Activity Report 2017

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
A2.1.2. - Object-oriented programming
A2.1.3. - 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
A2.2.3. - Run-time systems
A2.2.4. - Parallel architectures
A2.4. - 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
B9.4.1. - Computer science

1. Personnel

Research Scientists

Xavier Leroy [Team leader, Inria, Senior Researcher]
Umut Acar [Carnegie Mellon University & Inria, Advanced Research Position]
Damien Doligez [Inria, Researcher]
Fabrice Le Fessant [Inria, Researcher, until Sep 2017]
Jean-Marie Madiot [Inria, Researcher]
Luc Maranget [Inria, Researcher]
Michel Mauny [Inria, Senior Researcher]
François Pottier [Inria, Senior Researcher, HDR]

Michael Rainey [Inria, Starting Research Position]
Didier Rémy [Inria, Senior Researcher, HDR]

Post-Doctoral Fellows
Gergö Barany [Inria, from Mar 2017]
Adrien Guatto [Inria, from Sep 2017]

PhD Students
Vitalii Aksenov [Inria]
Armaël Guéneau [Université Paris Diderot]
Naomi Testard [Inria]
Thomas Williams [ENS Paris]

Technical staff
Sébastien Hinderer [Inria, Research Engineer, 80%]
Thomas Blanc [Inria, from Apr 2017 until Sep 2017]

Interns
Idir Lankri [Inria, from Jun 2017 until Jul 2017]
Danny Willems [Inria, from Feb 2017 until May 2017]

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 [45]. 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 [47] 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 [43], [41], 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 [50], 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 [48]. 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 languages 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 [46] and enforcement of data confidentiality through type-based inference of information flow and noninterference properties [49].

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 Caml language family has an impact on the teaching of programming. Caml Light 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 2017, Jacques-Henri Jourdan received the “prix du GDR GPL” (http://gdr-gpl.cnrs.fr/node/284) for his dissertation, entitled “Verasco: a Formally Verified C Static Analyzer”. Jacques-Henri was a Ph.D. student in the Gallium team, advised by Xavier Leroy.

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: In 2017, we have continued to work on a complete reimplementation of the proof manager. One objective is a cleaner interaction with the TLA$^+$ front-ends, in particular SANY, the standard parser and semantic analyzer. The reimplementation is also necessary for extending the scope of the fragment of TLA$^+$ that is handled by TLAPS, in particular full temporal logic and module instantiation.

- Participants: Damien Doligez, Stephan Merz and Martin Riener
- 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], Prashanth Mundkur [SRI International].

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:

- The support for 64-bit target processors that was initiated last year was improved and released as part of version 3.0 of CompCert. CompCert has been supporting 64-bit integer arithmetic since 2013. However, pointers and memory addresses were still assumed to be 32 bits wide. CompCert 3.0 lifts this restriction by parameterizing the compiler over the bit width of memory addresses. This required extensive changes throughout the back-end compiler passes and their correctness proofs.
- The x86 code generator, initially 32-bit only, was extended to handle 64-bit x86 as well. This is the first instantiation of the generic support for 64-bit target architectures mentioned above. This extension greatly improves the usability and performance of CompCert on servers and PCs, where x86 64-bit is the dominant architecture.
- Support for the RISC-V processor architecture was added to CompCert. Prashanth Mundkur contributed a prototype port targeting 32-bit RISC-V. Xavier Leroy extended this port to target 64-bit RISC-V as well and to integrate it in CompCert 3.1. While not commercially available yet, the RISC-V architecture is used in many academic verification projects.
- Several minor optimizations were added to address inefficiencies observed in AbsInt's customer code. The most notable one is the optimization of leaf functions to avoid return address reloads.
- Error and warning messages were improved and made more like those of GCC and Clang. Command-line flags were added to control which warning to emit and which warnings to treat as fatal errors.

We released version 3.0 of CompCert in February 2017 incorporating support for 64-bit architectures, and version 3.1 in August 2017 incorporating the other enhancements listed above.

Two papers describing industrial uses of CompCert for critical software were written, with Daniel Kästner from AbsInt as lead author. The first paper [24] was presented at the 2017 symposium of the British Safety-Critical Systems Club. The second paper [23] will be presented in January 2018 at the ERTS congress. 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.

7.1.2. A verified model of register aliasing in CompCert

Participants: Gergö Barany, Xavier Leroy.

In the setting of the ASSUME ITEA3 project, Gergö Barany and Xavier Leroy are working on implementing a CompCert back-end for the Kalray MPPA processor architecture. This architecture features pervasive register aliasing: each of its 64-bit registers can also be accessed as two separate 32-bit halves. The ARM architecture's floating-point register file is similarly aliased. Modifying a superregister invalidates the data stored in subregisters and vice versa; this behavior was not yet modeled in CompCert's semantics.

Integrating subregister aliasing in CompCert involved re-engineering much of its 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 is nearing completion.

7.1.3. Random program generation for compiler testing

Participant: Gergö Barany.

Randomized testing is a powerful tool for finding bugs in compilers. In a project aimed at finding missed compiler optimizations, Gergö Barany wanted to use such random testing techniques, but found that the standard random C program generator, Csmith, generates very large amounts of dead code. This is code whose results are never used and that can therefore be removed by the compiler.

The presence of large amounts of dead code prevents testing optimizations: almost all of the code is trivially removed by compilers' dead code elimination passes. Gergö resolved this problem by designing a new approach to random program generation. The new generator generates code backwards and performs a simultaneous liveness analysis of the program to rule out the generation of dead code. Its practical evaluation shows that it is much more efficient than Csmith at generating programs that compile to large amounts of machine code with a much more varied instruction mix than Csmith-generated code. In addition, the new generator is much faster than Csmith, because it is designed to work in a single, linear pass, without generating invalid states that cause backtracking. This work resulted in the development of the `ldrgen` tool, and was presented at LOPSTR 2017 [34].

7.1.4. 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 started work on adapting compiler correctness testing techniques for 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. The comparison is done by a custom binary analysis tool that Gergö developed for this purpose. This tool assigns scores to programs according to various 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. New criteria can be added using a simple plug-in system. If the binaries generated by different compilers are assigned different scores, the input program is considered interesting, and it is reduced to a minimal test case using an off-the-shelf program reducer (C-Reduce).

This automated process often results in small, simple examples of missed optimizations: optimizations that compilers should be able to perform, but that they failed to apply for various reasons. Gergö found previously unreported missing arithmetic optimizations, as well as individual cases of unnecessary register spilling, missed opportunities for register coalescing, dead stores, redundant computations, and missing instruction selection patterns. Several of these missed optimization issues were reported and fixed in the GCC, Clang, and CompCert compilers. An article describing this work is currently under review, and work is in progress on other binary analysis techniques that can find further missed optimizations.

7.1.5. Towards a verified compilation stack for concurrent programs

Participants: Jean-Marie Madiot, Andrew Appel [Princeton University].

The verified compiler CompCert compiles programs from C to assembly while preserving their semantics, thus allowing formal reasoning on source programs, which is much more tractable than reasoning on assembly code. It is however limited to sequential programs, running as one thread on one processor. Jean-Marie Madiot is working to extend CompCert to shared-memory concurrency *and* to provide users with techniques to reason and prove properties about concurrent programs.

Concurrent Separation Logic is used to reason about source programs and prove their correctness with respect to a "concurrent permission machine". The programs are compiled by a concurrency-aware version of CompCert. As of 2017, this has been done for the x86 architecture only.

This project is a continuation of a collaboration with Andrew Appel's team at Princeton University. Appel's team has been working for several years on the "Verified Software Toolchain" project, which provides users with tools to establish properties of sequential programs. Jean-Marie Madiot has been extending the program logic to shared-memory concurrency and developing a new proof of concurrent separation logic that is both formalised and usable in this setting. A paper has been submitted and rejected and is being improved.

Jean-Marie Madiot is now also working on a more general adaptation of CompCert to the reasoning principles of concurrency, and started a collaboration to adapt it to architectures other than x86 (see Section 7.3.4).

7.1.6. Verified compilation of Lustre

Participants: Xavier Leroy, Timothy Bourke [team Parkas], Lélio Brun [team Parkas], Pierre-Évariste Dagand [team Whisper], Marc Pouzet [team Parkas], Lionel Rieg [Yale University].

The Velus project of team Parkas develops a compiler for the Lustre reactive language that generates CompCert Clight intermediate code and is proved correct using the Coq proof assistant. A paper describing the Velus compiler and its verification was presented at the conference PLDI 2017 [20]. Xavier Leroy contributed to the verification of the final pass of Velus, the one that translates from the Obc object-oriented intermediate language of Velus to the Clight C-like, early intermediate language of CompCert. The correctness proof of this pass captures the shape of memory states during execution using formulas from separation logic. The separation logic assertions for CompCert memory states used in this proof come from a library that Xavier Leroy developed last year to help revise the proof of the "stacking" pass of CompCert, and that Timothy Bourke and Xavier Leroy later extended with a "magic wand" operator.

7.2. Language design and type systems

7.2.1. Refactoring with ornaments in ML

Participants: Thomas Williams, Didier Rémy.

Thomas Williams and Didier Rémy continued working on ornaments for program refactoring and program transformation in ML. Ornaments have been introduced as a way of describing changes in data type definitions that preserve the recursive structure but can reorganize, add, or drop pieces of data. After a new data structure has been described as an ornament of an older one, the functions that operate on the bare structure can be partially or sometimes totally lifted into functions that operate on the ornamented structure.

This year, Williams and Rémy continued working on the description of the lifting algorithm: 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 into 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. A paper has been accepted for presentation at POPL 2018 [14]. A research report gives more technical details [30].

On the practical side, several families of case studies have been explored, including refactoring and code specialization, as so as to make certain existing invariants apparent, or so as to use more efficient data structures. We improved the user interface of the prototype implementation so as to make it easier to write useful examples. We are currently developing a new version of the prototype that will handle most of the OCaml language.

7.3. Shared-memory parallelism

7.3.1. The Linux Kernel Memory Model

Participants: Luc Maranget, Jade Alglave [University College London–Microsoft Research, UK], 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 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. 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 progress as regards languages. To wit, a two-year effort to define a weak memory model for the Linux Kernel has yielded an article in the *Linux Weekly News* online technical magazine [31], and a scholarly paper accepted for publication at the *Architectural Support for Programming Languages and Operating Systems* (ASPLOS) conference in March 2018. While targeting different audiences, both articles describe a formal model that defines how Linux programs are supposed to behave. The model is of course a CAT model, hence is understood by the **herd** simulator (Section 7.3.3) that allows programmers to experiment and develop an intuition. The model has been tested against hardware and refined in consultation with maintainers. Finally, the ASPLOS article formalizes the *fundamental law of the Read-Copy-Update synchronization mechanism*, and proves that one of its implementations satisfies this law.

For the record, Luc Maranget also co-authored a paper that has been presented at POPL 2017 [22]. This work, which we described last year, is joint work with many researchers, including S. Flur and other members of P. Sewell's team (University of Cambridge) as well as M. Batty (University of Kent). Moreover, Luc Maranget still interacts with the Cambridge team, mostly by providing tests and performing comparisons between his axiomatic models and the operational models developed by this team.

7.3.2. ARMv8 and RISC-V memory models

Participants: Will Deacon [ARM Ltd], Luc Maranget, Jade Alglave [University College London–Microsoft Research, UK].

Jade Alglave and Luc Maranget helped Will Deacon, an engineer at ARM Ltd., who developed a model for the ARMv8 64-bit processor. Will wrote a CAT model, which ARM uses internally as a specification. (CAT is the domain-specific language for describing memory models and is understood by the **herd** simulator; see Section 7.3.3.) ARM's official documentation presents a natural language transliteration of the CAT model.

Luc Maranget also joined the RISC-V consortium (https://riscv.org/) as an individual and as a member of the memory model group. He takes part in the development of the memory model of this open architecture, mostly by writing CAT models and reviewing tests that will be part of the documentation. A CAT model will be part of the next version (V2.3) of the User-Level ISA Specification.

7.3.3. Improvements to the diy tool suite

Participants: Luc Maranget [**contact**], Jade Alglave [University College London–Microsoft Research, UK].

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

This year's new features are a model for the Linux Kernel developed as a collaborative effort (see Section 7.3.1) and an ongoing RISC-V model transliterated by Luc Maranget from the model elaborated by the RISC-V committee which Luc Maranget joined this year (see Section 7.3.2). Those new models were made possible due to significant extensions of **diy**, such as a new tool chain for RISC-V and the extension of the macro system so as to handle most of the memory-model-related macros used by Linux kernel developers.

7.3.4. Towards formal software verification with respect to weak memory models

Participants: Jean-Marie Madiot, Jade Alglave [University College London & Microsoft Research Cambridge], Simon Castellan [Imperial College London].

Past research efforts on weak memory models have provided both academia and industry with very efficient tools to precisely describe memory models and to carefully test them on a wide variety of architectures. While these models give us a good understanding of complex *hardware* behaviors, exploiting them to formally guarantee the good behavior of *software* remains practically out of reach.

A difficulty is that weak memory models are described in terms of properties of graphs of execution candidates. Because graphs are far from the usual way of defining programming language semantics, because execution candidates are not defined formally, and because existing proofs of “data-race freedom” (DRF) theorems are hard to fathom and formally imprecise, there is a strong demand in the programming language community for a formal account of weak memory models.

In 2017, Jean-Marie Madiot started a collaboration with weak memory model expert Jade Alglave and concurrent game semantics researcher Simon Castellan to tackle these problems. The idea is to have a formal description, using partial-order techniques similar to the ones used in game semantics, of execution candidates. On the other side, a given model of shared memory is then described in terms of partial orders, and the composition of those partial orders provides the final possible executions of a given program in a given architecture. This should yield a formal semantics for programs in a weak memory setting, and should allow proving a DRF theorem so as to connect this semantics to more standard sequentially consistent semantics. A success in this direction would finally allow tractable verification of concurrent programs, particularly in combination with Madiot’s ongoing work on a generalization to concurrency of the CompCert certified compiler (see Section 7.1.5).

7.3.5. Granularity control for parallel programs

Participants: Umut Acar, Vitaly Aksenov, Arthur Charguéraud, Adrien Guatto, Mike Rainey, Filip Sieczkowski.

The DeepSea team focused this year on the development of techniques for controlling granularity in parallel programs. Granularity control is an essential problem because creating too many tasks may induce overwhelming overheads, while creating too few tasks may harm the ability to process tasks in parallel. Granularity control turns out to be especially challenging for nested parallel programs, i.e., programs in which parallel constructs such as fork-join or parallel-loops can be arbitrarily nested. Two different approaches were investigated.

The first approach is based on the use of asymptotic complexity functions provided by the programmer, combined with runtime measurements to estimate the constant factors that apply. Combining these two sources of information allows to predict with reasonable accuracy the execution time of tasks. Such predictions may be used to guide the generation of tasks, by sequentializing computations of sufficiently-small size. An analysis is developed, establishing that task creation overheads are indeed bounded to a small fraction of the total runtime. These results builds upon prior work by the same authors [39], extending it with a carefully-designed algorithm for ensuring convergence of the estimation of the constant factors deduced from the measures, even in the face of noise and cache effects, which are taken into account in the analysis. The approach is demonstrated on a range of benchmarks taken from the state-of-the-art PBBS benchmark suite. A paper describing the results is under preparation.

The second approach is based on an instrumentation of the runtime system. The idea is to process parallel function calls just like normal function calls, by pushing a frame on the stack, and only subsequently promoting these frames as threads that might get scheduled on other cores. The promotion of frames takes place at regular time intervals, which is why we named this approach *heartbeat scheduling*. Unlike prior approaches such as *lazy scheduling*, in which promotion is guided by the workload of the system, heartbeat scheduling can be proved to induce only small scheduling overheads, and to not asymptotically reduce the amount of parallelism inherent in the program. The theory behind the approach is formalized in Coq. It is also implemented through instrumented C++ programs, and evaluated on PBBS benchmarks. A paper describing this approach was submitted to an international conference.

7.3.6. Non-zero indicators: a provably-efficient, concurrent data structure

Participants: Umut Acar, Mike Rainey.

This work, conducted in collaboration with Naama Ben David from Carnegie Mellon University, investigates the design and analysis of an implementation of a concurrent data structure called *non-zero indicator*. This data structure plays a crucial role in the scheduling of nested parallel programs: it is used to handle dependency resolution among parallel tasks. Concretely, a non-zero indicator is initialized with value 1, and it supports the following two concurrent operations, which may be invoked by threads that have knowledge that the counter is non-zero: (1) atomically increase the counter by one unit, and (2) atomically decrease the counter by one unit, and detect whether the counter reaches zero. While a trivial implementation can be set up using an atomic operation on a shared memory cell (e.g., fetch-and-add), the key challenge is to design a non-zero indicator that scales well to hundreds if not thousands of threads, without suffering from contention.

Prior work leverages dynamic tree data structures to tame contention [42]. Yet, such prior work, as well as most concurrent data structures in general, are analyzed empirically, omitting asymptotic bounds on their efficiency. In this work, we propose a new variant of a tree-based non-zero indicator implementation, for which we are able to present a formal analysis establishing bounds on the worst-case contention of concurrent updates. Our analysis is the first to achieve relevant bounds of this kind. Furthermore, we demonstrate in practice that our proposal improves scalability, compared with a naive fetch-and-add atomic counter, and also compared with the original tree-based data structure. Our work was presented at PPoPP [16].

7.3.7. Efficient sequence data structures for ML

Participants: Arthur Charguéraud, Mike Rainey.

The use of sequence containers, including stacks, queues, and double-ended queues, is ubiquitous in programming. When the maximal number of elements to be stored is not known in advance, containers need to grow dynamically. For this purpose, most ML programs rely on either lists or vectors. These data structures are inefficient, both in terms of time and space usage. In this work, we investigate the use of data structures based on *chunks*, adapting ideas from some of our prior work implemented in C++ [38]. Each chunk stores items in a fixed-capacity array. All chunks are linked together to represent the full sequence. These chunk-based structures save a lot of memory and generally deliver better performance than classic container data structures for long sequences. We measured a 2x speedup compared with vectors, and up to a 3x speedup compared with long lists. This work was presented at the ML Family Workshop [36]. Generalization of this work to double-ended sequences and to persistent sequences is under progress.

7.3.8. A parallel algorithm for the dynamic trees problem

Participants: Umut Acar, Vitaly Aksenov.

Dynamic algorithms are used to compute a property of some data while the data undergoes changes over time. Many dynamic algorithms have been proposed, but nearly all of them are sequential.

In collaboration with Sam Westrick (Carnegie Mellon University), Umut Acar and Vitaly Aksenov investigated the design of an efficient parallel dynamic tree data structure. This data structure supports four operations, namely insertion and deletion of vertices and edges; these operations can be executed in parallel. The proposed data structure is work-efficient and highly parallel. A preliminary version of this work was presented in a brief announcement at SPAA 2017 [15].

7.3.9. A concurrency-optimal binary search tree

Participant: Vitaly Aksenov.

In joint work with Vincent Gramoli (IT School of Information Technologies, Sydney), Petr Kuznetsov (Telecom ParisTech), Anna Malova (Washington University in St Louis), and Srivatsan Ravi (Purdue University), Vitaly Aksenov proposed a concurrency-optimal implementation of binary search trees. Concurrency-optimality means that the data structure allows all interleavings of the underlying sequential implementation, except those that would violate linearizability. Aksenov and co-authors show that none of the state-of-the-art concurrent binary search trees are concurrency-optimal, and they experimentally verify that the new concurrency-optimal binary search tree is competitive with known implementations. This work was presented at Euro-Par 2017 [17].

7.4. The OCaml language and system

7.4.1. The OCaml system

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

This year, we released four versions of the OCaml system: versions 4.04.1 and 4.04.2 are minor releases that fix about 16 issues; versions 4.05.0 and 4.06.0 are major releases that introduce some new features, many improvements in usability and performance, and fix about 100 issues. The most important new features are:

- Character strings are now immutable (read-only) by default. This completes the evolution of OCaml towards immutable strings that started in 2014 with the introduction of a compile-time option to separate text-like read-only strings from array-like read-write byte sequences. This option is now the default, making OCaml programs safer and clearer.
- Extensions of the "destructive substitution" operator over module signatures (*sig* `with type t := ...`) to make it more general and more widely usable.
- Support for the UTF8 encoding of Unicode characters in strings was improved with the introduction of an escape `\u{XXXX}` in string literals, and more importantly with a complete overhaul of the OCaml interface for Windows system calls that make them compatible with UTF8-encoded Unicode.
- An alternate register allocator based on linear scan was added and can be selected to reduce compilation times.

On the organization side, we switched to a deadline-based release cycle whereby a major release occurs at a set date with the features that are ready by that date, instead of waiting for a set of new features to be ready. Releases 4.05.0 and 4.06.0 were produced in this manner at 6-months intervals. Damien Doligez and Gabriel Scherer served as release managers.

Sébastien Hinderer worked on integrating `ocamltest`, the compiler's test driver he developed last year, in the 4.06 release of OCaml. He migrated a large part of the test suite from the former Makefile-based infrastructure to `ocamltest`. He also started to rewrite OCaml's build system so that the compiler can be built in parallel as much as its dependencies allow.

We have improved our Continuous Integration infrastructure by taking advantage of Jenkins features such as configuration matrices, adding five new architectures (ARM-64, Fedora, FreeBSD, PPC64-LE, Ubuntu), and upgrading to the latest version of MacOS. Our testing is now done on all of the major architectures that are officially supported by OCaml.

7.4.2. Type-checking the OCaml intermediate languages

Participants: Pierrick Couderc [ENSTA-ParisTech & OCamlPro], Grégoire Henry [OCamlPro], Fabrice Le Fessant, Michel Mauny.

This work aims at designing and implementing a consistency checker for the type-annotated abstract syntax trees (TASTs) produced by the OCaml compiler. When presented as inference rules, the different cases of this TAST checker can be read as the rules of the OCaml type system. Proving the correctness of (part of) the checker would prove the soundness of the corresponding part of the OCaml type system. A preliminary report on this work has been presented at the 17th Symposium on Trends in Functional Programming (TFP 2016).

In 2017, Pierrick Couderc formalized the consistency checker, and wrote a Coq proof of its correctness. The dissertation is being written, and Pierrick's Ph.D. defense should take place at the beginning of 2018.

7.4.3. Optimizing OCaml for satisfiability problems

Participants: Sylvain Conchon [LRI, Univ. Paris Sud], Albin Coquereau [ENSTA-ParisTech], Mohamed Iguernelala [OCamlPro], Fabrice Le Fessant, Michel Mauny.

This work aims at improving the performance of the Alt-Ergo SMT solver, implemented in OCaml. For safety reasons and to ease reasoning about its algorithms, the implementation of Alt-Ergo uses as much as possible 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.

We therefore explored the replacement of Alt-Ergo's default, functional, SAT solver by an imperative CDCL solver. In a first step, we reimplemented a C++ version of miniSAT in OCaml. A comparison of their respective performance showed that the OCaml version is slower and has more cache misses.

In a second step, we studied the use of the imperative miniSAT-like SAT solver in Alt-Ergo. The integration is actually not immediate because of the interaction between this solver and both the theories and the quantifier instantiation engines of Alt-Ergo. In fact, although the default (functional) SAT solver of Alt-Ergo is not as effective as a CDCL solver for reasoning on pure Boolean problems, its smart interaction with theories and instantation engines makes it quite effective in the context of program verification.

7.4.4. Type compatibility checking for dynamically-loaded OCaml data

Participants: Florent Balestrieri [ENSTA-ParisTech], Michel Mauny.

The SecureOCaml project (FUI 18) aims at enhancing the OCaml language and environment in order to make it more suitable for building secure applications, following the recommendations published by the French ANSSI in 2013. Florent Balestrieri (ENSTA-ParisTech) represents ENSTA-Paristech in this project for 2016 and 2017.

The first year has been dedicated to designing and producing an effective OCaml implementation that checks whether a memory graph – typically the result obtained by unmarshalling some data – is compatible with a given OCaml type, following the algorithm designed by Henry *et al.* in 2012. Because the algorithm requires a runtime representation of OCaml types, Florent Balestrieri implemented a library for generic programming in OCaml. This library was presented at the OCaml Users and Developers Workshop in 2016 [40]; an extended version of this paper has been submitted [33]. He also implemented a type-checker which, when given a type and a memory graph, checks whether the former could be the type of the latter. In 2017, Florent Balestrieri implemented a prototype type-checker for OCaml bytecode.

7.4.5. Visitors

Participant: François Pottier.

Traversing and transforming abstract syntax trees that involve name binding is notoriously difficult to do in a correct, concise, modular, customizable manner. In 2017, François Pottier addressed this problem in the setting of OCaml by proposing visitor classes as partial, composable descriptions of the operations that one wishes to perform on abstract syntax trees. By combining auto-generated visitor classes (which have no knowledge of binding) and hand-written visitor classes (each of which knows about a specific binding construct, a specific representation of names, and/or a specific operation on abstract syntax trees), a wide range of operations can be defined. A syntax extension for OCaml has been released under the name `visitors` and this work has been presented at the conference ICFP 2017 [13].

7.4.6. Improvements in Menhir

Participant: François Pottier.

In 2017, François Pottier incorporated several improvements, proposed by Frédéric Bour, to the Menhir parser generator. Many functions were added to Menhir's incremental API, which (at runtime) allows inspecting and updating the parser's state from the outside. A new library, MENHIRSDK, was introduced, which (at compile-time) allows inspecting the grammar and the automaton constructed by Menhir. Together, these improvements allow new features to be programmed outside of Menhir; the advanced error recovery mode implemented in the Merlin IDE is an example.

François Pottier also improved the termination test that takes place before parameterized symbols are expanded away. The new test, it is hoped, should reject the grammar if and only if expansion would not terminate. This improves the expressive power of the grammar description language.

7.5. Software specification and verification

7.5.1. Formal reasoning about asymptotic complexity

Participants: Armaël Guéneau, Arthur Charguéraud, François Pottier.

For several years, Arthur Charguéraud and 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. An extended version of their work on the UnionFind algorithm has appeared in the *Journal of Automated Reasoning* [11]. In this work, the complexity bounds that are established involve explicit constants: for instance, the complexity of *find* is $2\alpha(n)+4$.

Armaël Guéneau, who is supervised by Arthur Charguéraud and François Pottier, is working on relaxing this approach so as to use asymptotic bounds: e.g., the advertised complexity of *find* should be $O(\alpha(n))$. The challenge is to give a formal account of the O notation and of its properties and to develop techniques that make asymptotic reasoning as convenient in Coq as it seemingly is on paper.

For that purpose, this year, Armaël Guéneau developed two Coq libraries. A first library gives a formal definition of the O notation, provides proofs for many commonly used lemmas, as well as a number of tactics that automate the application of these lemmas. A second library implements a simple yet very useful mechanism, allowing the user to delay and collect proof obligations in Coq scripts. Using these libraries, Armaël extended the CFML tool with support for making asymptotic time complexity claims as part of specifications. He developed tactics that perform (guided) inference and resolution of recursive equations for the cost of recursive programs.

Armaël evaluated this framework on several small-scale case studies, namely simple algorithms such as binary search, selection sort, and the Bellman-Ford algorithm. This work has been accepted for publication at the conference ESOP 2018.

7.5.2. Revisiting the CPS transformation and its implementation

Participant: François Pottier.

While preparing an MPRI lecture on the CPS transformation, François Pottier did a machine-checked proof of semantic correctness for Danvy and Filinski's properly tail-recursive, one-pass, call-by-value CPS transformation.

He proposed a new first-order, one-pass, compositional formulation of the transformation. He pointed out that Danvy and Filinski's simulation diagram does not hold in the presence of `let` and proved a slightly more complex diagram, which involves parallel reduction. He suggested representing variables as de Bruijn indices and showed that, thanks to state-of-the-art libraries such as Autosubst, this does not represent a significant impediment to formalization. Finally, he noted that, given this representation of terms, it is not obvious how to efficiently implement the transformation. To address this issue, he proposed a novel higher-order formulation of the CPS transformation, proved that it is correct, and informally argued that it runs in time $O(n \log n)$.

This work has been submitted for publication in a journal.

7.5.3. Zenon

Participant: Damien Doligez.

This year, Damien Doligez did maintenance work on Zenon: updating to the latest version of OCaml and fixing a few bugs. He also started work on adding a few minor features, such as inductive proofs for mutually inductive types.

7.5.4. TLA+

Participants: Damien Doligez, Leslie Lamport [Microsoft Research], 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 [44], and to build tools for writing TLA+ specifications and mechanically checking the proofs.

Damien is still working on a new version of TLAPS and has started writing a formal description of the semantics of TLA+.

8. Bilateral Contracts and Grants with Industry

8.1. Bilateral Contracts with Industry

8.1.1. The Caml Consortium

Participants: Xavier Leroy [**contact**], Damien Doligez, 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 Caml. 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 16 member companies:

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

For a complete description of this structure, refer to http://caml.inria.fr/consortium/. Xavier Leroy chairs the scientific committee of the Consortium.

8.1.2. The OCaml Foundation

Participant: Michel Mauny.

Throughout 2017, Michel Mauny has been preparing the project of an OCaml Foundation, which should support OCaml in a more efficient way than the existing Caml Consortium could do, thanks to the facilities and flexibility provided by the recently created Inria Foundation. The goal is to raise enough funds to effectively support the development and evolution of OCaml, and to animate and grow its user and teaching communities.

8.1.3. Scientific Advisory for OCamlPro

Participant: Fabrice Le Fessant.

OCamlPro is a startup company founded in 2011 by Fabrice Le Fessant to promote the use of OCaml in the industry, by providing support, services and tools for OCaml to software companies. OCamlPro performs a lot of research and development, in close partnership with academic institutions such as IRILL, Inria and Univ. Paris Sud, and is involved in several collaborative projects with Gallium, such as the Bware ANR, the Vocal ANR and the Secur-OCaml FUI.

Since 2011, Fabrice Le Fessant has been a scientific advisor at OCamlPro, as part of a collaboration contract for Inria, to transfer his knowledge on the internals of the OCaml runtime and the OCaml compilers. Fabrice has left Inria in October 2017 to join OCamlPro on a full-time position.

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, Naomi Testard.

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”.

The library will be made available to all OCaml programmers and will be of particular interest to implementors of safety-critical OCaml programs, such as Coq, Astrée, Frama-C, CompCert, Alt-Ergo, as well as new projects. By offering verified program components, our work will provide the essential building blocks that are needed to significantly decrease the cost of developing new formally verified programs.

9.1.2. FUI Projects

9.1.2.1. Secur-OCaml

Participants: Damien Doligez, Fabrice Le Fessant.

The “Secur-OCaml” project (2015–2018) is 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 is invoved in this project to integrate security features in the OCaml language, to build a new independant interpreter for the language, and to update the recommendations for developers issued by the former LaFoSec project of ANSSI.

9.2. European Initiatives

9.2.1. FP7 & H2020 Projects

9.2.1.1. Deepsea

Participants: Umut Acar, Vitalii 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: 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.

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.
- Cambridge University and Microsoft Research Cambridge: 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 participated in the program committee of the ACM symposium on Principles of Programming Languages (POPL 2018), of the European Symposium on Programming (ESOP 2018), and of the second Principles of Secure Compilation workshop (PRISC 2018).

Jean-Marie Madiot was a member of the program committee of the Interaction and Concurrency Experience Workshop (ICE 2017).

Michel Mauny was a member of the program committee for Trends in Functional Programming in Education (TFPIE 2017).

François Pottier was program chair of the ACM SIGPLAN Workshop on Higher-Order Programming with Effects (HOPE 2017) and a member of the program committee of the Journées Françaises des Langages Applicatifs (JFLA 2018).

Mike Rainey was a member of the program committee for the IEEE International Parallel and Distributed Processing Symposium (IPDPS 2018).

Didier Rémy was a member of the program commitee of the International Symposium on Functional and Logic Programming (FLOPS 2018).

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 June 2017, he was on the editorial board for the Research Highlights column of Communications of the ACM.

Michel Mauny is 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

Until September 2017, Xavier Leroy was an appointed member of Inria's *Commission d'Évaluation*. He participated in the following Inria hiring committees: *jury d'admissibilité DR2* and *jury d'admissibilité CR1*.

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*.

10.2. Teaching - Supervision - Juries

10.2.1. Teaching

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

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

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

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

Licence: Armaël Guéneau, "Initiation à la programmation" (TP), "Projet informatique" (TP), "Concepts informatiques" (TD), "Langages et automates" (TD), 64 HETD, L1 and L2, Université Paris Diderot, France.

Licence: Thomas Williams, "Projet informatique" (TD), "Programation orientée objet et interfaces graphiques" (TD/TP), 64 HETD, L2, Université Paris Diderot, France.

10.2.2. Supervision

M1: Danny Willems, Université de Mons, supervised by François Pottier.

PhD in progress: Vitalii Aksenov, "Parallel Dynamic Algorithms", Université Paris Diderot, since September 2015, supervised by Umut Acar (co-advised with Anatoly Shalyto, ITMO University of Saint Petersburg, Russia).

PhD in progress: Pierrick Couderc (ENSTA-ParisTech & OCamlPro), "Typage modulaire du langage intermédiaire du compilateur OCaml," Université Paris-Saclay, since December 2014, supervised by Michel Mauny, Grégoire Henry (OCamlPro) and Fabrice Le Fessant.

PhD in progress: Albin Coquereau (ENSTA-ParisTech), "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, supervised 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, supervised by Arthur Charguéraud and François Pottier.

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

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

10.2.3. Juries

Xavier Leroy was on the Ph.D. committees of Quentin Carbonneaux (Yale University, August 2017) and of Gabriel Radanne (University Paris Diderot, November 2017).

François Pottier was a reviewer for the Ph.D. thesis of Sandro Stucki (École Polytechnique Fédérale de Lausanne, September 2017). He was a member of the jury for the GDR GPL dissertation award (*prix de thèse du GDR GPL*).

10.3. Popularization

Xavier Leroy wrote a popularization article describing the hunt for a hardware bug in Intel processors, which was published by the Web news site *The Next Web* [32].

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 p. , 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 p. , 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

Articles in International Peer-Reviewed Journals

[11] 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

[12] J.-H. JOURDAN, F. POTTIER. *A Simple, Possibly Correct LR Parser for C11*, in "ACM Transactions on Programming Languages and Systems (TOPLAS)", September 2017, vol. 39, n^{o} 4, pp. 1 - 36 [*DOI :* 10.1145/3064848], https://hal.archives-ouvertes.fr/hal-01633123

[13] F. POTTIER. *Visitors unchained*, in "Proceedings of the ACM on Programming Languages", August 2017, vol. 1, n^{o} ICFP, pp. 1 - 28 [*DOI :* 10.1145/3110272], https://hal.inria.fr/hal-01670735

[14] 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

[15] U. A. ACAR, V. AKSENOV, S. WESTRICK. *Brief Announcement: Parallel Dynamic Tree Contraction via Self-Adjusting Computation*, in "The 29th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '17)", Washington, United States, July 2017 [*DOI :* 10.1145/3087556.3087595], https://hal.inria.fr/hal-01664903

[16] U. A. ACAR, N. BEN-DAVID, M. RAINEY. *Contention in Structured Concurrency: Provably Efficient Dynamic Non-Zero Indicators for Nested Parallelism*, in "22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming", Austin, United States, February 2017 [*DOI :* 10.1145/3018743.3018762], https://hal.inria.fr/hal-01416531

[17] V. AKSENOV, V. GRAMOLI, P. KUZNETSOV, A. MALOVA, S. RAVI. *A Concurrency-Optimal Binary Search Tree*, in "23rd International European Conference on Parallel and Distributed Computing - Euro-Par 2017", Santiago de Compostella, Spain, August 2017, https://arxiv.org/abs/1702.04441 , https://hal.inria.fr/hal-01664898

[18] T. BALABONSKI, P. COURTIEU, L. RIEG, S. TIXEUIL, X. URBAIN. *Certified Gathering of Oblivious Mobile Robots: survey of recent results and open problems*, in "Formal Methods for Industrial Critical Systems and Automated Verification of Critical Systems (FMICS/AVOCS)", Turin, Italy, Lecture Notes in Computer Science, Springer, September 2017, vol. 10471, pp. 165-181 [*DOI :* 10.1007/978-3-319-67113-0_11], http://hal.upmc.fr/hal-01549942

[19] G. BARANY, J. SIGNOLES. *Hybrid Information Flow Analysis for Real-World C Code*, in "TAP 2017 - 11th International Conference on Tests & Proofs", Marburg, Germany, Springer, July 2017, vol. 10375, pp. 23-40 [*DOI :* 10.1007/978-3-319-61467-0_2], https://hal.inria.fr/hal-01658653

[20] T. BOURKE, L. BRUN, P.-E. DAGAND, X. LEROY, M. POUZET, L. RIEG. *A Formally Verified Compiler for Lustre*, in "PLDI 2017 - 38th ACM SIGPLAN Conference on Programming Language Design and Implementation", Barcelone, Spain, ACM, June 2017, https://hal.inria.fr/hal-01512286

[21] A. CHARGUÉRAUD, F. POTTIER. *Temporary Read-Only Permissions for Separation Logic*, in "Proceedings of the 26th European Symposium on Programming (ESOP 2017)", Uppsala, Sweden, April 2017, https://hal.inria.fr/hal-01408657

[22] S. FLUR, S. SARKAR, C. PULTE, K. NIENHUIS, L. MARANGET, K. E. GRAY, A. SEZGIN, M. BATTY, P. SEWELL. *Mixed-size Concurrency: ARM, POWER, C/C++11, and SC*, in "44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)", Paris, France, ACM, January 2017, https://hal.inria.fr/hal-01413221

[23] 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 - Embedded Real Time Software and Systems", Toulouse, France, 3AF, SEE, SIE, January 2018, https://hal.inria.fr/hal-01643290

[24] D. KÄSTNER, X. LEROY, S. BLAZY, B. SCHOMMER, M. SCHMIDT, C. FERDINAND. *Closing the Gap – The Formally Verified Optimizing Compiler CompCert*, in "SSS'17: Safety-critical Systems Symposium 2017", Bristol, United Kingdom, Developments in System Safety Engineering: Proceedings of the Twenty-fifth Safety-critical Systems Symposium, CreateSpace, February 2017, pp. 163-180, https://hal.inria.fr/hal-01399482

[25] F. POTTIER. *Verifying a Hash Table and Its Iterators in Higher-Order Separation Logic*, in "Certified Programs and Proofs", Paris, France, Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs (CPP 2017), January 2017, https://hal.inria.fr/hal-01417102

[26] M. RAAB, G. BARANY. *Challenges in Validating FLOSS Conguration*, in "OSS 2017 - The 13th International Conference on Open Source Systems", Buenos Aires, Argentina, OSS 2017: Open Source Systems: Towards Robust Practices, Springer, May 2017, vol. 496, pp. 101-114 [*DOI :* 10.1007/978-3-319-57735-7_11], https://hal.inria.fr/hal-01658595

[27] M. RAAB, G. BARANY. *Introducing Context Awareness in Unmodified, Context-unaware Software*, in "ENASE 2017 - 12th International Conference on Evaluation of Novel Approaches to Software Engineering", Porto, Portugal, April 2017, pp. 1-8, https://hal.inria.fr/hal-01658620

Research Reports

[28] X. LEROY, D. DOLIGEZ, A. FRISCH, J. GARRIGUE, D. RÉMY, J. VOUILLON. *The OCaml system release 4.06: Documentation and user's manual*, Inria, November 2017, pp. 1-726, https://hal.inria.fr/hal-00930213

[29] X. LEROY. *The CompCert C verified compiler: Documentation and user's manual: Version 3.1*, Inria, August 2017, pp. 1-68, https://hal.inria.fr/hal-01091802

[30] T. WILLIAMS, D. RÉMY. *A Principled Approach to Ornamentation in ML*, Inria, November 2017, https://hal.inria.fr/hal-01628060

Scientific Popularization

[31] J. ALGLAVE, L. MARANGET, P. MCKENNEY, A. STERN, A. PARRI. *A formal kernel memory-ordering model (Part 1 and 2)*, April 2017, Article published in the online magazine "Linux Weekly News" (LWN), available on the web at https://lwn.net/Articles/718628 and https://lwn.net/Articles/720550, https://hal.inria.fr/hal-01668178

[32] X. LEROY. *How I found a crash bug with hyperthreading in Intel's Skylake processors*, July 2017, News article at The Next Web (https://tnw.to/2tJ08uM), https://hal.inria.fr/hal-01620870

Other Publications

[33] F. BALESTRIERI, M. MAUNY. *Generic Programming in OCAML*, March 2017, working paper or preprint, https://hal.inria.fr/hal-01664286

[34] G. BARANY. *Liveness-Driven Random Program Generation*, December 2017, https://arxiv.org/abs/1709.04421 - Pre-proceedings paper presented at the 27th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2017), Namur, Belgium, 10-12 October 2017 (arXiv:1708.07854), https://hal.inria.fr/hal-01658563

[35] A. CHARGUÉRAUD, J.-C. FILLIÂTRE, M. PEREIRA, F. POTTIER. *VOCAL – A Verified OCAml Library*, September 2017, ML Family Workshop 2017, https://hal.inria.fr/hal-01561094

[36] A. CHARGUÉRAUD, M. RAINEY. *Efficient Representations for Large Dynamic Sequences in ML*, September 2017, ML Family Workshop, Poster, https://hal.inria.fr/hal-01669407

[37] M. RAAB, G. BARANY. *Introducing Context Awareness in Unmodified, Context-unaware Software*, December 2017, https://arxiv.org/abs/1702.06806 - working paper or preprint, https://hal.inria.fr/hal-01658638

References in notes

[38] U. A. ACAR, A. CHARGUÉRAUD, M. RAINEY. *Theory and Practice of Chunked Sequences*, A. S. SCHULZ, D. WAGNER (editors), Springer Berlin Heidelberg, Berlin, Heidelberg, 2014, pp. 25–36, https://doi.org/10.1007/978-3-662-44777-2_3

[39] U. A. ACAR, A. CHARGUÉRAUD, M. RAINEY. *Oracle-Guided Scheduling for Controlling Granularity in Implicitly Parallel Languages*, in "Journal of Functional Programming", November 2016, vol. 26 [*DOI :* 10.1017/S0956796816000101], https://hal.inria.fr/hal-01409069

[40] F. BALESTRIERI, M. MAUNY. *Generic Programming in OCaml*, in "OCaml 2016 - The OCaml Users and Developers Workshop", Nara, Japan, September 2016, https://hal.inria.fr/hal-01413061

[41] 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

[42] F. ELLEN, Y. LEV, V. LUCHANGCO, M. MOIR. *SNZI: Scalable NonZero Indicators*, in "Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing", PODC '07, 2007, pp. 13–22

[43] 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

[44] 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

[45] X. LEROY, D. DOLIGEZ, J. GARRIGUE, D. RÉMY, J. VOUILLON. *The Objective Caml system, documentation and user's manual – release 4.02*, Inria, August 2014, http://caml.inria.fr/pub/docs/manual-ocaml-4.02/

[46] 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

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

[48] 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

[49] 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

[50] 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