RESEARCH CENTRE

Paris

IN PARTNERSHIP WITH:

Collège de France

2020 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]
- Ioannis Filippidis [Inria, Starting Research Position, until Mar 2020]
- Xavier Leroy [Collège de France, Senior Researcher]
- Jean-Marie Madiot [Inria, Researcher]
- Luc Maranget [Inria, Researcher]
- Gabriel Radanne [Inria, Starting Research Position, until Nov 2020]
- Didier Rémy [Inria, Senior Researcher, HDR]

PhD Students

- Frédéric Bour [Tarides, CIFRE]
- Basile Clement [Inria]
- Nathanaël Courant [École Normale Supérieure de Paris]
- Paulo Emílio De Vilhena [Inria]
- Quentin Ladeveze [Inria]
- Glen Mével [Inria]
- Thomas Refis [Tarides, CIFRE, from Feb 2020]
- Léo Stefanesco [Collège de France, from Oct 2020]

Technical Staff

- Florian Angeletti [Inria, Engineer]
- Sébastien Hinderer [Inria, Engineer, from Mar 2020]

Interns and Apprentices

- Antoine Hacquard [Inria, until Jan 2020]

Administrative Assistant

- Hélène Milome [Inria]

Visiting Scientist

- Jacques Garrigue [Université de Nagoya, until Jul 2020]

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 [2], are examples.

2.3 Design, implementation, and evolution of OCaml

Our group's expertise in programming language design, formalization and implementation has traditionally been focused mainly on the programming language OCaml [27]. OCaml can be described as a high-level statically-typed general-purpose programming language. Its main features include first-class functions, algebraic data structures and pattern matching, automatic memory management, support for traditional imperative programming (mutable state, exceptions), and support for modularity and encapsulation (abstract types; modules and functors; objects and classes).

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

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

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

2.4 Software verification

We have already mentioned the importance of formal verification to achieve the highest levels of software quality. One of our major contributions to this field has been the verification of programming tools, namely the CompCert optimizing compiler for the C language [34] and the Verasco abstract interpretation-based static analyzer [7]. 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 Software

5.1.1 The CompCert 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 [34]. It comprises a back-end, which translates the Cminor intermediate language to PowerPC assembly and which can be reused for source languages other than C [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 CompCert C compiler and tools in several directions:

- Conformance with the ISO C 11 standard was improved: the `_Static_assert` construct is now supported, and several discrepancies were fixed.

- We introduced mechanisms to specify the semantics of built-in functions. We used them to provide semantics for a growing number of built-in functions, making them amenable to optimizations such as constant propagation and common subexpression elimination.
- The x86 code generator was updated to support Cygwin 64 as a target platform, making it easier to use CompCert under Windows.
- Conformance with the Application Binary Interfaces (ABI) for AArch64, PowerPC, RISC-V, and x86 was improved.
- The `clightgen` tool, which integrates CompCert in the Verified Software Toolchain developed by Andrew Appel's team at Princeton University [28], was improved so as to simplify the verification of C programs composed of multiple source files.

We released two versions of CompCert incorporating these improvements: version 3.7 in March 2020 and version 3.8 in November 2020.

5.1.2 The OCaml system

Participants Florian Angeletti, Frédéric Bour, Damien Doligez, Jacques Garrigue *(Nagoya University)*, Sébastien Hinderer, Xavier Leroy, Luc Maranget, Thomas Refis, David Allsop *(Cambridge University)*, Stephen Dolan *(Cambridge University)*, Alain Frisch *(Lexifi)*, Jacques-Henri Jourdan *(CNRS)*, Nicolás Ojeda Bär *(Lexifi)*, Gabriel Scherer *(Inria team Partout)*, Mark Shinwell *(Jane Street)*, Leo White *(Jane Street)*, Jeremy Yallop *(Cambridge University)*.

This year, we released two major versions of the OCaml system: version 4.10.0 in February and version 4.11.0 in August. We also released four minor versions (4.09.1, 4.10.1, 4.10.2, 4.11.1) with bug fixes and backports of new features. The main novelties in these releases are:

- The language has been extended to allow users to define their own multidimensional array-like indexing operators.
- A new memory profiler, `statmemprof`, is now available. It is based on statistical sampling of allocations, resulting in much lower run-time overhead than the earlier memory profiler, Spacetime.
- OCaml now fully supports the MacOS system running on ARM 64-bit processors, including the latest Macintosh models with "Apple silicon".
- A native-code generator for the RISC-V processor architecture has been added.
- A new memory allocator for the major heap is now available. It is based on a best-fit strategy and causes less heap fragmentation than the previous allocator.
- The exhaustiveness check has been made more precise for case analyses that involve GADTs or empty types.
- The runtime system can now record statistics and events in the standard CTF format.
- The integration of the Multicore OCaml project has begun, with a number of internal changes to the runtime system that will be required for full multicore support.
- More than one hundred usability improvements have been implemented, ranging from more readable stack backtraces through improved error messages to new standard library functions.
- About 80 bugs have been fixed.

5.1.3 The Menhir parser generator

Participants François Pottier.

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, a number of improvements and bug fixes have been performed:

- The algorithms for constructing LR automata have been re-implemented in a style that is more concise, more elegant and more efficient. A bug in our implementation of Pager's construction algorithm has been fixed. Our new implementation operates in two phases: first compute the automaton's states, then compute its transitions. It is significantly easier to describe and understand than our earlier implementation.
- A simplified error-handling strategy has been introduced. It removes a couple undesirable features of the traditional error-handling strategy inherited from `yacc`.
- Several "bit set" data structures have been improved, for greater speed.
- The test that determines whether macro-expansion will terminate has been re-implemented. The old test was discovered to be neither sound nor complete. The new test is hopefully both sound and complete.
- The manner in which Menhir performs type inference has been slightly modified so as to avoid certain (infrequent) situations in which incorrect types could be inferred.
- Menhir is now built using `dune` instead of `ocamlbuild`. This allows taking advantage of parallelism while building and testing Menhir.

5.1.4 The `odoc` documentation tool

Participants Jon Ludlam *(University of Cambridge)*, Gabriel Radanne, Florian Angeletti, Leo White *(Jane Street)*.

Rendering the documentation of a piece of OCaml code is a difficult task. Indeed, the OCaml module system allows setting up complex inter-dependencies that are difficult to compute and difficult to render in a concise document. `odoc` is our latest attempt at creating a documentation tool that handles the full complexity of the OCaml language.

This year, Gabriel Radanne rewrote a significant portion of `odoc` to provide improved HTML output, make it possible to produce other document formats, and introduce the ability to produce `man` pages. Florian Angeletti then implemented PDF output and integrated the usage of `odoc` in the official OCaml distribution. Concurrently, Jon Ludlam and Leo White rewrote the resolution mechanism of `odoc`, which led to a joint presentation at the OCaml workshop.

5.1.5 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, a feature branch for handling virtual memory was created, of which Jade Alglave and Luc Maranget are (almost) the exclusive authors. It is the backbone of our current research. Moreover, many authors (most of whom work at ARM Ltd.) have been contributing in various manners:

- Test and build infrastructure.
- Extension of the simulated instruction set.
- Architectural extensions: capabilities (ARM Morello), SIMD (ARM Neon; work in progress).

5.1.6 The TLAPS proof system

Participants Damien Doligez, Leslie Lamport *(Microsoft Research)*, Ioannis Filippidis, Stephan Merz *(Inria team VeriDis)*.

Damien Doligez heads 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 [31], and to build tools for writing TLA+ specifications and mechanically checking the proofs.

This year, we made a bug-fix release of TLAPS (version 1.4.5). We have also been testing the new support for the ENABLED operator and the action composition operator in TLA+ proofs. We hope to release version 1.5.0 with this support very soon.

We also made a maintenance release of Zenon (version 0.8.5) to make it compatible with the latest version of OCaml.

5.1.7 Sek, an efficient sequence library for OCaml

Participants Arthur Charguéraud, François Pottier.

This year, Arthur Charguéraud and François Pottier developed Sek, a library that offers efficient implementations of ephemeral sequences, persistent sequences, and ephemeral iterators on both kinds of sequences. Sek publishes 4 abstract types and over 150 operations. Its data structures involve complex balancing invariants, shared mutable state, and a subtle ownership policy that determines when an object can be updated in place and when it must be copied. It is intended to offer very good performance, in terms of both space and time, in most usage scenarios; thus, it should remove the need for more specialized data structures, such as stacks, queues, deques, catenable deques, and so on. The library has been published and is available via OCaml's package manager, `opam`. No paper about Sek has been published at this time. It is worth noting that testing Sek was our main motivation for developing Monolith (§5.1.8).

5.1.8 Monolith, a library for testing OCaml libraries

Participants François Pottier.

This year, François Pottier developed Monolith, an OCaml library whose purpose is to facilitate black-box testing of other OCaml libraries. Monolith provides a rich specification language, which allows the user to describe her library's API, and an engine, which generates clients of this API and executes them. This reduces the problem of testing a library to the simpler problem of testing a complete program. Testing can then be performed either in a purely random manner or with the help of an off-the-shelf fuzzer, such as AFL. This work is described in a paper that will be presented at JFLA 2021 [20]. Monolith has been used to test and debug Sek (§5.1.7).

5.2 New software

5.2.1 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

Contacts: Xavier Leroy, Damien Doligez

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

5.2.2 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/

Authors: Xavier Leroy, Bernhard Schommer, Guillaume Melquiond, Jacques-Henri Jourdan, Sylvie Boldo

Contact: Xavier Leroy

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

Partner: AbsInt Angewandte Informatik GmbH

5.2.3 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/

Authors: Jade Alglave, Luc Maranget

Contact: Luc Maranget

Participants: Jade Alglave, Luc Maranget

Partner: University College London UK

5.2.4 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: François Pottier

5.2.5 CFML

Name: Interactive program verification using characteristic formulae

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

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

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

Contact: Arthur Charguéraud

Participants: Arthur Charguéraud, Armaël Guéneau, François Pottier

5.2.6 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 back-end. 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

Contacts: Stephan Merz, Damien Doligez

Participants: Damien Doligez, Stephan Merz, Ioannis Filippidis

Partner: Microsoft

5.2.7 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

Author: Damien Doligez

Contact: Damien Doligez

Participant: Damien Doligez

5.2.8 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 Programming language design and implementation

6.1.1 Evolution of the OCaml type system

Participants Florian Angeletti, Jacques Garrigue *(Nagoya University)*, Thomas Refis, Didier Rémy, Leo White *(Jane Street)*, Gabriel Scherer *(Inria team Partout)*.

Throughout this year, taking advantage of Jacques Garrigue's visit, we have worked to improve the type system, its robustness, and its implementation. This includes:

- Giving a proper formalization of the typing of generalized algebraic data types (GADTs) and pattern matching on GADTs, including making progress towards their formalization in Coq.
- Introducing notions of injective and nominal types, which allow a more complete typing of GADTs. These features have been presented by Jacques Garrigue at the 2020 ML Family Workshop.[2] Injective types will be part of OCaml 4.12.
- Adding support for naming existential type variables in pattern matching constructs.
- Adding support for GADTs inside disjunctive patterns.
- Improving the readability of the type-checker's code.

6.1.2 Refactoring with ornaments in ML

Participants Didier Rémy, Ambre Williams *(Google Paris)*.

[2] http://www.mlworkshop.org/workshops/ml2020/ml2020-injectivity.pdf

Ambre Williams and Didier Rémy have been working on ornaments for ML, a technique that allows code refactoring and evolution based on the transformations of datatypes. 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 improved and completed the formalization of ornaments, which she presented in her PhD dissertation in December [22].

6.1.3 Linear types

Participants Gabriel Radanne, Hannes Saffrich *(University of Freiburg)*, Peter Thiemann *(University of Freiburg)*.

Linear types, recently made popular by the Rust programming language, allow to statically check the usage of resources such as file descriptors, network connections, or dynamically allocated memory.

In 2019, Peter Thiemann, Hannes Saffrich and Gabriel Radanne developed the language Affe, which combines Rust's flagship features, namely linear types and ownership, with the ease of use of the programming languages of the ML family, thanks to support for functional programming, GC-by-default, and full type inference.

This year, they wrote a paper that describes Affe and its inner workings and demonstrates its soundness [14]. This paper was presented at ICFP 2020 by Gabriel Radanne.

6.1.4 An incremental type-checker for OCaml modules

Participants Gabriel Radanne, Didier Rémy, Jacques Garrigue *(Nagoya University)*, Thomas Refis.

Modules are a core feature of ML languages, allowing to assemble pieces of software in a high-level and composable fashion. OCaml benefits from a particularly rich module system which was originally described more than two decades ago [35, 33], but has significantly grown since.

This year, Gabriel Radanne, in collaboration with Didier Rémy and Jacques Garrigue, started formalizing a new module system which combines all of the features that have been introduced since the last formalization effort by Xavier Leroy. This new system also improves inference and provides a solid basis for further experiments, such as the "modular implicits" that are currently being investigated by Thomas Refis and Didier Rémy (§6.1.5). Gabriel Radanne started a "clean room" implementation of a prototype type-checker for this new module system.

6.1.5 Designing and formalizing modular implicits

Participants Thomas Refis, Didier Rémy, Gabriel Radanne, 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 [37]. 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.

In 2020, Thomas Refis started a PhD under the supervision of Didier Rémy. He aims to revive this project and eventually to merge it into mainline OCaml.

This year, Thomas and Didier worked on formalizing the semantics of modular explicits, which can be seen as syntactic sugar for first-class functors. They also started studying the resolution mechanism implemented in the earlier prototype of modular implicits, so as to understand its limitations and its interaction with the OCaml type-checker.

6.1.6 Partial type inference with second-order types

Participants Didier Rémy.

Adding second-order types to ML in a way that smoothly integrates with ML-style type inference for first-order types has been a challenge for many years. In 2007, Le Botlan and Rémy proposed a solution to this problem, named MLF, that is still the state of the art today. Unfortunately, this solution has not been adopted, as it is a bit involved and goes beyond System F, which is the reference system for second-order types. As a result, partial type inference in the presence of second-order types remains a topic of research, and researchers continue to propose ad hoc solutions.

Didier Rémy has been investigating a new approach using the powerful inference engine of MLF to infer a typing derivation in MLF, which is then lowered to a derivation of System F. This process fails if no System F derivation exists. Preliminary investigations are promising.

6.1.7 Automatic synthesis of high-performance numerical libraries

Participants Basile Clement.

Basile Clement has been working on developing a domain-specific language and a machine-learning-based compiler for the synthesis of high-performance numerical libraries, with a focus on GPUs.

One of the key challenges is to design a language that is expressive enough to capture complex loop-based code transformations while keeping the proof of semantic preservation simple. Another challenge is to accurately capture the potential peculiarities of the hardware, so as to expose the relation between the program and its performance, as far as possible, to the learning component.

This year, Basile worked on a formal semantics for an array language with explicit loops and an equational semantics. He wrote a small prototype verifier in OCaml to test the equivalence of a generated implementation with a given specification, as well as a specification in Coq for a subset of the language.

6.1.8 Analysis of an LR parser's stack

Participants Frédéric Bour.

A few years ago, building on prior work by Jeffery, François Pottier implemented in the Menhir parser generator a principled method for producing good syntax error messages [36]. This method chooses an error message based solely on the state in which a syntax error was detected by the LR automaton.

This year, Frédéric Bour investigated a more ambitious approach, with the aim of providing messages that are better suited to certain specific situations. This approach is able to exploit not only the current state of the LR automaton, but also to analyze the shape of its stack, thanks to a novel form of regular expressions.

6.1.9 Formalization of equational monadic reasoning for combined choice

Participants Jacques Garrigue *(Nagoya University)*, Reynald Affeldt *(AIST)*, David Nowak *(AIST)*, Takafumi Saikawa *(Nagoya University)*.

Following work by Affeldt, Nowak and Saikawa on formalized equational monadic reasoning for programs with a variety of effects, we tackled the problem of providing a concrete model for a monad that combines nondeterministic choice with probabilistic choice. The resulting monad is fully formalized in Coq, and relies on a formalization of discrete probabilities and convex distributions. A report has been submitted for publication. This also required a formalization of axiomatic convex spaces, which was separately published at CICM 2020.[3]

6.2 Shared-memory concurrency

6.2.1 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 a decade. This year, they submitted a paper entitled "Armed cats: formal concurrency modeling at Arm" to the journal *Transaction on Programming Languages and Systems* (TOPLAS). The article has been accepted modulo revisions, and a revised version has been submitted in December. 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 two provably equivalent alternative formulations. Our results have been confirmed via vast experimental campaigns, synthesized at `http://diy.inria.fr/mixed/`. 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. We already have interesting experimental results, having experimentally demonstrated the necessity of systems programming idioms recommended by ARM's official documentation.

6.2.2 Unifying axiomatic and operational weak memory models

Participants Quentin Ladeveze, Jean-Marie Madiot, Jade Alglave *(ARM Ltd & University College London)*, Simon Castellan *(Inria team Celtique)*.

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

[3] `https://link.springer.com/chapter/10.1007%2F978-3-030-53518-6_2`

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 formalization 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-synchronized programs can assume a strong memory model.

This year, Jean-Marie Madiot also started developing a tool that transforms models written in the format of the memory model simulator `herd` into models that can be reasoned about in the Coq proof assistant. This allowed him to build mechanized proofs of properties of existing models: inclusion of models, equivalence of models, and equivalence of different acyclicity conditions. Quentin Ladeveze's formalization of the DRF-SC property of C/C++11 programs (§6.2.3) fits inside the same framework.

6.2.3 A mechanized proof of DRF-SC for the RC11 memory model

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 proof of the DRF-SC property that is independent of the memory model.

DRF-SC is a well-known property that most memory models are intended to satisfy. In a weak memory model that satisfies this property, if all of the executions of a program under the sequentially consistent (SC) model are data-race-free (DRF), then all executions of the program under the weak memory model are also consistent with SC. In practice, this property allows a programmer to completely ignore the possible weak behaviors of her program, provided she writes a data-race-free program. Proofs of this property exist for some weak memory models. Our goal is to design a generic proof, by relying on properties that are shared by all of the models that satisfy this property.

This year, Quentin Ladeveze formalized in the Coq proof assistant a memory model that describes the behavior of C/C++11 programs and a proof that this model enjoys the DRF-SC property.

A paper that describes this formalization has been accepted for presentation at JFLA 2021 [19].

6.3 Software specification and verification

6.3.1 Verified code generation in the polyhedral model

Participants Nathanaël Courant, Xavier Leroy.

The polyhedral model is a high-level intermediate representation for loop nests iterating over arrays and matrices, as found in numerical code. It supports a great many loop optimizations (fusion, splitting, interchange, blocking, etc) in a uniform, mathematically-elegant manner.

In 2018, Nathanaël Courant, under Xavier Leroy's supervision, developed a Coq formalization of the polyhedral model. He implemented and verified a code generator that produces efficient sequential code out of an optimized polyhedral representation.

This year, Nathanaël Courant and Xavier Leroy wrote a paper that describes this verification work [11]. This paper has been published and presented at the conference POPL 2021.

6.3.2 A separation logic for effect handlers

Participants Paulo Emílio de Vilhena, François Pottier.

Paulo Emílio de Vilhena is a second-year PhD student, advised by François Pottier. His aim is to devise a separation logic with support for verifying programs that exploit effect handlers.

This year, Paulo made considerable progress towards that goal. He defined a calculus with support for effect handlers, formalized its operational semantics in Coq, and proposed a new separation logic, based on Iris [30], for this calculus. To demonstrate the usability of this logic, he carried out two short yet nontrivial case studies. A paper on this topic has been published and presented at the conference POPL 2021 [5].

6.3.3 A program logic for Multicore Ocaml

Participants Glen Mével, Jacques-Henri Jourdan *(CNRS)*, François Pottier.

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.

At the beginning of this year, we proposed a program logic for Multicore OCaml, named Cosmo. Cosmo extends Iris [30] with a notion of "view" that allows the programmer to reason in an explicit (yet abstract) manner about the current thread's view of memory, and about the manner in which views are transferred from one thread to another via reads and writes of atomic memory locations. This work has been published and presented by Glen at ICFP 2020 [13].

Since then, Glen extended Cosmo with support for arrays. This extension allows reasoning on a wider variety of programs. Using this extended logic, Glen proved the functional correctness of a concurrent first-in first-out queue, whose invariant is nontrivial.

6.3.4 Algebraically closed fields in Isabelle/HOL

Participants Paulo Emílio de Vilhena, Lawrence Paulson *(University of Cambridge)*.

During the summer of 2018, Paulo Emílio de Vilhena did a research internship under the supervision of Lawrence Paulson at the University of Cambridge. The topic was the formalization of mathematics in the proof assistant Isabelle/HOL. Paulo and Lawrence verified in Isabelle/HOL the fundamental theorem of algebra, which states that every field has an algebraic closure.

Although the majority of this work was complete by the end of the internship, the results were published only this year at the conference IJCAR 2020 [17].

6.3.5 Towards an efficient, verified proof checker for Coq

Participants Nathanaël Courant.

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

One of the key challenges is to perform strong reduction, whereas computers are better suited to weak reduction. Strong reduction is a form of computation that involves both concrete values and symbolic variables, and where it is permitted to perform reductions in the body of a function before this function is actually called. Weak reduction, in contrast, involves concrete values only, and does not evaluate a function until it is invoked. With strong reduction, care must be taken not to evaluate too much, and to avoid reducing the body of functions that will not appear in the final term. Besides, another difficult problem is to compare reduced terms, or even to combine the convertibility test with reduction itself.

This year, Nathanaël worked on defining a big-step semantics for strong call-by-need λ-calculus. He also wrote a Coq proof that such a semantics is compatible with standard small-step reduction for the λ-calculus, and extended this result to show that it still holds in the presence of data constructors and pattern matching.

6.3.6 An interactive, modular proof environment for OCaml

Participants Frédéric Bour, François Pottier.

Frédéric Bour aims to design and implement a tool that allows verifying simple properties of OCaml programs. As a starting point, he is adapting the methodology of the VeriFast verifier [29] to OCaml programs.

This year, Frédéric worked on the translation of a subset of OCaml to a custom verification language. Verification is then handled by two different layers: an ad-hoc verifier based on separation logic for modeling effects and the Z3 theorem prover for first-order properties.

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 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, 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).

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 International research visitors

8.1.1 Visits of international scientists

Jacques Garrigue (Nagoya University) visited us in Paris from September 2019 to June 2020. He has long been one of the key designers and implementors of the OCaml type system. During his visit, we have collaborated 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: organisation

François Pottier is a member of the ICFP steering committee.

9.1.2 Scientific events: selection

Xavier Leroy was a member of the program committee of CC 2021, the 30th International Conference on Compiler Construction, and of PriSC 2021, the Workshop on Principles of Secure Compilation.

Luc Maranget was a member of the program committee of POPL 2021, the 48th ACM SIGPLAN Symposium on Principles of Programming Languages.

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

9.1.3 Journals

Xavier Leroy is an area editor for the Journal of the ACM, in charge of the Programming Languages area. He is also a member of the editorial board of Journal of Automated Reasoning.

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

9.1.4 Research administration

Damien Doligez chairs the Caml Consortium.

Luc Maranget is member of Inria *Commission d'évaluation*. He was a member of two *Chargé de Recherche* hiring committees.

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

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

9.2 Teaching - Supervision - Juries

9.2.1 Teaching

In 2020, the members of our team have taught or assisted in teaching the following courses:

- Open lectures: Xavier Leroy, *Sémantiques mécanisées: quand la machine raisonne sur ses langages*, 19 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, 22 HETD, MPRI, Université de Paris, France.
- Master (M2): “Functional programming and type systems”, Didier Rémy, 24 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.
- Master (M1): “Compilation and Analysis of Programs”, Gabriel Radanne, 28 HETD, Master IF, ENS Lyon, France.
- Licence (L3): Jean-Marie Madiot, “Introduction à l’informatique”, 40 HETD, École Polytechnique, France.
- Licence (L3): Basile Clement, “Langages de programmation et compilation” (course taught by Jean-Christophe Filliâtre), 64 HETD, École Normale Supérieure, France.
- Licence (L3): Nathanaël Courant, “Mécanismes de la programmation orientée objet”, 40 HETD, École Polytechnique, France.
- Licence (L3): Nathanaël Courant, “Préparation aux concours SWERC”, 45 HETD, École Polytechnique, France.
- Licence (L2): Glen Mével, “Projet informatique 4”, 32 HETD, Université de Paris, France.
- Licence (L1): Glen Mével, “Initiation à la programmation 1 en Python”, 32 HETD, Université de Paris, France.

9.2.2 Supervision

The following PhD theses are in progress or have been defended in 2020:

- PhD (CIFRE) in progress: Frédéric Bour, “An interactive, modular proof environment for OCaml”, Université de Paris, since October 2019 (approved by ANRT in August 2020), advised by François Pottier and Thomas Gazagnaire (Tarides).
- PhD in progress: Basile Clement, “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 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.
- 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 (CIFRE) in progress: Thomas Refis, “Modular Implicits: Design, Formalization, and Implementation”, Université de Paris, since February 2020, advised by Didier Rémy and Thomas Gazagnaire (Tarides).
- PhD: Ambre Williams, “Refactoring functional programs with ornaments”, Université de Paris, defended on December 14, 2020 [22], advised by Didier Rémy.

9.2.3 Juries

Xavier Leroy chaired the jury for the PhD defense of Lélio Brun (Université PSL, July 2020). He was a member of the jury for the PhD defenses of Raphaël Rieu-Helft (Université Paris-Saclay, November 2020) and Darius Mercadier (Sorbonne Université, November 2020).

François Pottier was a reviewer for the PhD theses of Lionel Parreaux (EPFL, June 2020) and Ralf Jung (MPI-SWS, August 2020).

9.3 Popularization

9.3.1 Interventions

Xavier Leroy gave a popularization talk on contact tracing applications at lycée Montaigne, Paris, in October 2020.

10 Scientific production

10.1 Major publications

[1] 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.

[2] 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.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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] N. Pouillard and F. Pottier. 'A unified treatment of syntax with binders'. In: *Journal of Functional Programming* 22.4–5 (2012), pp. 614–704. URL: http://dx.doi.org/10.1017/S0956796812000251.

10.2 Publications of the year

International journals

[11] 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.

[12] 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.

[13] G. Mével, J.-H. Jourdan and F. Pottier. 'Cosmo: A Concurrent Separation Logic for Multicore OCaml'. In: *Proceedings of the ACM on Programming Languages* 4.ICFP (2nd Aug. 2020). DOI: 10.1145/3408978. URL: https://hal.archives-ouvertes.fr/hal-02929998.

[14] G. Radanne, H. Saffrich and P. Thiemann. 'Kindly bent to free us'. In: *Proceedings of the ACM on Programming Languages* 4.ICFP (2nd Aug. 2020), pp. 1–29. DOI: 10.1145/3408985. URL: https://hal.archives-ouvertes.fr/hal-02938020.

[15] P. E. de Vilhena, F. Pottier and J.-H. Jourdan. 'Spy Game: Verifying a Local Generic Solver in Iris'. In: *Proceedings of the ACM on Programming Languages* 4.POPL (Jan. 2020). DOI: 10.1145/3371101. URL: https://hal.archives-ouvertes.fr/hal-02351562.

International peer-reviewed conferences

[16] B. Simner, S. Flur, C. Pulte, A. Armstrong, J. Pichon-Pharabod, L. Maranget and P. Sewell. 'ARMv8-A system semantics: instruction fetch in relaxed architectures'. In: ESOP 2020 - 29th European Symposium on Programming. Dublin, Ireland, 25th Mar. 2020. URL: https://hal.inria.fr/hal-02509910.

[17] P. E. de Vilhena and L. C. Paulson. 'Algebraically Closed Fields in Isabelle/HOL'. In: *Automated Reasoning*. IJCAR 2020 - International Joint Conference on Automated Reasoning. Vol. 12167. LNCS - Lecture Notes in Computer Science. Paris, France, 24th June 2020, pp. 204–220. DOI: 10.1007/978-3-030-51054-1_12. URL: https://hal.inria.fr/hal-03083589.

National peer-reviewed Conferences

[18] 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.

[19] 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.

[20] 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.

Scientific books

[21] X. Leroy. *Software, between mind and matter*. Inaugural lecture at Collège de France. 10th Mar. 2020. URL: https://hal.inria.fr/hal-02392159.

Doctoral dissertations and habilitation theses

[22] A. Williams. 'Refactoring functional programs with ornaments'. Université de Paris / Université Paris Diderot (Paris 7), 14th Dec. 2020. URL: https://hal.inria.fr/tel-03126602.

Reports & preprints

[23] 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.

[24] 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.

[25] S. Castellan, L. Stefanesco and N. Yoshida. *Concurrent Game Semantics: Easy as Pi*. Inria, 2nd Feb. 2020. URL: https://hal.inria.fr/hal-03128187.

[26] X. Leroy. *The CompCert C verified compiler: Documentation and user's manual*. Inria, 16th Nov. 2020, pp. 1–78. URL: https://hal.inria.fr/hal-01091802.

[27] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy and J. Vouillon. *The OCaml system release 4.11: Documentation and user's manual*. Inria, 19th Aug. 2020, pp. 1–820. URL: https://hal.inria.fr/hal-00930213.

10.3 Cited publications

[28] 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.

[29] B. Jacobs and F. Piessens. *The VeriFast Program Verifier*. Tech. rep. CW-520. Department of Computer Science, Katholieke Universiteit Leuven, Aug. 2008. URL: http://people.cs.kuleuven.be/~bart.jacobs/verifast/verifast.pdf.

[30] R. Jung, R. Krebbers, J.-H. Jourdan, A. Bizjak, L. Birkedal and D. Dreyer. 'Iris from the ground up: A modular foundation for higher-order concurrent separation logic'. In: *Journal of Functional Programming* 28 (2018), e20. URL: https://people.mpi-sws.org/~dreyer/papers/iris-ground-up/paper.pdf.

[31] L. Lamport. 'How to write a 21st century proof'. In: *Journal of Fixed Point Theory and Applications* 11 (1 2012), pp. 43–63. URL: http://dx.doi.org/10.1007/s11784-012-0071-6.

[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. 'Applicative functors and fully transparent higher-order modules'. In: *Principles of Programming Languages (POPL)*. Jan. 1995, pp. 142–153. URL: https://hal.inria.fr/hal-01499966.

[34] X. Leroy. 'Formal verification of a realistic compiler'. In: *Communications of the ACM* 52.7 (2009), pp. 107–115. URL: http://doi.acm.org/10.1145/1538788.1538814.

[35] X. Leroy. 'Manifest types, modules, and separate compilation'. In: *Principles of Programming Languages (POPL)*. Jan. 1994, pp. 109–122. URL: https://hal.inria.fr/hal-01499976.

[36] 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.

[37] 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.