cybersecurity.

9.3. Popularization

9.3.1. Interventions

Xavier Leroy was on the committee of the 2019 Sephora Berrebi scholarship for women in mathematics and computer science. At the award ceremony (Paris, France, February 2019), he gave a popular science talk on Grace Hopper and the birth of the compiler.

Xavier Leroy gave popularization talks on deductive software verification at the Aerospace Lab Conferences of ONERA (Palaiseau, France, June 2019) and at the Summer 2019 BOB conference (Berlin, Germany, August 2019).

10. Bibliography

Major publications by the team in recent years

[1] J. ALGLAVE, L. MARANGET, M. TAUTSCHNIG. *Herding cats: modelling, simulation, testing, and data-mining for weak memory*, in "ACM Transactions on Programming Languages and Systems", 2014, vol. 36, n^{o} 2, http://dx.doi.org/10.1145/2627752

[2] T. BALABONSKI, F. POTTIER, J. PROTZENKO. *The design and formalization of Mezzo, a permission-based programming language*, in "ACM Transactions on Programming Languages and Systems", 2016, vol. 38, n^{o} 4, pp. 14:1–14:94, http://doi.acm.org/10.1145/2837022

[3] K. CHAUDHURI, D. DOLIGEZ, L. LAMPORT, S. MERZ. *Verifying Safety Properties With the TLA+ Proof System*, in "Automated Reasoning, 5th International Joint Conference, IJCAR 2010", Lecture Notes in Computer Science, Springer, 2010, vol. 6173, pp. 142–148, http://dx.doi.org/10.1007/978-3-642-14203-1_12

[4] J. CRETIN, D. RÉMY. *System F with Coercion Constraints*, in "CSL-LICS 2014: Computer Science Logic / Logic In Computer Science", ACM, 2014, http://dx.doi.org/10.1145/2603088.2603128

[5] A. GUÉNEAU, J.-H. JOURDAN, A. CHARGUÉRAUD, F. POTTIER. *Formal Proof and Analysis of an Incremental Cycle Detection Algorithm*, in "Interactive Theorem Proving", J. HARRISON, J. O'LEARY, A. TOLMACH (editors), Leibniz International Proceedings in Informatics, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, September 2019, vol. 141, https://hal.inria.fr/hal-02167236

[6] J.-H. JOURDAN, V. LAPORTE, S. BLAZY, X. LEROY, D. PICHARDIE. *A Formally-Verified C Static Analyzer*, in "POPL'15: 42nd ACM Symposium on Principles of Programming Languages", ACM Press, January 2015, pp. 247-259, http://dx.doi.org/10.1145/2676726.2676966

[7] X. LEROY. *A formally verified compiler back-end*, in "Journal of Automated Reasoning", 2009, vol. 43, n^{o} 4, pp. 363–446, http://dx.doi.org/10.1007/s10817-009-9155-4

[8] X. LEROY. *Formal verification of a realistic compiler*, in "Communications of the ACM", 2009, vol. 52, n^{o} 7, pp. 107–115, http://doi.acm.org/10.1145/1538788.1538814

[9] G. MÉVEL, J.-H. JOURDAN, F. POTTIER. *Time Credits and Time Receipts in Iris*, in "European Symposium on Programming", Lecture Notes in Computer Science, Springer, April 2019, vol. 11423, pp. 3-29 [*DOI :* 10.1007/978-3-030-17184-1_1], https://hal.archives-ouvertes.fr/hal-02183311

[10] N. POUILLARD, F. POTTIER. *A unified treatment of syntax with binders*, in "Journal of Functional Programming", 2012, vol. 22, n^{o} 4–5, pp. 614–704, http://dx.doi.org/10.1017/S0956796812000251

Publications of the year

Doctoral Dissertations and Habilitation Theses

[11] A. GUÉNEAU. *Mechanized Verification of the Correctness and Asymptotic Complexity of Programs*, Université de Paris, December 2019, https://hal.inria.fr/tel-02437532

Articles in International Peer-Reviewed Journals

[12] A. CHARGUÉRAUD, F. POTTIER. *Verifying the Correctness and Amortized Complexity of a Union-Find Implementation in Separation Logic with Time Credits*, in "Journal of Automated Reasoning", March 2019, vol. 62, n^{o} 3, pp. 331–365 [*DOI :* 10.1007/s10817-017-9431-7], https://hal.inria.fr/hal-01652785

[13] P. E. DE VILHENA, F. POTTIER, J.-H. JOURDAN. *Spy Game: Verifying a Local Generic Solver in Iris*, in "Proceedings of the ACM on Programming Languages", January 2020, n^{o} 4 [*DOI :* 10.1145/3371101], https://hal.archives-ouvertes.fr/hal-02351562

Invited Conferences

[14] X. LEROY. *In Search of Software Perfection : An introduction to deductive software verification*, in "BOB Summer 2019 Konferenz", Berlin, Germany, August 2019, https://hal.inria.fr/hal-02392114

International Conferences with Proceedings

[15] A. GUÉNEAU, J.-H. JOURDAN, A. CHARGUÉRAUD, F. POTTIER. *Formal Proof and Analysis of an Incremental Cycle Detection Algorithm : (extended version)*, in "Interactive Theorem Proving", Portland, United States, J. HARRISON, J. O'LEARY, A. TOLMACH (editors), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, September 2019, n^o 141, https://hal.inria.fr/hal-02167236

[16] G. MÉVEL, J.-H. JOURDAN, F. POTTIER. *Time Credits and Time Receipts in Iris*, in "European Symposium on Programming", Prague, Czech Republic, Lecture Notes in Computer Science, Springer, April 2019, vol. 11423, pp. 3-29 [*DOI :* 10.1007/978-3-030-17184-1_1], https://hal.archives-ouvertes.fr/hal-02183311

Scientific Books (or Scientific Book chapters)

[17] S. KREMER, L. MÉ, D. RÉMY, V. ROCA. *Cybersecurity : Current challenges and Inria's research directions*, Inria white book, Inria, January 2019, n^{o} 3, 172 p. , https://hal.inria.fr/hal-01993308

[18] S. KREMER, L. MÉ, D. RÉMY, V. ROCA. *Cybersécurité : Défis actuels et axes de recherche à l'Inria*, Inria white book, Inria, May 2019, n^{o} 3, 18 p. , https://hal.inria.fr/hal-02414281

[19] X. LEROY. *Le logiciel, entre l'esprit et la matière*, Leçons inaugurales du Collège de France, Fayard, April 2019, vol. 284, https://hal.inria.fr/hal-02370113

[20] X. LEROY. *Le logiciel, entre l'esprit et la matière : Leçon inaugurale prononcée au Collège de France le jeudi 15 novembre 2018*, OpenEdition Books, December 2019, https://hal.inria.fr/hal-02405754

[21] X. LEROY. *Software, between mind and matter*, Inaugural lecture at Collège de France, Collège de France, November 2019, https://hal.inria.fr/hal-02392159

Research Reports

[22] X. LEROY, D. DOLIGEZ, A. FRISCH, J. GARRIGUE, D. RÉMY, J. VOUILLON. *The OCaml system release 4.09: Documentation and user's manual*, Inria, September 2019, pp. 1-789, https://hal.inria.fr/hal-00930213

[23] X. LEROY. *The CompCert C verified compiler: Documentation and user's manual : Version 3.6*, Inria, September 2019, pp. 1-78, https://hal.inria.fr/hal-01091802

Other Publications

[24] C. MOREL. *Type inference and modular elaboration with constraints for ML extended with type abbreviations*, Université Paris Diderot-Paris 7, September 2019, https://hal.inria.fr/hal-02361707

References in notes

[25] L. LAMPORT. *How to write a 21st century proof*, in "Journal of Fixed Point Theory and Applications", 2012, vol. 11, pp. 43–63, http://dx.doi.org/10.1007/s11784-012-0071-6