Activity Report 2018

Project-Team GALLIUM

Programming languages, types, compilation and proofs

RESEARCH CENTER
Paris

THEME
Proofs and Verification

Table of contents

Project-Team GALLIUM

Creation of the Project-Team: 2006 May 01

Keywords:

Computer Science and Digital Science:

A1.1.1. - Multicore, Manycore
A1.1.3. - Memory models
A2.1. - Programming Languages
A2.1.1. - Semantics of programming languages
A2.1.3. - Object-oriented programming
A2.1.4. - Functional programming
A2.1.6. - Concurrent programming
A2.1.11. - Proof languages
A2.2. - Compilation
A2.2.1. - Static analysis
A2.2.2. - Memory models
A2.2.4. - Parallel architectures
A2.2.5. - Run-time systems
A2.4. - Formal method for verification, reliability, certification
A2.4.1. - Analysis
A2.4.3. - Proofs
A2.5.4. - Software Maintenance & Evolution
A7.1.2. - Parallel algorithms
A7.2. - Logic in Computer Science
A7.2.2. - Automated Theorem Proving
A7.2.3. - Interactive Theorem Proving

Other Research Topics and Application Domains:

B5.2.3. - Aviation
B6.1. - Software industry
B6.6. - Embedded systems
B9.5.1. - Computer science

1. Team, Visitors, External Collaborators

Research Scientists

Xavier Leroy [Team leader, Inria, Senior Researcher, until Oct 2018; Collège de France, Prof, since Nov 2018]
Umut Acar [Inria, Advanced Research Position, until Apr 2018]
Damien Doligez [Inria, Researcher]
Ioannis Filippidis [Inria, Starting Research Position, since Oct 2018]
Fabrice Le Fessant [Inria, Researcher, until Jan 2018]
Jean-Marie Madiot [Inria, Researcher]
Luc Maranget [Inria, Researcher]
Michel Mauny [Inria, Senior Researcher]

François Pottier [Inria, Senior Researcher, HDR]
Mike Rainey [Inria, Starting Research Position, until Feb 2018]
Didier Rémy [Inria, Senior Researcher, HDR]

Post-Doctoral Fellow
Gergö Barany [Inria, until Aug 2018]

PhD Students
Vitaly Aksenov [Inria, until Aug 2018]
Armaël Guéneau [Université Paris Diderot]
Glen Mével [Inria, since Nov 2018]
Naomi Testard [Inria]
Thomas Williams [ENS Paris, until Aug 2018]

Technical staff
Sébastien Hinderer [Inria]

Interns
Lucas Baudin [ENS Paris, from Mar 2018 to Aug 2018]
Nathanaël Courant [Inria, from Mar 2018 to Jul 2018]
Glen Mével [ENS Cachan, from Mar 2018 to Aug 2018]
Simon Colin [École Polytechnique, from April 2018 to July 2018]
Émilie Guermeur [Inria, from May 2018 to Jul 2018]
Sébastien Michelland [Inria, from Jun 2018 to Jul 2018]

Administrative Assistant
Laurence Bourcier [Inria]

2. Overall Objectives

2.1. Research at Gallium

The research conducted in the Gallium group aims at improving the safety, reliability and security of software through advances in programming languages and formal verification of programs. Our work is centered on the design, formalization and implementation of functional programming languages, with particular emphasis on type systems and type inference, formal verification of compilers, and interactions between programming and program proof. The OCaml language and the CompCert verified C compiler embody many of our research results. Our work spans the whole spectrum from theoretical foundations and formal semantics to applications to real-world problems.

3. Research Program

3.1. Programming languages: design, formalization, implementation

Like all languages, programming languages are the media by which thoughts (software designs) are communicated (development), acted upon (program execution), and reasoned upon (validation). The choice of adequate programming languages has a tremendous impact on software quality. By “adequate”, we mean in particular the following four aspects of programming languages:

- **Safety.** The programming language must not expose error-prone low-level operations (explicit memory deallocation, unchecked array access, etc) to programmers. Further, it should provide constructs for describing data structures, inserting assertions, and expressing invariants within programs. The consistency of these declarations and assertions should be verified through compile-time verification (e.g. static type-checking) and run-time checks.

- **Expressiveness.** A programming language should manipulate as directly as possible the concepts and entities of the application domain. In particular, complex, manual encodings of domain notions into programmatic notations should be avoided as much as possible. A typical example of a language feature that increases expressiveness is pattern matching for examination of structured data (as in symbolic programming) and of semi-structured data (as in XML processing). Carried to the extreme, the search for expressiveness leads to domain-specific languages, customized for a specific application area.
- **Modularity and compositionality.** The complexity of large software systems makes it impossible to design and develop them as one, monolithic program. Software decomposition (into semi-independent components) and software composition (of existing or independently-developed components) are therefore crucial. Again, this modular approach can be applied to any programming language, given sufficient fortitude by the programmers, but is much facilitated by adequate linguistic support. In particular, reflecting notions of modularity and software components in the programming language enables compile-time checking of correctness conditions such as type correctness at component boundaries.
- **Formal semantics.** A programming language should fully and formally specify the behaviours of programs using mathematical semantics, as opposed to informal, natural-language specifications. Such a formal semantics is required in order to apply formal methods (program proof, model checking) to programs.

Our research work in language design and implementation centers on the statically-typed functional programming paradigm, which scores high on safety, expressiveness and formal semantics, complemented with full imperative features and objects for additional expressiveness, and modules and classes for compositionality. The OCaml language and system embodies many of our earlier results in this area [37]. Through collaborations, we also gained experience with several domain-specific languages based on a functional core, including distributed programming (JoCaml), XML processing (XDuce, CDuce), reactive functional programming, and hardware modeling.

3.2. Type systems

Type systems [39] are a very effective way to improve programming language reliability. By grouping the data manipulated by the program into classes called types, and ensuring that operations are never applied to types over which they are not defined (e.g. accessing an integer as if it were an array, or calling a string as if it were a function), a tremendous number of programming errors can be detected and avoided, ranging from the trivial (misspelled identifier) to the fairly subtle (violation of data structure invariants). These restrictions are also very effective at thwarting basic attacks on security vulnerabilities such as buffer overflows.

The enforcement of such typing restrictions is called type-checking, and can be performed either dynamically (through run-time type tests) or statically (at compile-time, through static program analysis). We favor static type-checking, as it catches bugs earlier and even in rarely-executed parts of the program, but note that not all type constraints can be checked statically if static type-checking is to remain decidable (i.e. not degenerate into full program proof). Therefore, all typed languages combine static and dynamic type-checking in various proportions.

Static type-checking amounts to an automatic proof of partial correctness of the programs that pass the compiler. The two key words here are *partial*, since only type safety guarantees are established, not full correctness; and *automatic*, since the proof is performed entirely by machine, without manual assistance from the programmer (beyond a few, easy type declarations in the source). Static type-checking can therefore be viewed as the poor man's formal methods: the guarantees it gives are much weaker than full formal verification, but it is much more acceptable to the general population of programmers.

3.2.1. Type systems and language design.

Unlike most other uses of static program analysis, static type-checking rejects programs that it cannot prove safe. Consequently, the type system is an integral part of the language design, as it determines which programs are acceptable and which are not. Modern typed languages go one step further: most of the language design is determined by the *type structure* (type algebra and typing rules) of the language and intended application area. This is apparent, for instance, in the XDuce and CDuce domain-specific languages for XML transformations [35], [32], whose design is driven by the idea of regular expression types that enforce DTDs at compile-time. For this reason, research on type systems – their design, their proof of semantic correctness (type safety), the development and proof of associated type-checking and inference algorithms – plays a large and central role in the field of programming language research, as evidenced by the huge number of type systems papers in conferences such as Principles of Programming Languages.

3.2.2. Polymorphism in type systems.

There exists a fundamental tension in the field of type systems that drives much of the research in this area. On the one hand, the desire to catch as many programming errors as possible leads to type systems that reject more programs, by enforcing fine distinctions between related data structures (say, sorted arrays and general arrays). The downside is that code reuse becomes harder: conceptually identical operations must be implemented several times (say, copying a general array and a sorted array). On the other hand, the desire to support code reuse and to increase expressiveness leads to type systems that accept more programs, by assigning a common type to broadly similar objects (for instance, the `Object` type of all class instances in Java). The downside is a loss of precision in static typing, requiring more dynamic type checks (downcasts in Java) and catching fewer bugs at compile-time.

Polymorphic type systems offer a way out of this dilemma by combining precise, descriptive types (to catch more errors statically) with the ability to abstract over their differences in pieces of reusable, generic code that is concerned only with their commonalities. The paradigmatic example is parametric polymorphism, which is at the heart of all typed functional programming languages. Many forms of polymorphic typing have been studied since then. Taking examples from our group, the work of Rémy, Vouillon and Garrigue on row polymorphism [42], integrated in OCaml, extended the benefits of this approach (reusable code with no loss of typing precision) to object-oriented programming, extensible records and extensible variants. Another example is the work by Pottier on subtype polymorphism, using a constraint-based formulation of the type system [40]. Finally, the notion of "coercion polymorphism" proposed by Cretin and Rémy[5] combines and generalizes both parametric and subtyping polymorphism.

3.2.3. Type inference.

Another crucial issue in type systems research is the issue of type inference: how many type annotations must be provided by the programmer, and how many can be inferred (reconstructed) automatically by the type-checker? Too many annotations make the language more verbose and bother the programmer with unnecessary details. Too few annotations make type-checking undecidable, possibly requiring heuristics, which is unsatisfactory. OCaml requires explicit type information at data type declarations and at component interfaces, but infers all other types.

In order to be predictable, a type inference algorithm must be complete. That is, it must not find *one*, but *all* ways of filling in the missing type annotations to form an explicitly typed program. This task is made easier when all possible solutions to a type inference problem are *instances* of a single, *principal* solution.

Maybe surprisingly, the strong requirements – such as the existence of principal types – that are imposed on type systems by the desire to perform type inference sometimes lead to better designs. An illustration of this is row variables. The development of row variables was prompted by type inference for operations on records. Indeed, previous approaches were based on subtyping and did not easily support type inference. Row variables have proved simpler than structural subtyping and more adequate for type-checking record update, record extension, and objects.

Type inference encourages abstraction and code reuse. A programmer's understanding of his own program is often initially limited to a particular context, where types are more specific than strictly required. Type inference can reveal the additional generality, which allows making the code more abstract and thus more reuseable.

3.3. Compilation

Compilation is the automatic translation of high-level programming languages, understandable by humans, to lower-level languages, often executable directly by hardware. It is an essential step in the efficient execution, and therefore in the adoption, of high-level languages. Compilation is at the interface between programming languages and computer architecture, and because of this position has had considerable influence on the design of both. Compilers have also attracted considerable research interest as the oldest instance of symbolic processing on computers.

Compilation has been the topic of much research work in the last 40 years, focusing mostly on high-performance execution ("optimization") of low-level languages such as Fortran and C. Two major results came out of these efforts: one is a superb body of performance optimization algorithms, techniques and methodologies; the other is the whole field of static program analysis, which now serves not only to increase performance but also to increase reliability, through automatic detection of bugs and establishment of safety properties. The work on compilation carried out in the Gallium group focuses on a less investigated topic: compiler certification.

3.3.1. Formal verification of compiler correctness.

While the algorithmic aspects of compilation (termination and complexity) have been well studied, its semantic correctness – the fact that the compiler preserves the meaning of programs – is generally taken for granted. In other terms, the correctness of compilers is generally established only through testing. This is adequate for compiling low-assurance software, themselves validated only by testing: what is tested is the executable code produced by the compiler, therefore compiler bugs are detected along with application bugs. This is not adequate for high-assurance, critical software which must be validated using formal methods: what is formally verified is the source code of the application; bugs in the compiler used to turn the source into the final executable can invalidate the guarantees so painfully obtained by formal verification of the source.

To establish strong guarantees that the compiler can be trusted not to change the behavior of the program, it is necessary to apply formal methods to the compiler itself. Several approaches in this direction have been investigated, including translation validation, proof-carrying code, and type-preserving compilation. The approach that we currently investigate, called *compiler verification*, applies program proof techniques to the compiler itself, seen as a program in particular, and use a theorem prover (the Coq system) to prove that the generated code is observationally equivalent to the source code. Besides its potential impact on the critical software industry, this line of work is also scientifically fertile: it improves our semantic understanding of compiler intermediate languages, static analyses and code transformations.

3.4. Interface with formal methods

Formal methods collectively refer to the mathematical specification of software or hardware systems and to the verification of these systems against these specifications using computer assistance: model checkers, theorem provers, program analyzers, etc. Despite their costs, formal methods are gaining acceptance in the critical software industry, as they are the only way to reach the required levels of software assurance.

In contrast with several other Inria projects, our research objectives are not fully centered around formal methods. However, our research intersects formal methods in the following two areas, mostly related to program proofs using proof assistants and theorem provers.

3.4.1. Software-proof codesign

The current industrial practice is to write programs first, then formally verify them later, often at huge costs. In contrast, we advocate a codesign approach where the program and its proof of correctness are developed in interaction, and we are interested in developing ways and means to facilitate this approach. One possibility that we currently investigate is to extend functional programming languages such as OCaml with the ability to state logical invariants over data structures and pre- and post-conditions over functions, and interface with automatic or interactive provers to verify that these specifications are satisfied. Another approach that we practice is to start with a proof assistant such as Coq and improve its capabilities for programming directly within Coq.

3.4.2. Mechanized specifications and proofs for programming language components

We emphasize mathematical specifications and proofs of correctness for key language components such as semantics, type systems, type inference algorithms, compilers and static analyzers. These components are getting so large that machine assistance becomes necessary to conduct these mathematical investigations. We have already mentioned using proof assistants to verify compiler correctness. We are also interested in using them to specify and reason about semantics and type systems. These efforts are part of a more general research topic that is gaining importance: the formal verification of the tools that participate in the construction and certification of high-assurance software.

4. Application Domains

4.1. High-assurance software

A large part of our work on programming languages and tools focuses on improving the reliability of software. Functional programming, program proof, and static type-checking contribute significantly to this goal.

Because of its proximity with mathematical specifications, pure functional programming is well suited to program proof. Moreover, functional programming languages such as OCaml are eminently suitable to develop the code generators and verification tools that participate in the construction and qualification of high-assurance software. Examples include Esterel Technologies's KCG 6 code generator, the Astrée static analyzer, the Caduceus/Jessie program prover, and the Frama-C platform. Our own work on compiler verification combines these two aspects of functional programming: writing a compiler in a pure functional language and mechanically proving its correctness.

Static typing detects programming errors early, prevents a number of common sources of program crashes (null dereferences, out-of bound array accesses, etc), and helps tremendously to enforce the integrity of data structures. Judicious uses of generalized abstract data types (GADTs), phantom types, type abstraction and other encapsulation mechanisms also allow static type checking to enforce program invariants.

4.2. Software security

Static typing is also highly effective at preventing a number of common security attacks, such as buffer overflows, stack smashing, and executing network data as if it were code. Applications developed in a language such as OCaml are therefore inherently more secure than those developed in unsafe languages such as C.

The methods used in designing type systems and establishing their soundness can also deliver static analyses that automatically verify some security policies. Two examples from our past work include Java bytecode verification [38] and enforcement of data confidentiality through type-based inference of information flow and noninterference properties [41].

4.3. Processing of complex structured data

Like most functional languages, OCaml is very well suited to expressing processing and transformations of complex, structured data. It provides concise, high-level declarations for data structures; a very expressive pattern-matching mechanism to destructure data; and compile-time exhaustiveness tests. Therefore, OCaml is an excellent match for applications involving significant amounts of symbolic processing: compilers, program analyzers and theorem provers, but also (and less obviously) distributed collaborative applications, advanced Web applications, financial modeling tools, etc.

4.4. Rapid development

Static typing is often criticized as being verbose (due to the additional type declarations required) and inflexible (due to, for instance, class hierarchies that must be fixed in advance). Its combination with type inference, as in the OCaml language, substantially diminishes the importance of these problems: type inference allows programs to be initially written with few or no type declarations; moreover, the OCaml approach to object-oriented programming completely separates the class inheritance hierarchy from the type compatibility relation. Therefore, the OCaml language is highly suitable for fast prototyping and the gradual evolution of software prototypes into final applications, as advocated by the popular "extreme programming" methodology.

4.5. 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.

5. Highlights of the Year

5.1. Highlights of the Year

5.1.1. Awards

In 2018, Xavier Leroy received the "Grand prix" jointly awarded by Inria and Académie des sciences.

Gergö Barany received the Best Paper Award for the paper "Finding Missed Compiler Optimizations by Differential Testing" [19] at the 27th International Conference on Compiler Construction (CC 2018).

6. New Software and Platforms

6.1. Compcert

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

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

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

6.2. Diy

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

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

6.3. Menhir

KEYWORDS: Compilation - Context-free grammars - Parsing
FUNCTIONAL DESCRIPTION: Menhir is a LR(1) parser generator for the OCaml programming language. That is, Menhir compiles LR(1) grammar specifications down to OCaml code. Menhir was designed and implemented by François Pottier and Yann Régis-Gianas.

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

6.4. OCaml

KEYWORDS: Functional programming - Static typing - Compilation
FUNCTIONAL DESCRIPTION: The OCaml language is a functional programming language that combines safety with expressiveness through the use of a precise and flexible type system with automatic type inference. The OCaml system is a comprehensive implementation of this language, featuring two compilers (a bytecode compiler, for fast prototyping and interactive use, and a native-code compiler producing efficient machine code for x86, ARM, PowerPC and System Z), a debugger, a documentation generator, a compilation manager, a package manager, and many libraries contributed by the user community.

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

6.5. PASL

KEYWORD: Parallel computing
FUNCTIONAL DESCRIPTION: PASL is a C++ library for writing parallel programs targeting the broadly available multicore computers. The library provides a high level interface and can still guarantee very good efficiency and performance, primarily due to its scheduling and automatic granularity control mechanisms.

- Participants: Arthur Charguéraud, Michael Rainey and Umut Acar
- Contact: Michael Rainey
- URL: http://deepsea.inria.fr/pasl/

6.6. ZENON

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

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

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

6.7. OPAM Builder

KEYWORDS: Ocaml - Continuous integration - Opam
FUNCTIONAL DESCRIPTION: OPAM Builder checks in real-time the installability on a computer of all packages after any modification of the repository. To achieve this result, it uses smart mechanisms to compute incremental differencies between package updates, to be able to reuse cached compilations, and switch from a quadratic complexity to a linear complexity.

- Partner: OCamlPro
- Contact: Fabrice Le Fessant
- URL: http://github.com/OCamlPro/opam-builder

6.8. TLAPS

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: Ioannis Filippidis joined the development team in November 2018 and started designing and implementing support for reasoning about TLA+'s ENABLED construct.

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

6.9. CFML

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

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

6.10. ldrgen

Liveness-driven random C code generator
KEYWORDS: Code generation - Randomized algorithms - Static program analysis
FUNCTIONAL DESCRIPTION: The ldrgen program is a generator of C code: On every call it generates a new random C function and prints it to the standard output. The generator is "liveness-driven", which means that it tries to avoid generating dead code: All the computations it generates are (in a certain, limited sense) actually used to compute the function's return value. This is achieved by generating the program backwards, in combination with a simultaneous liveness analysis that guides the random generator's choices.

- Participant: Gergö Barany
- Contact: Gergö Barany
- Publication: Liveness-Driven Random Program Generation
- URL: https://github.com/gergo-/ldrgen

7. New Results

7.1. Formal verification of compilers and static analyzers

7.1.1. The CompCert formally-verified compiler

Participants: Xavier Leroy, Daniel Kästner [AbsInt GmbH], Michael Schmidt [AbsInt GmbH], Bernhard Schommer [AbsInt GmbH].

In the context of our work on compiler verification (see section 3.3.1), since 2005, we have been developing and formally verifying a moderately-optimizing compiler for a large subset of the C programming language, generating assembly code for the ARM, PowerPC, RISC-V and x86 architectures [9]. This compiler comprises a back-end part, translating the Cminor intermediate language to PowerPC assembly and reusable for source languages other than C [8], and a front-end translating the CompCert C subset of C to Cminor. The compiler is mostly written within the specification language of the Coq proof assistant, from which Coq's extraction facility generates executable OCaml code. The compiler comes with a 100000-line machine-checked Coq proof of semantic preservation establishing that the generated assembly code executes exactly as prescribed by the semantics of the source C program.

This year, we improved the CompCert C compiler in several directions:

- A new built-in function, `__builtin_ais_annot` makes it easy to transfer annotations (also known as flow facts) written at the source code level in AbsInt's aiS annotation language all the way down to the level of the generated machine code. The aiT static analyzer for Worst-Case Execution Times, which operates at the machine code level, can then take advantage of these annotations to produce better WCET estimates.
- In preparation for a qualification with respect to industry standards for avionics software, conformance with the ISO C 1999 and ISO C 2011 standards was improved, with the addition of many diagnostics required by the standards.
- Performance of the generated code was slightly improved via changes to the heuristics for function inlining and for instruction selection.
- The semantic modeling of external function calls was made more precise, reflecting the fact that these functions can destroy some registers and some stack locations.

We released three versions of CompCert incorporating these improvements: version 3.2 in January 2018, version 3.3 in May 2018, and version 3.4 in September 2018.

Two papers on CompCert were presented at conferences. The first paper, with Daniel Kästner as lead author, was presented at the 2018 ERTS congress [22]. It describes the use of CompCert to compile software for nuclear power plant equipment developed by MTU Friedrichshafen, and the required certification of CompCert according to the IEC 60880 regulations for the nuclear industry. The second paper, with Bernhard Schommer as lead author, was presented at the 2018 WCET workshop [23]. It describes the `__builtin_ais_annot` source-level annotation mechanism mentioned above and its uses to help WCET analysis.

7.1.2. *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.

Nathanaël Courant, as part of his MPRI Master's internship and under Xavier Leroy's supervision, developed a Coq formalization of the polyhedral model. He then implemented and proved correct in Coq a code generator that produces efficient sequential code from an optimized polyhedral representation. Code generation is a delicate part of polyhedral compilation, involving complex, error-prone algorithms. Nathanaël Courant's verified code generator includes the major algorithms from Cédric Bastoul's reference paper [31]. The Coq specifications and proofs are available at https://github.com/Ekdohibs/PolyGen.

7.1.3. *Testing compiler optimizations*

Participant: Gergö Barany.

Compilers should be correct, but they should ideally also generate machine code that is as efficient as possible. Gergö Barany continued work on testing the quality of the generated code.

In a differential testing approach, one generates random C programs, compiles them with different compilers, then compares the generated code using a custom binary analysis tool. This tool finds missed optimizations by comparing criteria such as the number of instructions, the number of reads from the stack (for comparing the quality of register spilling), or the numbers of various other classes of instructions affected by optimizations of interest.

The system has found previously unreported missing optimizations in the GCC, Clang, and CompCert compilers. An article [19] was presented at the 27th International Conference on Compiler Construction (CC 2018), where it was honored with the Best Paper Award.

7.1.4. *A verified model of register aliasing in CompCert*

Participants: Gergö Barany, Xavier Leroy.

Some CPU architectures such as ARM feature register aliasing: Each of its 64-bit floating-point registers can also be accessed as two separate 32-bit halves. Modifying a superregister changes (invalidates) the data stored in subregisters and vice versa, but this behavior was not yet modeled in CompCert's semantics.

We continued work on re-engineering much of CompCert's semantic model of the register file and of the call stack. Rather than simple mappings of locations to values, the register file and the stack are now modeled more realistically as blocks of memory containing bytes that represent fragments of values. In this way, we can verify a semantic model in which a 64-bit register or stack slot may contain either a single 64-bit value or a pair of two unrelated 32-bit values. This ongoing work was presented at the workshop on Syntax and Semantics of Low-Level Languages (LOLA 2018) [25].

7.2. Language design and type systems

7.2.1. Refactoring with ornaments in ML

Participants: Thomas Williams, Lucas Baudin, Didier Rémy.

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

Williams and Rémy improved the formalisation of the lifting framework: using ornament inference, an ML program is first elaborated into a generic program, which can be seen as a template for all possible liftings of the original program. The generic program is defined in a superset of ML. It can then be instantiated with specific ornaments, and simplified back to an ML program. Williams and Rémy studied the semantics of this intermediate language and used it to prove the correctness of the lifting, using logical relations techniques. This work has been presented at POPL 2018 [12]. More technical details appear in a research report [43].

Lucas Baudin and Dider Rémy also studied the inverse transformation, disornamentation, which allows removing pieces of information from a data structure and adjusting the code accordingly. They showed that the framework of ornamentation can also be used to allow mixed ornamentation and disornamentation transformations. They also designed a new patch language to describe in a more robust manner how the code must be modified during such transformations. This enables a new class of applications, such as maintaining two views of a data structure in sync. For example, the location information in an abstract syntax tree, which is used to report error messages but obfuscates the code, can be projected away, leading to a simpler version of the code, which can then be modified and often automatically reornamented into the richer version of the code with locations. Disonamentation has been presented by Lucas Baudin at the ML 2018 workshop. Ornamentation, including mixed disornamentation, has also been presented at the MSFP 2018 workshop in Oxford.

A small prototype with ornamentation has been written by Thomas Williams and extended with disornamentation by Lucas Baudin. Thomas Williams has also started developing a new version of the prototype that will handle most of the OCaml language.

7.3. Shared-memory concurrency

7.3.1. The Linux Kernel Memory Model

Participants: Luc Maranget, Jade Alglave [University College London & ARM Ltd], Paul Mckenney [IBM Corporation], Andrea Parri [Sant'Anna School of Advanced Studies, Pisa, Italy], Alan Stern [Harvard University].

Modern multi-core and multi-processor computers do not follow the intuitive "sequential consistency" model that would define a concurrent execution as the interleaving of the executions of its constituent threads and that would command instantaneous writes to the shared memory. This situation is due both to in-core optimisations such as speculative and out-of-order execution of instructions, and to the presence of sophisticated (and cooperating) caching devices between processors and memory. Luc Maranget is taking part in an international research effort to define the semantics of the computers of the multi-core era, and more generally of shared-memory parallel devices or languages, with a clear initial focus on devices.

This year saw a publication on languages in an international conference. A multi-year effort to define a weak memory model for the Linux Kernel has yielded a scholarly paper [18] presented at the *Architectural Support for Programming Languages and Operating Systems* (ASPLOS) conference in March 2018. The article describes a formal model, the *Linux Kernel Memory Model* (LKMM), which defines how Linux kernel programs are supposed to behave. The model, a CAT model, can be simulated using the **herd** simulator, allowing programmers to experiment and develop intuitions. The model was tested against hardware and refined in consultation with Linux maintainers. Finally, the ASPLOS paper formalizes the *fundamental law of the Read-Copy-Update synchronization mechanism* and proves that one of its implementations satisfies this law. It is worth noting that the LKMM is now part of the Linux kernel source (in the `tools/`) section). Luc Maranget and his co-authors are the official maintainers of this document.

7.3.2. The ARMv8 and RISC-V memory model

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

Jade Alglave and Luc Maranget are working on a mixed-size version of the ARMv8 memory model. This model builds on the `aarch64.cat` model authored last year by Will Deacon (ARM Ltd). This ongoing work is subject to IP restrictions which we hope to lift next year.

Luc Maranget is an individual member of the memory model group of the RISC-V consortium (https://riscv.org/). Version V2.3 of the User-Level ISA Specification is now complete and should be released soon. This version features the first occurrence of a detailed memory model expressed in English, as well as its transliteration in CAT authored by Luc Maranget.

7.3.3. Work on diy

Participant: Luc Maranget.

This year, new synchronisation primitives were added to the Linux kernel memory model; ARMv8 atomic instructions were added; and more.

A more significant improvement is the introduction of *mixed-size* accesses. The tools can now handle a new view of memory, where memory is made up of elementary cells (typically *bytes*) that can be read or written as groups of contiguous cells (typically up to *quadwords* of 8 bytes). This preliminary work paves the way to the simulation of more elaborate memory models.

7.3.4. Unifying axiomatic and operational weak memory models

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

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

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

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

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

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

7.3.5. *Granularity control for parallel programs*

Participants: Umut Acar, Vitaly Aksenov, Arthur Charguéraud, Adrien Guatto [Université Paris Diderot], Mike Rainey, Filip Sieczkowski [University of Wrocław].

This year, the DeepSea team continued their work on granularity control techniques for parallel programs.

A first line of research is based on the use of programmer-supplied asymptotic complexity functions, combined with runtime measurements. This work first appeared at PPoPP 2018 [16] in the form of a brief announcement, and was subsequently accepted for publication at PPoPP 2019 as a full paper.

A second line of research, known as *heartbeat scheduling*, is based on instrumenting the runtime system so that parallel function calls are initially executed as normal function calls, by pushing a frame on the stack, and subsequently can be promoted and become independent threads. This research has been presented at PLDI 2018 [14].

7.3.6. *Theory and analysis of concurrent algorithms*

Participant: Vitaly Aksenov.

Vitaly Aksenov, in collaboration with Petr Kuznetsov (Télécom ParisTech) and Anatoly Shalyto (ITMO University), proved that no wait-free linearizable implementation of a stack using read, write, compare & swap and fetch & add operations can be help-free. This proof corrects a mistake in an earlier proof by Censor-Hillel et al. The result was published at the the International Conference on Networked Systems (NETYS 2018) [17].

Vitaly Aksenov, in collaboration with Dan Alistarh (IST Austria) and Petr Kuznetsov (Télécom ParisTech), worked on performance prediction for coarse-grained locking. They describe a simple model that can be used to predict the throughput of coarse-grained lock-based algorithms. They show that their model works well for CLH locks, and thus can be expected to work for other popular lock designs such as TTAS or MCS. This work appeared as a brief announcement at PODC 2018 [16].

The aforementioned results by Vitaly Aksenov are also covered in his Ph.D. manuscript [11].

7.4. The OCaml language and system

7.4.1. *The OCaml system*

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

This year, we released three versions of the OCaml system: versions 4.06.1 and 4.07.1 are minor releases that fix 7 and 8 issues, respectively; version 4.07.0 is a major release that introduces many improvements in usability and performance, and fixes about 40 issues. The main novelties are:

- The standard library modules were reorganized to appear as sub-modules of a new `Stdlib` module. The purpose of this reorganization is to facilitate the addition of new standard library modules while minimize risks of conflicts with user modules of the same name.
- Modules `Float` (floating-point operations) and `Seq` (sequences) were added to the standard library, taking advantage of the new organization mentioned above.
- Since 4.01, it has been possible to select a variant constructor or record field from a sub-module that is not opened in the current scope, if type information is available at the point of use. This now also works for GADT constructors.
- The GC now handles the accumulation of custom blocks in the minor heap better. This solves some memory-usage issues observed in code which allocates a large amount of small custom blocks, typically small bigarrays.

7.4.2. Package management infrastructure

Participant: Damien Doligez.

This year, Damien Doligez has worked on the `opamcheck` tool, which is designed to check the compatibility of different versions of OCaml on the whole code base of `opam`, OCaml's package manager. As a by-product of this work, he has proposed numerous fixes to the `opam` package repository and to its dependency graph.

7.4.3. Work on the compiler's test suite and build system

Participant: Sébastien Hinderer.

In 2018, Sébastien Hinderer has worked on the OCaml compiler's test suite. More precisely, he has finished porting over 800 tests in the compiler's test suite so that they can be run by the tool `ocamltest`, developed by Sébastien earlier. To achieve this, it has been necessary to extend both `ocamltest` and the domain-specific language that is used to describe how tests should be executed.

In addition, Sébastien has fixed and properly documented the procedure that is used to bootstrap the OCaml compiler. Being able to compile the compiler using itself is an important feature: it is crucial, for instance, when the compiler is released. In addition to fixing the bootstrap procedure, Sébastien has introduced a way to test this procedure through continuous integration, which guarantees that it will not be broken again in the future.

Finally, Sébastien has continued to improve and refactor the compiler's build system, and, most importantly, has replaced the hand-written configuration script by an `autoconf`-generated one, which will be part of the upcoming 4.08 release of OCaml. This represents an important step towards the ability to produce cross-compilers for OCaml, which has been a long-standing issue for the whole OCaml community.

7.4.4. Optimizing OCaml for satisfiability problems

Participants: Sylvain Conchon [LRI, Univ. Paris-Saclay], Albin Coquereau [ENSTA-ParisTech], Mohamed Iguernlala [OCamlPro], Fabrice Le fessant [OCamlPro], Michel Mauny.

This work aims at improving the performance of the Alt-Ergo SMT solver, which is implemented in OCaml. For safety reasons, and to ease reasoning about its algorithms, the implementation of Alt-Ergo uses a functional programming style and persistent data structures, which are sometimes less efficient than imperative style and mutable data. Moreover, some efficient algorithms, such as CDCL SAT solvers, are naturally expressed in an imperative style.

Following our previous work on optimizing Alt-Ergo's built-in SAT solver, some efforts were needed to enable the comparison of our solver with other SMT solvers. We developed an OCaml library for parsing and type-checking SMT-LIB2. Since Alt-Ergo natively uses a polymorphic typing discipline, and since the community needs such advanced features, we proposed an extension of the SMT-LIB2 syntax where functions may be polymorphic.

The resulting new version of Alt-Ergo was presented at the 2018 SMT Workshop in Oxford [33]. Comparisons of Alt-Ergo with other SMT solvers, mainly developed in C++, took place during the competition that is associated with the workshop. They showed that Alt-Ergo's performance is similar to that of its competitors.

Albin Coquereau's Ph.D. defense is planned for Spring 2019.

7.4.5. Improvements in Menhir

Participant: François Pottier.

In 2018, the OCaml parser of the OCaml compiler was migrated from `ocamlyacc` to Menhir, at last. François Pottier took this opportunity to partially clean up the parser, reducing redundancy by taking advantage of Menhir's features. In the future, we hope to continue to work on the OCaml parser by improving the quality of its syntax error messages.

This cleanup work was also an occasion to revisit Menhir's grammar description language: François Pottier designed and implemented a new input syntax for Menhir, which seems slightly more powerful and elegant than the previous syntax.

7.5. Software specification and verification

7.5.1. Formal reasoning about asymptotic complexity

Participants: Armaël Guéneau, Arthur Charguéraud [team Camus], François Pottier.

For a couple years, Armaël Guéneau, Arthur Charguéraud, François Pottier have been investigating the use of Separation Logic, extended with Time Credits, as an approach to the formal verification of the time complexity of OCaml programs. In particular, Armaël has developed in Coq a theory and a set of tactics that allow working with asymptotic complexity bounds. He has presented the main aspects of this work at the conference ESOP 2018 [21]. Furthermore, a key part of the machinery for working with asymptotic complexity bounds has been released as a standalone, reusable Coq library, procrastination. Armaël presented this library at the Coq Workshop in July 2018 [29].

In 2018, Armaël has worked on a more ambitious case study, namely a recent incremental cycle detection algorithm, whose amortized complexity analysis is nontrivial. A machine-checked proof has been completed; a paper is in preparation.

7.5.2. Time Credits and Time Receipts in Iris

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

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

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

Time receipts are a new concept, which (we showed) allows proving that certain undesirable events, such as integer overflows, cannot occur until a very long time has elapsed. Glen extended Iris with time receipts and proved the soundness of this extension. As an application of time credits and receipts together, Jacques-Henri Jourdan updated Charguéraud and Pottier's earlier verification of the Union-Find data structure [3] and proved that integer ranks cannot realistically overflow, even if they are stored using only $\log W$ bits, where W is the number of bits in a machine word.

This work has been first submitted to POPL 2019, then (after significant revision) re-submitted to ESOP 2019.

7.5.3. Verified Interval Maps

Participant: François Pottier.

In the setting of ANR project Vocal, which aims to build a library of verified data structures for OCaml, François Pottier carried out a formal reconstruction of "interval maps". An interval map, a data structure proposed by Bonichon and Cuoq in 2010, represents a set of possible heaps, that is, a set of mappings of integer addresses to abstract values. Interval maps are used in the Frama-C program analysis tool. François Pottier re-implemented this data structure in Coq and carried out a formal verification of its main operations. This work, which represents about 4 months of work, remains unpublished at this time. It would be desirable to publish it and to envision its integration in Frama-C; this however requires further effort.

7.5.4. Chunked Sequences

Participants: Émilie Guermeur, Arthur Charguéraud, François Pottier.

In June and July 2018, Émilie Guermeur, an undergraduate student at Carnegie Mellon University (Pittsburgh, USA) did a 6-week internship, co-advised by Arthur Charguéraud and François Pottier. She wrote a full-fledged OCaml implementation of "chunked sequences", a data structure which offers an efficient representation of sequences of elements. This data structure exists in two forms, a persistent form and an ephemeral (mutable) form; efficient conversion operations are offered. François Pottier subsequently implemented a test harness, based on afl-fuzz, which allowed us to submit Émilie's code to intensive testing and detect and fix a few bugs. This work is not yet published; we intend to pursue it in 2019, to publish the library and perhaps to verify it.

7.5.5. TLA+

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

Damien Doligez is head of the "Tools for Proofs" team in the Microsoft-Inria Joint Centre. The aim of this project is to extend the TLA+ language with a formal language for hierarchical proofs, formalizing Lamport's ideas [36]. This requires building tools to help write TLA+ specifications and mechanically check proofs.

Since October 2018, Ioannis Filippidis has been working on extending the TLAPS tool to deal with proofs of temporal properties. Under some well-defined circumstances, an occurrence of the ENABLED operator applied to a formula f can be replaced by a version of f where the primed variables are replaced by new existentially-quantified variables. The result is a first-order formula that can be sent to one of TLAPS's first-order backends. This rewriting of ENABLED suffices to prove a large class of liveness properties. Ioannis has started implementing this in TLAPS.

8. Bilateral Contracts and Grants with Industry

8.1. Bilateral Contracts with Industry

8.1.1. The Caml Consortium

Participants: Damien Doligez [**contact**], Xavier Leroy, Michel Mauny, Didier Rémy.

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.

The Consortium currently has 15 member companies:

- Aesthetic Integration
- Ahrefs
- Be Sport
- Bloomberg
- CEA
- Citrix
- Docker
- Esterel Technologies
- Facebook
- Jane Street
- Kernelyze LLC
- LexiFi
- Microsoft
- OCamlPro
- SimCorp

For a complete description of this structure, please refer to https://ocaml.org/consortium/index.html.

The Caml Consortium is being gradually phased out. In the future, it should be entirely replaced by the OCaml Foundation, described next (§8.1.2).

8.1.2. The OCaml Foundation

Participant: Michel Mauny.

In June 2018, Michel Mauny created the OCaml Software Foundation (OCSF), a structure sheltered by the Inria Foundation. The OCSF now has a few patrons. With the help of Yann Régis-Gianas, it is running the Learn-OCaml project, which aims at developing the usage of OCaml in higher education. A paper that presents the project has been accepted for publication at JFLA 2019 [20]. The OCaml Software Foundation and the Learn-OCaml project have been presented at the 2018 OCaml workshop.

The OCaml Software Foundation is expecting more patrons at the beginning of 2019, and shall organize meetings where donors discuss and produce suggestions for actions of general interest to be funded.

9. Partnerships and Cooperations

9.1. National Initiatives

9.1.1. ANR projects

9.1.1.1. Vocal

Participants: Armaël Guéneau, Xavier Leroy, François Pottier.

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

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

9.1.2. FUI Projects

9.1.2.1. Secur-OCaml

Participants: Damien Doligez, Fabrice Le Fessant.

The “Secur-OCaml” project (2015–2018) has been coordinated by the OCamlPro company, with a consortium focusing on the use of OCaml in security-critical contexts, while OCaml is currently mostly used in safety-critical contexts. Gallium has been involved in this project to integrate security features in the OCaml language, to build a new independent interpreter for the language, and to update the recommendations for developers issued by the former LaFoSec project of ANSSI. The end-of-project meeting took place in September 2018.

9.2. European Initiatives

9.2.1. FP7 & H2020 Projects

9.2.1.1. Deepsea

Participants: Umut Acar, Vitaly Aksenov, Arthur Charguéraud, Adrien Guatto, Michael Rainey.

The Deepsea project (2013–2018) is coordinated by Umut Acar and funded by FP7 as an ERC Starting Grant. Its objective is to develop abstractions, algorithms and languages for parallelism and dynamic parallelism, with applications to problems on large data sets.

9.2.2. ITEA3 Projects

9.2.2.1. Assume

Participants: Gergö Barany, Xavier Leroy, Luc Maranget.

ASSUME (2015–2018) is an ITEA3 project involving France, Germany, Netherlands, Turkey and Sweden. The French participants are coordinated by Jean Souyris (Airbus) and include Airbus, Kalray, Sagem, ENS Paris, and Inria Paris. The goal of the project is to investigate the usability of multicore and manycore processors for critical embedded systems. Our involvement in this project focuses on the formalisation and verification of memory models and of automatic code generators from reactive languages, as well as on extensions to the CompCert C compiler.

9.3. International Initiatives

9.3.1. Informal International Partners

- Princeton University: interactions between the CompCert verified C compiler and the Verified Software Toolchain developed at Princeton.
- The University of Cambridge and ARM Ltd, Cambridge and Imperial College London: formal modeling and testing of weak memory models.

10. Dissemination

10.1. Promoting Scientific Activities

10.1.1. Scientific Events Selection

10.1.1.1. Member of the Conference Program Committees

Xavier Leroy was on the program committee of CADO 2018, the special session on Compiler, Architecture, Design and Optimization of the 16th International Conference on High Performance Computing and Simulation.

Michel Mauny has been a member of the program committee of the International Symposium on Image, Video and Communications (ISIVC 2018).

François Pottier was a member of the program committee of ICFP 2018, the ACM International Conference on Functional Programming.

Didier Rémy was a member of the program committee of FLOPS 2018, the 14th International Symposium on Functional and Logic Programming.

10.1.2. Journal

10.1.2.1. Member of the Editorial Boards

Xavier Leroy is area editor (programming languages) for Journal of the ACM. He is a member of the editorial board of Journal of Automated Reasoning.

Until September 2018, Michel Mauny has been a member of the steering committee of the OCaml workshop.

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

Didier Rémy is a member of the steering committee of the ML Family workshop.

10.1.3. Research Administration

In 2018, Michel Mauny was chairman of the Scientific Committee of the Caml Consortium. He organized its annual meeting in December 2018.

Since May 2018, Michel Mauny has been Chief Executive Officer of the Inria Foundation.

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 *Deputy Scientific Director* (ADS) in charge of *Algorithmics, Programming, Software and Architecture*.

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

10.2. Teaching - Supervision - Juries

10.2.1. Teaching

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

Master (M2): "Semantics, languages and algorithms for multi-core programming", Luc Maranget, 18 HETD, Université Paris Diderot, France.

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

Master (M2): "Functional programming and type systems", Didier Rémy, 18 HETD, Université Paris Diderot, France.

Licence (L3): Jean-Marie Madiot, "Les principes des langages de programmation", 40 HETD, École Polytechnique, France.

Master (M1): Michel Mauny, "Principles of Programming Languages", 32 HETD, ENSTA-ParisTech, France.

Open lectures: Xavier Leroy, *Programmer = démontrer? La correspondance de Curry-Howard aujourd'hui*, 16 HETD, Collège de France, France.

10.2.2. Supervision

PhD: Vitaly Aksenov, "Synchronization Costs in Parallel Programs and Concurrent Data Structures", ITMO University of Saint Petersburg (Russia) and Université Paris Diderot, September 26, 2018, advised by Petr Kuznetsov and Anatoly Shalyto [11].

PhD: Pierrick Couderc, “Vérification des résultats de l’inférence du compilateur OCaml”, Université Paris-Saclay, October 23, 2018, advised by Michel Mauny et Fabrice Le Fessant [34].

PhD in progress: Albin Coquereau, “Amélioration de performances pour le solveur SMT Alt-Ergo: conception d’outils d’analyse, optimisations et structures de données efficaces pour OCaml,” Université Paris-Saclay, since October 2015, advised by Michel Mauny, Sylvain Conchon (LRI, Université Paris-Sud) and Fabrice Le Fessant.

PhD in progress: Armaël Guéneau, “Towards Machine-Checked Time Complexity Analyses”, Université Paris Diderot, since September 2016, advised by Arthur Charguéraud and François Pottier.

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

PhD in progress: Naomi Testard, “Reasoning about Effect Handlers and Cooperative Concurrency”, Université Paris Diderot, since January 2017, advised by François Pottier.

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

10.2.3. Juries

Xavier Leroy was a member of the jury for the Habilitation defense of Julien Signoles (Université Paris Sud, July 2018).

Xavier Leroy chaired the jury for the Ph.D. defense of Mario Pereira (Université Paris Sud, December 2018).

François Pottier was a reviewer for Steven Keuchel’s PhD thesis (Ghent University), defended on June 5, 2018.

François Pottier was a reviewer for Martin Clochard’s PhD thesis (Université Paris-Saclay), defended on March 30, 2018.

10.3. Popularization

10.3.1. Articles and contents

- For online publications (Interstices*, Images des Maths, Binaire, Wikipedia), and more widely blog articles

  Xavier Leroy wrote a short introduction to software sciences in general and to his lectures at Collège de France. This text was published by the “Binaire” blog of *Le Monde* [30].

10.3.2. Interventions

Gergö Barany gave a talk titled “Finding Missed Optimizations in LLVM (and other compilers)” at the 2018 European LLVM Developers Meeting, explaining his research on testing the quality of compiler optimizations to practitioners in compiler development.

11. Bibliography

Major publications by the team in recent years

[1] J. Alglave, L. Maranget, M. Tautschnig. *Herding cats: modelling, simulation, testing, and data-mining for weak memory*, in "ACM Transactions on Programming Languages and Systems", 2014, vol. 36, n^{o} 2, article no 7, http://dx.doi.org/10.1145/2627752

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

[3] A. CHARGUÉRAUD, F. POTTIER. *Verifying the Correctness and Amortized Complexity of a Union-Find Implementation in Separation Logic with Time Credits*, in "Journal of Automated Reasoning", September 2017 [*DOI :* 10.1007/s10817-017-9431-7], https://hal.inria.fr/hal-01652785

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

[5] J. CRETIN, D. RÉMY. *System F with Coercion Constraints*, in "CSL-LICS 2014: Computer Science Logic / Logic In Computer Science", ACM, 2014, article no 34, http://dx.doi.org/10.1145/2603088.2603128

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

[7] D. LE BOTLAN, D. RÉMY. *Recasting MLF*, in "Information and Computation", 2009, vol. 207, n^o 6, pp. 726–785, http://dx.doi.org/10.1016/j.ic.2008.12.006

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

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

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

Publications of the year

Doctoral Dissertations and Habilitation Theses

[11] V. AKSENOV. *Synchronization Costs in Parallel Programs and Concurrent Data Structures*, ITMO University ; Paris Diderot University, September 2018, https://hal.inria.fr/tel-01887505

Articles in International Peer-Reviewed Journals

[12] T. WILLIAMS, D. RÉMY. *A Principled Approach to Ornamentation in ML*, in "Proceedings of the ACM on Programming Languages", January 2018, pp. 1-30 [*DOI :* 10.1145/3158109], https://hal.inria.fr/hal-01666104

International Conferences with Proceedings

[13] U. A. ACAR, V. AKSENOV, A. CHARGUÉRAUD, M. RAINEY. *Provably and Practically Efficient Granularity Control*, in "PPoPP 2019 - Principles and Practice of Parallel Programming", Washington DC, United States, February 2019 [*DOI :* 10.1145/3293883.3295725], https://hal.inria.fr/hal-01973285

[14] U. A. ACAR, A. CHARGUÉRAUD, A. GUATTO, M. RAINEY, F. SIECZKOWSKI. *Heartbeat scheduling: provable efficiency for nested parallelism*, in "PLDI'18 - 39th ACM SIGPLAN Conference on Programming Language Design and Implementation", Philadelphia, United States, ACM Press, June 2018 [*DOI :* 10.1145/3192366.3192391], https://hal.inria.fr/hal-01937946

[15] V. AKSENOV, U. A. ACAR, A. CHARGUÉRAUD, M. RAINEY. *Poster: Performance challenges in modular parallel programs*, in "PPoPP 2018 - 23rd ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming", Vienna, Austria, February 2018, vol. 18 [*DOI :* 10.1145/3178487.3178516], https://hal.inria.fr/hal-01887717

[16] V. AKSENOV, D. ALISTARH, P. KUZNETSOV. *Brief Announcement: Performance Prediction for Coarse-Grained Locking*, in "PODC 2018 - ACM Symposium on Principles of Distributed Computing", Egham, United Kingdom, July 2018 [*DOI :* 10.1145/3212734.3212785], https://hal.inria.fr/hal-01887733

[17] V. AKSENOV, P. KUZNETSOV, A. SHALYTO. *On Helping and Stacks*, in "The International Conference on Networked Systems", Essaouira, Morocco, May 2018, https://hal.inria.fr/hal-01888607

[18] J. ALGLAVE, L. MARANGET, P. MCKENNEY, A. PARRI, A. STERN. *Frightening Small Children and Disconcerting Grown-ups: Concurrency in the Linux Kernel*, in "ASPLOS2018 - 23rd ACM International Conference on Architectural Support for Programming Languages and Operating Systems", Williamsburg, VA, United States, March 2018 [*DOI :* 10.1145/3173162.3177156], https://hal.inria.fr/hal-01873636

[19] G. BARANY. *Finding Missed Compiler Optimizations by Differential Testing*, in "CC'18 - 27th International Conference on Compiler Construction", Vienna, Austria, February 2018 [*DOI :* 10.1145/3178372.3179521], https://hal.inria.fr/hal-01682683

[20] C. BOZMAN, B. CANOU, R. DI COSMO, P. COUDERC, L. GESBERT, G. HENRY, F. LE FESSANT, M. MAUNY, C. MOREL, L. PEYROT. *Learn-OCaml : un assistant à l'enseignement d'OCaml*, in "Journées Francophones des Langages Applicatifs (JFLA)", Les Rousses, France, January 2019, https://hal.inria.fr/hal-01962838

[21] A. GUÉNEAU, A. CHARGUÉRAUD, F. POTTIER. *A Fistful of Dollars: Formalizing Asymptotic Complexity Claims via Deductive Program Verification*, in "ESOP 2018 - 27th European Symposium on Programming", Thessaloniki, Greece, A. AHMED (editor), LNCS - Lecture Notes in Computer Science, Springer, April 2018, vol. 10801, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018 [*DOI :* 10.1007/978-3-319-89884-1_19], https://hal.inria.fr/hal-01926485

[22] D. KÄSTNER, J. BARRHO, U. WÜNSCHE, M. SCHLICKLING, B. SCHOMMER, M. SCHMIDT, C. FERDINAND, X. LEROY, S. BLAZY. *CompCert: Practical Experience on Integrating and Qualifying a Formally Verified Optimizing Compiler*, in "ERTS2 2018 - 9th European Congress Embedded Real-Time Software and Systems", Toulouse, France, 3AF, SEE, SIE, January 2018, pp. 1-9, https://hal.inria.fr/hal-01643290

[23] B. SCHOMMER, C. CULLMANN, G. GEBHARD, X. LEROY, M. SCHMIDT, S. WEGENER. *Embedded Program Annotations for WCET Analysis*, in "WCET 2018: 18th International Workshop on Worst-Case Execution Time Analysis", Barcelona, Spain, Dagstuhl Publishing, July 2018, vol. 63 [*DOI :* 10.4230/OASICS.WCET.2018.8], https://hal.inria.fr/hal-01848686

National Conferences with Proceedings

[24] G. BARANY, G. SCHERER. *Génération aléatoire de programmes guidée par la vivacité*, in "JFLA 2018 - Journées Francophones des Langages Applicatifs", Banyuls-sur-Mer, France, January 2018, https://hal.inria.fr/hal-01682691

Conferences without Proceedings

[25] G. BARANY. *A more precise, more correct stack and register model for CompCert*, in "LOLA 2018 - Syntax and Semantics of Low-Level Languages 2018", Oxford, United Kingdom, July 2018, https://hal.inria.fr/hal-01799629

Research Reports

[26] X. LEROY, D. DOLIGEZ, A. FRISCH, J. GARRIGUE, D. RÉMY, J. VOUILLON. *The OCaml system release 4.07: Documentation and user's manual*, Inria, July 2018, pp. 1-752, https://hal.inria.fr/hal-00930213

[27] X. LEROY. *The CompCert C verified compiler: Documentation and user's manual: Version 3.4*, Inria, September 2018, pp. 1-77, https://hal.inria.fr/hal-01091802

[28] F. PESSAUX, D. DOLIGEZ. *Compiling Programs and Proofs: FoCaLiZe Internals*, Ensta ParisTech, May 2018, https://hal.archives-ouvertes.fr/hal-01801276

Other Publications

[29] A. GUÉNEAU. *Procrastination: A proof engineering technique*, July 2018, Coq Workshop 2018, The Coq Workshop 2018 is a part of FLoC 2018, https://hal.inria.fr/hal-01962659

[30] X. LEROY. *À la recherche du logiciel parfait*, November 2018, Post on the "Binaire" popular science blog of Le Monde, https://hal.inria.fr/hal-01966252

References in notes

[31] C. BASTOUL. *Code Generation in the Polyhedral Model Is Easier Than You Think*, in "PACT'04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques", IEEE Computer Society, 2004, pp. 7–16

[32] V. BENZAKEN, G. CASTAGNA, A. FRISCH. *CDuce: an XML-centric general-purpose language*, in "Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming", C. RUNCIMAN, O. SHIVERS (editors), ACM, 2003, pp. 51–63, https://www.lri.fr/~benzaken/papers/icfp03.ps

[33] S. CONCHON, A. COQUEREAU, M. IGUERNELALA, A. MEBSOUT. *Alt-Ergo 2.2*, in "Proceedings of the 16th International Workshop on Satisfiability Modulo Theories, SMT 2018", Oxford, UK, 2018, https://github.com/OCamlPro/alt-ergo/blob/next/publications/Alt-Ergo-2.2–SMT-Workshop-2018.pdf

[34] P. COUDERC. *Vérification des résultats de l'inférence de types du langage OCaml*, Université Paris-Saclay, 2018

[35] H. HOSOYA, B. C. PIERCE. *XDuce: A Statically Typed XML Processing Language*, in "ACM Transactions on Internet Technology", 2003, vol. 3, n^{o} 2, pp. 117–148, http://doi.acm.org/10.1145/767193.767195

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

[37] X. LEROY, D. DOLIGEZ, A. FRISCH, J. GARRIGUE, D. RÉMY, J. VOUILLON. *The Objective Caml system, documentation and user's manual – release 4.07*, Inria, July 2018, http://caml.inria.fr/pub/docs/manual-ocaml-4.07/

[38] X. LEROY. *Java bytecode verification: algorithms and formalizations*, in "Journal of Automated Reasoning", 2003, vol. 30, n^{o} 3–4, pp. 235–269, http://dx.doi.org/10.1023/A:1025055424017

[39] B. C. PIERCE. *Types and Programming Languages*, MIT Press, 2002

[40] F. POTTIER. *Simplifying subtyping constraints: a theory*, in "Information and Computation", 2001, vol. 170, n^{o} 2, pp. 153–183, http://gallium.inria.fr/~fpottier/publis/fpottier-ic01.ps.gz

[41] F. POTTIER, V. SIMONET. *Information Flow Inference for ML*, in "ACM Transactions on Programming Languages and Systems", January 2003, vol. 25, n^{o} 1, pp. 117–158, http://dx.doi.org/10.1145/596980.596983

[42] D. RÉMY, J. VOUILLON. *Objective ML: A simple object-oriented extension to ML*, in "24th ACM Conference on Principles of Programming Languages", ACM Press, 1997, pp. 40–53, http://gallium.inria.fr/~remy/ftp/objective-ml!popl97.pdf

[43] T. WILLIAMS, D. RÉMY. *A Principled Approach to Ornamentation in ML*, Inria, November 2017, n^{o} RR-9117 [*DOI :* 10.1145/NNNNNNN.NNNNNNN], https://hal.inria.fr/hal-01628060