RESEARCH CENTRE

Paris

IN PARTNERSHIP WITH:

Collège de France

2021

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 Rémy [Inria, Senior Researcher, HDR]
- Léo Stefanesco [Collège de France, Researcher, until Sep 2021]

Faculty Member

- Xavier Leroy [Collège de France, Professor]

PhD Students

- Clément Blaudeau [Université de Paris, from Oct 2021]
- Frédéric Bour [Tarides, CIFRE]
- Basile Clement [Inria]
- Nathanaëlle Courant [École Normale Supérieure de Paris]
- Quentin Ladeveze [Inria]
- Alexandre Moine [Inria, from Oct 2021]
- Glen Mével [Inria]
- Thomas Refis [Tarides]
- Paulo Emílio de Vilhena [Inria]

Technical Staff

- Florian Angeletti [Inria, Engineer]
- Sébastien Hinderer [Inria, Engineer]

Interns and Apprentices

- Clément Blaudeau [Inria, from Feb 2021 until Aug 2021]
- Alexandre Moine [Inria, from Mar 2021 until Aug 2021]
- Émile Trotignon [Inria, from Feb 2021 until Jun 2021]

Administrative Assistant

- Hélène Milome [Inria]

Visiting Scientist

- Mário Pereira [Universidade Nova de Lisboa, from Oct 2021]

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 [29]. 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 [33] and the Verasco abstract interpretation-based static analyzer [8]. 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

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

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.

2.5 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 New software and platforms

5.1 New software

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

URL: `https://ocaml.org/`

Publications: hal-03145030, hal-01929508, hal-03125031, hal-00772993, hal-00914493, hal-00914560, inria-00074804, hal-01499973, hal-01499946

Contact: Damien Doligez

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

5.1.2 Compcert

Name: The CompCert formally-verified C compiler

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

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

Release Contributions: 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).

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

5.1.3 Diy

Name: Do It Yourself

Keyword: Parallelism

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

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

Contact: Luc Maranget

Participants: Jade Alglave, Luc Maranget

Partner: University College London UK

5.1.4 Menhir

Keywords: Compilation, Context-free grammars, Parsing

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

Publications: hal-01633123, hal-01417004

Contact: Francois Pottier

5.1.5 CFML

Name: Interactive program verification using characteristic formulae

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

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

URL: `http://www.chargueraud.org/softs/cfml/`

Contact: Arthur Chargueraud

Participants: Arthur Chargueraud, Armaël Guéneau, Francois Pottier

5.1.6 TLAPS

Name: TLA+ proof system

Keyword: Proof assistant

Functional 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.

News of the Year: In 2020, we published a minor release, fixing some issues notably for the SMT backend. Substantial work was devoted to supporting liveness reasoning, in particular proofs about the ENABLED and action composition constructions of TLA+. We also prepared support for current versions of the Isabelle back-end prover.

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

Contact: Stephan Merz

Participants: Damien Doligez, Stephan Merz, Ioannis Filippidis

Partner: Microsoft

5.1.7 ZENON

Name: The Zenon automatic theorem prover

Keywords: Automated theorem proving, First-order logic

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

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

URL: `http://zenon-prover.org/`

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

Contact: Damien Doligez

Participant: Damien Doligez

5.1.8 hevea

Name: hevea is a fast latex to html translator.

Keywords: LaTeX, Web

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

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

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

URL: `http://hevea.inria.fr/`

Author: Luc Maranget

Contact: Luc Maranget

6 New results

6.1 Software

6.1.1 The CompCert formally-verified compiler

Participants: Xavier Leroy, Michael Schmidt *(AbsInt GmbH)*, Bernhard Schommer *(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 [33]. 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 [32], and a front-end, which translates the "CompCert C" subset of C to Cminor. The compiler is written mostly within the specification language of the Coq proof assistant, out of which Coq's extraction facility generates executable OCaml code. The compiler comes with a 100000-line machine-checked proof of semantic preservation, establishing that the generated assembly code executes exactly as prescribed by the semantics of the source C program.

This year, we improved the handling of *bit fields* as members of structure and union types. Previously, bit fields were implemented by an unverified source-to-source translation that would turn bit field operations into bit-level manipulations over integers. Now, bit field declarations and manipulations are given fully formal semantics as part of the CompCert C input language, and their compilation into bit-level operations is proved correct. The new, verified implementation of bit fields not only increases confidence, but also enables CompCert to support the memory layout of bit fields specified in the ELF ABI standards, something that could not be achieved with the earlier, unverified implementation.

Other improvements on CompCert include:

- reduced compilation times, thanks to a new implementation of the "binary trie" data structure (§6.4.1);
- support for the AArch64 architecture under macOS, also known as "Apple silicon";
- the branch tunneling optimization was made slightly more effective;
- the `clightgen` tool, which produces Coq representations of the Clight intermediate representation of programs (allowing CompCert to be used in conjunction with the Verified Software Toolchain [31]), was extended to also produce Coq representations of the CompCert C abstract syntax trees.

We released two versions of CompCert incorporating these improvements: version 3.9 in May 2021 and version 3.10 in November 2021.

6.1.2 The OCaml system

Participants: Florian Angeletti, Frédéric Bour, Damien Doligez, Sébastien Hinderer, Xavier Leroy, Luc Maranget, Thomas Refis, David Allsop *(Cambridge University)*, Stephen Dolan *(Cambridge University)*, Jacques Garrigue *(University of Nagoya)*, Sadiq Jaffer *(OCaml Labs)*, Tom Kelly *(OCaml Labs)*, Nicolás Ojeda Bär *(Lexifi)*, KC Sivaramakrishnan *(IIT Madras)*, Gabriel Scherer *(EPI Parsifal)*, Mark Shinwell *(Jane Street)*, Leo White *(Jane Street)*, Jeremy Yallop *(Cambridge University)*.

This year, we released two major versions of the OCaml system: version 4.12.0 in February and version 4.13.0 in September. We also released three minor versions (4.11.2, 4.12.1, 4.13.1) with bug fixes.

Much of our work on OCaml this year was dedicated to preparing the merge of Multicore OCaml, the extension of OCaml with shared-memory parallelism developed at OCaml Labs. In particular, we rewrote parts of the OCaml run-time system to avoid relying on the page table, a feature of the current GC that Multicore OCaml no longer supports. We also added a run-time checker for "naked pointers outside the heap", another feature of the current OCaml run-time system that goes away in Multicore OCaml.

Multicore OCaml requires the ability to interrupt threads at predictable points in order to perform stop-the-world garbage collection. To this end, we added a new pass to the native-code compiler that inserts “safe points” (polls for interrupt requests) in the generated code, guaranteeing that all executions cross a safe point after a finite amount of time. This relies on a clever static analysis for loops, both intra- and inter-procedural.

Other novelties in the 4.12 and 4.13 releases include:

- The “Apple silicon” machines (ARM 64-bit processors under macOS) are now supported.
- Abstract type constructors can now be declared injective, making them more usable in the context of generalized algebraic data types (GADTs).
- It is now possible to give explicit names for existentially-quantified type variables inside GADT patterns.
- Module types can now be substituted inside signatures.
- Compiler warnings can be selected by mnemonic names in addition to numbers.
- New algorithms for data flow analyses in the native-code compiler were implemented, avoiding previous worst cases that were exponential in the nesting of loops.
- Signal handling was redesigned and reimplemented to make it more robust and compatible with parallelism.
- The best-fit memory allocator, introduced in 2020, is now the default allocator.
- A new algorithm generates more user-friendly error messages for type errors involving functor applications or inclusions with multiple arguments.
- A new API was introduced for ephemerons. It is more restricted but gives much better guarantees with respect to determinism, which will be especially useful in Multicore Ocaml.
- More than 60 usability improvements were implemented, ranging from better error and warning messages to new standard library functions to performance improvements.
- About 70 bugs were fixed.

6.1.3 The Menhir parser generator

Participants: Frédéric Bour, François Pottier, Émile Trotignon.

Since 2005, François Pottier has been developing Menhir, an LR(1) parser generator for OCaml. Menhir is used both within our group (where it is exploited, in particular, by the OCaml and CompCert compilers) and by many users in the OCaml community.

This year, two major improvements have been introduced:

- The reachability algorithm designed and developed by Frédéric Bour [18] has been integrated. This algorithm replaces an earlier algorithm, whose time and space requirements it divides by two or three orders of magnitude (§6.2.4).
- Menhir’s code back-end has been re-implemented from scratch. The original back-end, implemented in 2005 by François Pottier, produced OCaml code that contained numerous unsafe type casts, for efficiency reasons. The new back-end, implemented by Émile Trotignon and François Pottier, produces well-typed code. A generalized algebraic data type (GADT) is used to describe the shape of the parser’s stack and its relation with the current state of the automaton. Efficiency is not compromised. On the contrary, the safety afforded by strong type-checking allows new optimizations to be introduced.

6.1.4 The `diy` tool suite

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

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. On distinctive feature of our system is Cat, a domain-specific language for memory models.

This year, Luc Maranget extended the tools by adding some features for the X86_64 architecture: this served to support ongoing research work [21]. Extensions were made in particular to **klitmus**, a tool that produces executable kernel modules, so as to test system-level memory model features. **klitmus** now accepts assembly programs as its input, while it was previously limited to accepting kernel-style C programs. Jade Alglave and Luc Maranget collaborated to finalize and integrate the "KVM" extension (support for virtual memory). Moreover, many authors (most of whom work at ARM Ltd.) have made various contributions:

- "Bare-metal" execution of tests, without OS support.
- Refinement of test generation so as to better reflect the semantics of instructions.
- Support for self-modifying code in **herd**.

6.2 Programming language design and implementation

6.2.1 Formalizing and improving OCaml modules

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

Clément Blaudeau did a master's internship in the spring of 2021, from February to July, and started a PhD in October, co-advised by Didier Rémy and Gabriel Radanne.

The well-known paper *F-ing modules* [38] gives a semantics of ML modules by elaboration into System F. During his internship, Clément proposed a similar translation for the "generative" fragment of the OCaml module system. (In the generative fragment, every application of a functor produces fresh, incompatible types.) He introduced a new intermediate type system that serves as a specification and helps establish a link between the standard but somewhat ad hoc presentation of OCaml modules (which is based on "paths") and the "F-ing modules" presentation (which is based on existential types).

Since October, Clément Blaudeau has applied the same method to the more complex "applicative" fragment, where two identical applications of a functor produce compatible types, and has discovered interesting simplifications of the "F-ing modules" approach. In particular, the intermediate type system, which serves as a specification but does not elaborate terms, can use a (non-constructive) skolemization rule, which allows for abstraction a posteriori.

6.2.2 Designing and formalizing modular implicits

Participants: Thomas Refis, Didier Rémy, Leo White *(Jane Street).*

A few years ago, White *et al.* suggested a way to add ad-hoc polymorphism to OCaml, in the form of modular implicits [40]. This new language feature can in fact be viewed as the combination of two independent parts. The first component, *modular explicits*, is an extension to ML's core language with a seemingly dependent arrow type. The second component, an *implicit resolution mechanism*, finds suitable values for omitted arguments in function applications, based on the type constraints that apply to these missing arguments.

Continuing last year's work on the formalization of modular explicits, this year, we studied the interaction and overlap between first-class modules and the *module-dependent functions* introduced by modular explicits.

6.2.3 Improving OCaml's treatment of higher-rank polymorphism

Participants: Thomas Refis, Didier Rémy, Gabriel Scherer, Leo White *(Jane Street)*.

OCaml's type-checker relies on integer *levels* to efficiently perform operations such as *generalization*, *instantiation*, and ensuring that a local type variable does not escape its scope. Unfortunately, this mechanism cannot currently be used to enforce the well-scopedness of higher-rank types, which today can appear in OCaml when using polymorphic methods and polymorphic record fields, and in the future will also appear when using *module-dependent functions*.

In 2021, we have been working on extending this mechanism to also handle higher-rank types, with the aim of making the implementation of the OCaml type-checker simpler and more efficient.

6.2.4 A faster reachability algorithm for LR(1) parsers

Participants: Frédéric Bour, François Pottier.

Last year, Frédéric Bour did preliminary work on an analysis of the structure of the stack of an LR(1) parser. He investigated a new approach to constructing syntax error messages, extending the approach proposed in 2016 by François Pottier [37], with the aim of providing finer-grained messages and a different developer experience. Although this work has not yet come to fruition, it led Frédéric to proposing a new algorithm for analysing reachability in LR(1) automata.

The reachability problem is to determine under what conditions each transition of an LR(1) automaton can be taken, that is, which (minimal) input fragment and which lookahead symbol allow taking this transition. This new algorithm has been implemented by Frédéric in the Menhir parser generator, increasing performance (compared to the previous algorithm) by up to three orders of magnitude on real-world grammars. This result has been presented at SLE 2021 [18].

6.2.5 Debootstrapping the OCaml compiler

Participants: Nathanaëlle Courant, Julien Lepiller *(Yale University)*, Gabriel Scherer.

The OCaml compiler is built using a previous, precompiled version of itself. As shown by Thompson [39], this can cause bugs or even intentional backdoors to self-reproduce in the compiled version of the compiler. Based on earlier unpublished work, Nathanaëlle Courant and Gabriel Scherer wrote (with the help of Julien Lepiller) a *debootstrapped* version of the OCaml compiler, which is able to compile OCaml without using any binary (i.e., non-source) file. A paper on this topic has been accepted for presentation at the conference ‹Programming› 2022.

6.3 Shared-memory concurrency

6.3.1 Axiomatic memory models

Participants: Jade Alglave *(ARM Ltd–University College London)*, Will Deacon *(Google Inc.)*, Antoine Hacquard *(EPITA, Paris)*, 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 optimizations 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. This year, the paper "Armed cats: formal concurrency modeling at Arm" was published in *Transactions on Programming Languages and Systems* (TOPLAS) [13]. The paper presents an extension of the AArch64 (ARMv8) and x86 (TSO) models to *mixed-size* accesses. This extension is of practical interest, as many programs, in particular system programs, access memory using different sizes—*e.g.*, by mixing byte and word accesses—which may overlap. Our work introduces a general treatment of memory model extensions, producing three provably equivalent alternative formulations. Our results are confirmed via vast experimental campaigns. The paper includes work by Antoine Hacquard, who was an intern at Cambium in 2019.

Concurrently, Jade Alglave and Luc Maranget are designing another extension to the ARM memory model, namely virtual memory. This work in progress aims to account for the interaction of the memory model and of virtual memory. Understanding this interaction is an absolute necessity in order to implement correct operating systems. Luc Maranget is specifically in charge of software development and of experiments. The amount of experimental work required by this project appears to be much more significant than in similar previous projects, due to the numerous features of virtual memory, such as permissions, translation lookaside buffers (TLBs), alternative mappings of the same physical page, and more. As a consequence, although the first half of the year has been devoted to this topic, we have not yet reached a stage where a paper can be submitted. However, a satisfactory model has been designed, and an internal document has been written. We plan to pursue this work next year, aiming for a publication.

6.3.2 The Intel x86 memory model

Participants: Azalea Raad *(Imperial College, London)*, Luc Maranget, Viktor Vafeiadis *(MPI-SWS)*.

The existing semantic formalizations of the Intel x86 architecture cover only a small fragment of the features that are relevant to the consistency semantics of multi-threaded programs as well as the persistency semantics of programs that use non-volatile memory. This year, Raad, Maranget and Vafeiadis extended these formalizations to cover: (1) non-temporal writes, which provide higher performance and are used to ensure that updates are flushed to memory; (2) reads and writes to other Intel x86 memory types, namely "uncacheable", "write-combined", and "write-through"; as well as (3) the interaction between these features. Luc Maranget extended the **herd** and **litmus** tools with support for these new features (§6.1.4), performed a large set of experiments, and contributed to the design of a new axiomatic model. This work has been accepted for presentation at the conference POPL 2022 [21].

6.3.3 A generic proof of DRF-SC

Participants: Quentin Ladeveze, Luc Maranget, Jean-Marie Madiot.

Quentin Ladeveze, who is co-advised by Luc Maranget and Jean-Marie Madiot, has been working on a memory-model-independent proof of the fact that a certain set of conditions imply the DRF-SC property.

A model with this property guarantees that every race-free execution behaves like a sequentially consistent execution.

Last year, Quentin Ladeveze formalized a memory model that describes the behavior of C/C++11 programs and a proof that this model enjoys the DRF-SC property. These results were presented at JFLA 2021 [23].

This year, Quentin Ladeveze worked on extending this proof to other memory models. The major difficulty of this task is to handle the memory models that use more complex definitions of the dependency relation between two events on the shared memory to forbid so-called out-of-thin-air executions. These more complex definitions are more accurate and allow more optimizations, but they make it much more difficult to change the value read at some point in the execution to obtain another execution of the same program. Yet, linking two executions of the same program by changing a read value is central to the proof of the DRF-SC guarantee for models such as ARMv7. Thus, efforts have been focused on defining a set of necessary conditions guaranteeing that the dependency relation is strong enough to allow this linking between executions, and the integration of more realistic models in the generic DRF-SC proof. This work has been inspired by similar techniques used in compiler correctness proofs for the C/C++11 memory model, where the gap between the coarse-grained dependency relation of the C/C++11 model and the fine-grained dependency relation of hardware memory models needed to be addressed. This ongoing work has not been published; an internal document is being prepared. As of January 2021, Quentin Ladeveze abandoned his PhD thesis.

6.4 Software specification and verification

6.4.1 An extensional, verified, efficient implementation of binary tries

Participants: Xavier Leroy, Andrew Appel *(Princeton University)*.

Finite maps (lookup tables) are a ubiquitous data structure. Binary tries are a simple yet effective implementation of finite maps indexed by positive integers. The CompCert verified compiler and the Verified Software Toolchain [31] make intensive use of binary tries. However, the implementation of binary tries used in CompCert and VST (prior to this work) has two drawbacks. First, it is not “extensional”, which means that two tries that map equal keys to equal values are not necessarily considered equal by Coq. This is very inconvenient. Second, it handles sparse maps inefficiently.

Andrew Appel and Xavier Leroy investigated a number of alternative implementations of binary tries to overcome these limitations. While Coq’s dependent types can be used to enforce extensionality, they result in inefficient evaluation within the Coq proof assistant; yet, VST requires efficient evaluation. In response, Appel and Leroy developed *canonical binary tries*, a first-order representation of binary tries that enforces extensionality by construction and reduces the memory usage of sparse maps. The design, verification, and performance evaluation of canonical binary tries appear in a paper that has been submitted for publication [25].

6.4.2 Towards an efficient, verified proof checker for Coq

Participants: Nathanaëlle Courant.

Nathanaëlle Courant, who is advised by Xavier Leroy, has been working on 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 worked on writing an efficient convertibility test for the λ-calculus extended with data constructors and pattern matching. This algorithm attempts in parallel to test the convertibility of function arguments and the convertibility of the whole term. She is also beginning to work on a Coq proof of the correctness of this convertibility test.

6.4.3 Verification of tensor compilers

Participants: Basile Clement.

Basile Clement, who is advised by Xavier Leroy, has been working on translation validation for tensor compilers, and in particular for 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 the correctness of the code generated by the compiler relative to the specification.

This year, Basile worked on a translation-validation approach to the topic, based on a refinement mapping technique: each assignment in the generated code is mapped to a specific instance of a definition in the specification. He developed a Coq proof of the formal correctness of his approach, as well as a practical tool to verify Halide programs. This work has been the topic of a paper that has been accepted for presentation at OOPSLA 2022.

6.4.4 Modular coinduction for program equivalence

Participants: Jean-Marie Madiot, Damien Pous *(CNRS)*, Davide Sangiorgi *(Inria team FOCUS)*.

When one needs to prove that a program satisfies a specification or that an optimized program behaves in the same way as an unoptimized program, the propery of interest is *program equivalence*. One of the techniques used to establish program equivalence is the *bisimulation* proof method, a coinductive construction of relations on programs that originated in concurrency theory for the study of process calculi. Bisimulations are also used in practice, for example as elemental blocks in the soundness proof of the CompCert certified optimizing compiler.

The bisimulation proof method can be enhanced by employing "bisimulation up-to" techniques. A comprehensive theory of such enhancements has been developed for first-order transition systems and bisimulations, based on an abstract theory of fixed points and compatible functions.

in previous work, Jean-Marie Madiot, Damien Pous, and Davide Sangiorgi have transported this theory onto languages whose transition systems and bisimulations go beyond those of first-order models. The approach consists in exhibiting fully abstract translations of the more sophisticated models onto first-order models. This allows reusing the large corpus of techniques that are available for first-order systems. The only ingredient that must be manually supplied is the compatibility of the basic techniques that are specific to the new languages. The method has been investigated for the π-calculus, the λ-calculus, and for a λ-calculus with references.

This previous work introduces a notion "compatibility up-to" in order to assemble several techniques whose soundness depends on each other, in a "mutually coinductive" way. This has led to the development of the notion of "companion" as a cleaner way of doing the same thing and more. In a more recent journal paper [17], Madiot, Pous and Sangiorgi have upgraded this previous work to use the new "companion" construction. This paper also presents new results and new refactorings of previous proofs.

6.4.5 An object semantics for certified layers

Participants: Jérémie Koenig *(Yale University)*, Paul-André Mellies *(IRIF, CNRS)*, Arthur Oliveira Vale *(Yale University)*, Zhong Shao *(Yale University)*, Léo Stefanesco.

Certified layers are a methodology for developing certified programs. The idea is to slice the system in layers, which can be composed vertically and horizontally. It can be desirable to specify objects only through their observable behavior instead of their internal state.

In a paper that will be presented at POPL 2022 [36], we develop a new semantics for certified layers, inspired by Uday Reddy's work on an object semantics of Algol. Reddy's work itself was based on Girard's coherence spaces, where a concurrent object is specified as a set of allowed traces. Our paper gives an equivalent presentation, based on game semantics.

6.4.6 An interactive, modular proof environment for OCaml

Participants: Frédéric Bour, François Pottier.

Frédéric Bour, who is advised by François Pottier, has been working on an interactive tool for verifying properties of OCaml programs. The starting point is an adaptation of the methodology developed in the VeriFast verifier to OCaml programs.

The tool currently deals with a hypothetical "Mini-ML" language, so as to concentrate on the core problems and avoid the incidental complexity that comes with a full-blown programming language. This mini-language features parametric polymorphism, recursive data types, and a built-in separation logic.

Verification is handled by two distinct layers: an ad-hoc verifier based on separation logic, which deals with mutable state, and the Z3 theorem prover, which deals with first-order logical formulae.

When the program cannot be verified, a symbolic approximation of its state is reconstructed and displayed via a user interface that resembles a traditional runtime debugger.

This tool has been used to verify a simple FIFO queue backed by a mutable linked list. A specification was initially expressed using inductive lists; later on, a different specification was written, based on built-in logical sequences; the new specification allows stronger automation.

Future work includes extending the logic and the tool with more features, such as arrays and modules; porting the tool to work directly on a subset of OCaml; and introducing a means of performing partial or gradual program verification, that is, of verifying only some aspects of a program.

6.4.7 Specification and verification of a transient stack

Participants: Arthur Charguéraud *(Inria team Cassis)*, Alexandre Moine, François Pottier.

Between March and August 2021, Alexandre Moine's M2 internship was co-supervised by Arthur Charguéraud and François Pottier. Alexandre used CFML2 (Charguéraud's implementation of Separation Logic inside the Coq proof assistant) to specify and verify the functional correctness and time complexity of a transient stack. A transient data structure is a package of an ephemeral data structure, a persistent data structure, and fast conversions between them. Transient data structures are used at scale in some programming languages, such as Clojure and JavaScript. The transient stack studied by Alexandre is a scaled-down version of Sek, a general-purpose sequence data structure. Internally, it relies on fixed-capacity arrays, or chunks, which can be shared between several ephemeral and persistent stacks. Dynamic tests are used to determine whether a chunk can be updated in place or must be copied: a chunk can be updated if it is uniquely owned or if the update is monotonic. There are very few examples of verified transient data structures in the literature, let alone examples of transient data structures whose correctness and time complexity are verified, so we believe that this is an interesting contribution. This result has been accepted for presentation at the conference Certified Programs and Proofs (CPP) 2022 [20] and has received a Distinguished Paper Award (out of three).

6.4.8 A separation logic for effect handlers

Participants: Paulo Emílio de Vilhena, François Pottier.

Paulo Emílio de Vilhena, who is advised by François Pottier, aims to devise a separation logic with support for verifying programs that exploit effect handlers. In January 2021, he gave a talk at POPL 2021 where he presented their paper on this topic [15].

In the following months, Paulo continued working on this project. He conducted several case studies where he applied his program logic to the verification of specific example programs. In particular, he verified the following programs:

- an implementation of ML references using effect handlers;
- a simple SAT solver that uses multi-shot continuations to perform backtracking;
- a minimalist automatic differentiation library, implemented using mutable state and effect handlers.

The last case study is the subject of a paper that has been submitted for publication in a journal.

During the summer, Paulo was invited by Amin Timany to visit the LogSem research group, which is led by Lars Birkedal at the University of Aarhus (Denmark). During the visit, he worked with Timany and Birkedal on the formalization of several type systems for effect handlers. In particular, they designed a relational model for a type system that supports general references and a simple form of effect polymorphism. Such a model is useful not only to prove the correctness of the type system but also to prove that two programs, possibly exploiting effect handlers and mutable state, are observationally equivalent.

In July 2021, Paulo attended a Dagstuhl Seminar on "Scalable Handling of Effects". There, he presented his ongoing work to an expert audience. The abstract of his talk appears in the online report of the seminar.

6.4.9 A separation logic for heap space under garbage collection

Participants: 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.

To address this problem, Jean-Marie Madiot and François Pottier propose SL$\diamond$, a Separation Logic equipped with a ghost operation for "logical deallocation". During program verification, the programmer (or, more accurately, the author of the proof) can argue that an object must be unreachable and therefore can be logically deallocated, an operation that creates a number of space credits. To allow proving unreachability facts, the logic maintains pointed-by assertions that record the immediate predecessors of every object. Although this new Separation Logic is still not as easy to use as we would like, we believe that it is an important step forward. This result has been accepted for presentation at the conference POPL 2022 [16].

6.4.10 A separation logic for Multicore OCaml

Participants: Glen Mével, Jacques-Henri Jourdan *(LMF, ENS Paris-Saclay, Université Paris-Saclay, CNRS)*, François Pottier.

Glen Mével, who is co-advised by Jacques-Henri Jourdan and François Pottier, has been working on designing Cosmo, 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.

In 2020, Glen used Cosmo to verify the functional correctness of a concurrent queue data structure. This work has since been completed, published and presented by Glen at ICFP 2021 [19].

6.4.11 A separation logic for time complexity with debits and credits

Participants: Glen Mével, Jacques-Henri Jourdan *(LMF, ENS Paris-Saclay, Université Paris-Saclay, CNRS)*, François Pottier.

In 2018, during Glen's research internship, Glen, Jacques-Henri and François reconstructed *time credits* in the settings of Iris, a general-purpose separation logic [34]. Time credits are a logical tool that allows bounding the (amortized) time complexity of a program. On top of time credits, Glen, Jacques-Henri and François proposed a specification and a verified implementation of *thunks* (also known as *suspensions*), a simple mutable data structure that plays a fundamental role in lazy functional programming. The specification involves *debits*: in short, a thunk (a suspended computation) carries a debt that must be paid off before the thunk can be forced (evaluated).

This year, Glen, Jacques-Henri and François improved and extended this specification of thunks. The new specification introduces a new reasoning rule, which allows anticipating a debt. This rule is required to verify the amortized time complexity of a lazy persistent queue devised by Okasaki [35]. Proving that the new specification is satisfied required deep changes to the proof. This ongoing work is currently being completed and should be submitted for publication in 2022.

6.4.12 A separation logic for concrete distributed programs with abstract models

Participants: Lars Birkedal *(Aarhus University)*, Léon Gondelman *(Aarhus University)*, Abel Nieto *(Aarhus University)*, Simon Oddershede Gregersen *(Aarhus University)*, Léo Stefanesco, Amin Timany *(Aarhus University)*.

Traditional Hoare logics guarantee that a closed program that has been proved correct will not crash. This means that, in order to check that the specification of one module is "good" in some sense, one must write a client of this module whose safety implies the desired properties of the module. This is somewhat indirect.

Trillium, in contrast, is a logic in which one can directly express the correctness of a module. The desired behavior is described by a state transition system M, and the logic guarantees that *any* client which uses this module is simulated by the state transition system M. As a case study, we prove that an implementation of the Paxos distributed consensus algorithm implements its specification, expressed in TLA+.

As another application, this logic allows proving that a concurrent program (with shared memory) terminates, under the assumption that the scheduler is fair.

These results appear in a draft paper and in Léo Stefanesco's PhD thesis.

7 Bilateral contracts and grants with industry

7.1 Bilateral contracts with industry

7.1.1 The Caml Consortium

Participants: Damien Doligez.

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

Damien Doligez chairs the Caml Consortium.

The Consortium currently has 9 member companies:

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

In the future, we would like to replace the Caml Consortium with the OCaml Software Foundation, discussed below. For the moment, however, the Caml Consortium remains alive, because the licensing conditions that it offers are unique.

7.1.2 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.

7.2 Bilateral grants with industry

7.2.1 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.

7.2.2 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 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).

8 Partnerships and cooperations

8.1 International initiatives

8.1.1 Visits of international scientists

Participants: Mário Pereira.

Mário Pereira, a postdoctoral researcher at Universidade Nova de Lisboa (Portugal), visited our team in November and December 2021. Mário is funded by an MSCA grant. We collaborated on the design of Gospel, a specification language for OCaml. We hope to continue this collaboration in the future.

8.1.2 Visits to international teams

Participants: Paulo Emílio de Vilhena, Glen Mével.

Paulo Emílio de Vilhena spent three months (from mid-June to mid-September) at Aarhus University (Denmark), where he visited the LogSem research group (§6.4.8). His visit was funded by Aarhus University.

Glen Mével spent six months (from June to November 2021) in an intership at BedRock Systems (Germany), a company that develops verified software systems.

9 Dissemination

Participants: Clément Blaudeau, Frédéric Bour, Basile Clement, Nathanaëlle Courant, Paulo Emílio de Vilhena, Quentin Ladeveze, Xavier Leroy, Jean-Marie Madiot, Luc Maranget, Alexandre Moine, Glen Mével, François Pottier, Thomas Refis, Didier Rémy, Léo Stefanesco.

9.1 Promoting scientific activities

9.1.1 Chairs and members of program committees

Chairs of conference program committees Frédéric Bour was the program chair of the 2021 OCaml Workshop, which brings together industrial users of OCaml with academics and hackers who are working on extending the language, type system, and tools. Jean-Marie Madiot was one of the two co-chairs of the 2021 Coq Workshop, the 12th international workshop for users, contributors and developers of the Coq proof assistant.

Members of conference program committees Frédéric Bour was a member of the program committees of the 2021 ML Workshop and of the conference JFLA 2022. Xavier Leroy was a member of the program commitee of CoqPL 2022, the 8th international workshop on Coq for Programming Languages. François Pottier was a member of the program committee for the conference POPL 2022.

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 boards of the Journal of Functional Programming and the Proceedings of the ACM on Programming Languages.

9.1.3 Research administration

Luc Maranget is a member of Inria's *Commission d'évaluation*. This year, he co-authored a report on software evaluation [27]. 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.

9.2 Teaching – Supervision – Juries

9.2.1 Teaching

In 2021, the members of our team have taught or assisted in teaching the following courses:

- Open lectures: Xavier Leroy, *Logiques de programmes : quand la machine raisonne sur ses logiciels*, 16 HETD, Collège de France, France.
- Master (M2): *Proofs of Programs*, Jean-Marie Madiot, 18 HETD, MPRI, Université de Paris, France.
- Master (M2): *Programming shared-memory multicore machines*, Luc Maranget, 18 HETD, MPRI, Université de Paris, France. Luc Maranget is in charge of this course.
- Master (M2): *Functional programming and type systems*, François Pottier, 15 HETD, MPRI, Université de Paris, France.
- Master (M2): *Functional programming and type systems*, Didier Rémy, 15 HETD, MPRI, Université de Paris, France. Didier Rémy is in charge of this course. Didier Rémy is also Inria's delegate in the pedagogical team and management board of MPRI.

- Licence (L3): Jean-Marie Madiot, *Introduction à l'informatique*, 40 HETD, École Polytechnique, France.
- Licence (L3): Jean-Marie Madiot, *Programmation fonctionnelle*, 24 HETD, Université de Paris, France.
- Licence (L3): Nathanaëlle Courant, *Langages de programmation et compilation*, 40 HETD, ENS Paris, France.
- Licence (L1): Glen Mével, *Concepts informatiques*, 32 HETD, Université de Paris, France.

9.2.2 Supervision

The following PhD theses are in progress or have been defended in 2021:

- PhD in progress: Clément Blaudeau, *Formalizing and improving OCaml modules*, since October 2021, advised by Didier Rémy.
- PhD (CIFRE) in progress: Frédéric Bour, *An interactive, modular proof environment for OCaml*, Université de Paris, since August 2020, advised by François Pottier and Thomas Gazagnaire (Tarides).
- PhD in progress: Basile Clement, *On the design and verification of tensor compilers*, École Normale Supérieure, since September 2018, advised by Xavier Leroy since October 2019.
- PhD in progress: Nathanaëlle Courant, *Towards an efficient, formally-verified proof checker for Coq*, Université de Paris, since September 2019, advised by Xavier Leroy.
- PhD in progress: Paulo Emílio de Vilhena, *Proof of programs with effect handlers*, Université de Paris, since September 2019, advised by François Pottier.
- PhD in progress: Quentin Ladeveze, *Generic conditions for DRF-SC in axiomatic memory models*, Université de Paris, since October 2019, advised by Luc Maranget and Jean-Marie Madiot. Resigned in January 2021.
- PhD in progress: Glen Mével, *Towards a system for proving the correctness of concurrent Multicore OCaml programs*, Université de Paris, since November 2018, advised by Jacques-Henri Jourdan and François Pottier.
- PhD in progress: Alexandre Moine, *Formal verification of space bounds*, Université de Paris, since October 2021, advised by Arthur Charguéraud and François Pottier.
- PhD (CIFRE) in progress: Thomas Refis, *Designing and formalizing modular implicits*, Université de Paris, since February 2021, advised by Didier Rémy and Thomas Gazagnaire (Tarides).
- PhD: Léo Stefanesco, *Asynchronous and Relational Soundness Theorems for Concurrent Separation Logic*, Université de Paris, advised by Paul-André Mélliès, defended on November 12, 2021.

9.2.3 Juries

Xavier Leroy acted as a reviewer (*rapporteur*) and jury member for the PhD defense of Cyril Six (Université Grenoble Alpes, July 2021). He was a member of the jury for the Habilitation defense of Sylvain Boulmé (Université Grenoble Alpes, September 2021). He chaired the jury for the PhD defense of Denis Merigoux (Université PSL, December 2021). Luc Maranget was a member of a recruiting jury for *Chargés de Recherche*. François Pottier chaired the juries of the PhD defenses of Victor Lanvin (Université de Paris; defended November 9, 2021) and Léo Stefanesco (Université de Paris; defended November 12, 2021).

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] 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.

[5] 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.

[6] P. Emílio De Vilhena and F. Pottier. 'A Separation Logic for Effect Handlers'. In: *Proceedings of the ACM on Programming Languages* (Jan. 2021). DOI: 10.1145/3434314. URL: https://hal.inria.fr/hal-03049514.

[7] 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.

[8] 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.

[9] 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.

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

10.2 Publications of the year

International journals

[13] 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.

[14] 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.

[15] P. Emílio De Vilhena and F. Pottier. 'A Separation Logic for Effect Handlers'. In: *Proceedings of the ACM on Programming Languages* 5.POPL (17th Jan. 2021). DOI: 10.1145/3434314. URL: https://hal.inria.fr/hal-03049514.

[16] 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.

[17] 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.

International peer-reviewed conferences

[18] F. Bour and F. Pottier. 'Faster reachability analysis for LR(1) parsers'. In: SLE 2021 - ACM SIGPLAN International Conference on Software Language Engineering. Chicago, United States, Oct. 2021. DOI: 10.1145/3486608.3486903. URL: https://hal.inria.fr/hal-03478172.

[19] G. Mével and J.-H. Jourdan. 'Formal Verification of a Concurrent Bounded Queue in a Weak Memory Model'. In: ICFP 2021 - 26th ACM SIGPLAN International Conference on Functional Programming. Vol. 5. ICFP. Virtual, Japan, 22nd Aug. 2021. DOI: 10.1145/3473571. URL: https://hal.archives-ouvertes.fr/hal-03298759.

[20] 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.

[21] 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.

National peer-reviewed Conferences

[22] F. Bour, B. Clément and G. Scherer. 'Tail Modulo Cons'. In: JFLA 2021 - Journées Francophones des Langages Applicatifs. Saint Médard d'Excideuil, France, 6th Apr. 2021. URL: https://hal.inria.fr/hal-03146495.

[23] Q. Ladeveze. 'Mécanisation du modèle RC11 et de la propriété DRF-SC'. In: JFLA 2021 - 32es Journées Francophones des Langages Applicatifs. Saint Médard d'Excideuil, France, 6th Apr. 2021. URL: https://hal.inria.fr/hal-03081928.

[24] F. Pottier. 'Strong Automated Testing of OCaml Libraries'. In: JFLA 2021 - 32es Journées Francophones des Langages Applicatifs. Saint Médard d'Excideuil, France, 3rd Feb. 2021. URL: https://hal.inria.fr/hal-03049511.

Reports & preprints

[25] A. W. Appel and X. Leroy. *Efficient Extensional Binary Tries*. 10th Oct. 2021. URL: https://hal.inria.fr/hal-03372247.

[26] A. Canteaut, M. A. Fernández, L. Maranget, S. Perin, M. Ricchiuto, M. Serrano and E. Thomé. *Évaluation des Logiciels*. Inria, 14th Jan. 2021. URL: https://hal.inria.fr/hal-03110723.

[27] A. Canteaut, M. A. Fernández, L. Maranget, S. Perin, M. Ricchiuto, M. Serrano and E. Thomé. *Software Evaluation*. Inria, 14th Jan. 2021. URL: https://hal.inria.fr/hal-03110728.

[28] X. Leroy. *The CompCert C verified compiler: Documentation and user's manual.* Inria, 19th Nov. 2021, pp. 1–79. URL: https://hal.inria.fr/hal-01091802.

[29] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy and J. Vouillon. *The OCaml system release 4.13: Documentation and user's manual.* Inria, 24th Sept. 2021, pp. 1–876. URL: https://hal.inria.fr/hal-00930213.

10.3 Other

Softwares

[30] [SW] A. Moine, A. Charguéraud and F. Pottier, *Specification and Verification of a Transient Stack (Artifact)*, 9th Dec. 2021. LIC: MIT License. HAL: ⟨hal-03473197⟩, URL: https://hal.inria.fr/hal-03473197, VCS: https://gitlab.inria.fr/amoine/cfml-sek, SWHID: ⟨swh:1:dir:80952d7602e8e9614979f1b1f50fdad8a4c369f1;origin=https://hal.archives-ouvertes.fr/hal-03473197;visit=swh:1:snp:781fc4f58efbdfa6ea16006272793bf7c610d760;anchor=swh:1:rel:3df60f40567bfdf2b365cac2fe61c6bb38d94503;path=/⟩.

10.4 Cited publications

[31] A. W. Appel. *Program Logics for Certified Compilers.* Cambridge University Press, 2014. URL: http://www.cambridge.org/de/academic/subjects/computer-science/programming-languages-and-applied-logic/program-logics-certified-compilers?format=HB.

[32] 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.

[33] 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.

[34] G. Mével, J.-H. Jourdan and F. Pottier. 'Time credits and time receipts in Iris'. In: *European Symposium on Programming (ESOP).* Vol. 11423. Lecture Notes in Computer Science. Springer, Apr. 2019, pp. 1–27. URL: http://cambium.inria.fr/~fpottier/publis/mevel-jourdan-pottier-time-in-iris-2019.pdf.

[35] C. Okasaki. *Purely Functional Data Structures.* Cambridge University Press, 1999. URL: http://www.cambridge.org/us/catalogue/catalogue.asp?isbn=0521663504.

[36] A. Oliveira Vale, P.-A. Melliès, Z. Shao, J. Koenig and L. Stefanesco. 'Layered and Object-Based Game Semantics *'. In: *Proceedings of the ACM on Programming Languages* (Jan. 2022). DOI: 10.1145/3498703. URL: https://hal.inria.fr/hal-03456034.

[37] F. Pottier. 'Reachability and error diagnosis in LR(1) parsers'. In: *Compiler Construction (CC).* Mar. 2016, pp. 88–98. URL: http://gallium.inria.fr/~fpottier/publis/fpottier-reachability-cc2016.pdf.

[38] 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).

[39] K. Thompson. 'Reflections on trusting trust'. In: *ACM Turing award lectures.* 1983. URL: https://dl.acm.org/doi/pdf/10.1145/358198.358210.

[40] L. White, F. Bour and J. Yallop. 'Modular implicits'. In: *Electronic Proceedings in Theoretical Computer Science* 198 (Dec. 2015), pp. 22–63. URL: http://dx.doi.org/10.4204/EPTCS.198.2.