Activity Report 2016

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:

1.1.3. - Memory models
2.1.1. - Semantics of programming languages
2.1.2. - Object-oriented programming
2.1.3. - Functional programming
2.1.6. - Concurrent programming
2.1.11. - Proof languages
2.2.1. - Static analysis
2.2.2. - Memory models
2.2.3. - Run-time systems
2.2.4. - Parallel architectures
2.4.1. - Analysis
2.4.3. - Proofs
2.5.4. - Software Maintenance & Evolution
4.5. - Formal methods for security
7.1. - Parallel and distributed algorithms
7.4. - Logic in Computer Science

Other Research Topics and Application Domains:

5.2.3. - Aviation
6.1. - Software industry
6.3.1. - Web
6.5. - Information systems
6.6. - Embedded systems
9.4.1. - Computer science

1. Members

Research Scientists

Xavier Leroy [Team leader, Senior Researcher, Inria]
Umut Acar [Advanced Research position, Carnegie Mellon University]
Arthur Charguéraud [Researcher, Inria, 40%]
Damien Doligez [Researcher, Inria]
Fabrice Le Fessant [Researcher, Inria]
Luc Maranget [Researcher, Inria]
Michel Mauny [Senior Researcher, Inria]
François Pottier [Senior Researcher, Inria, HDR]
Michael Rainey [Starting Research position, Inria]
Didier Rémy [Senior Researcher, Inria, HDR]

Faculty Member

Pierre Courtieu [Associate Professor on délégation, CNAM, until Aug 2016]

Engineer
Sébastien Hinderer [Research Engineer, Inria, 70%, from Apr 2016]

PhD Students
Vitalii Aksenov [Inria]
Armaël Guéneau [ENS Lyon, from Sep 2016]
Jacques-Henri Jourdan [Inria, until Mar 2016, granted by ANR VERASCO project]
Gabriel Scherer [ENS Paris and Inria, until Jan 2016]
Thomas Williams [ENS Paris]

Post-Doctoral Fellows
Adrien Guatto [Inria, until Sep 2016]
Filip Sieczkowski [Inria, until Sep 2016]

Visiting Scientist
Andrea Parri [Sant'Anna School of Advanced Studies, Pisa, Italy, from May 2016 until Nov 2016]

Administrative Assistant
Laurence Bourcier [Inria]

Others
Jacques-Pascal Deplaix [Student Intern, Epitech, from Mar 2016 until Aug 2016]
Felipe Garay [Student Intern, Universidad de Santiago de Chile, from Feb 2016 until Apr 2016]
Ambroise Lafont [Student Intern, École Polytechnique, from Apr 2016 until August 2016]

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 [49]. 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 [52] 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 [46], [43], 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 [55], 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 [53]. Finally, the notion of "coercion polymorphism" proposed by Cretin and Rémy[3] 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 [50] and enforcement of data confidentiality through type-based inference of information flow and noninterference properties [54].

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

Xavier Leroy received the 2016 Royal Society Milner Award "in recognition of his exceptional achievements in computer programming which includes the design and implementation of the OCaml programming language".

Xavier Leroy received one of the two 2016 Van Wijngaarden Awards from Centrum Wiskunde & Informatica (Amsterdam).

Xavier Leroy received the ACM SIGPLAN Most Influential POPL Paper Award for his POPL 2006 paper, *Formal certification of a compiler back-end or: programming a compiler with a proof assistant* [51].

6. New Software and Platforms

6.1. CompCert

Participants: Xavier Leroy [**contact**], Sandrine Blazy [team Celtique], Jacques-Henri Jourdan, Bernhard Schommer [AbsInt GmbH].

The CompCert project investigates the formal verification of realistic compilers usable for critical embedded software. Such verified compilers come with a mathematical, machine-checked proof that the generated executable code behaves exactly as prescribed by the semantics of the source program. By ruling out the possibility of compiler-introduced bugs, verified compilers strengthen the guarantees that can be obtained by applying formal methods to source programs. AbsInt Angewandte Informatik GmbH sells a commercial version of CompCert with long-term maintenance.

- URL: http://compcert.inria.fr/ (academic), http://www.absint.com/compcert/ (commercial).

6.2. Diy

Participants: Luc Maranget [**contact**], Jade Alglave [Microsoft Research, Cambridge].

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 or generic (LISA) assembler 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. Recent versions also take a subset of the C language as input, so as to test and simulate the C11 model. Recent releases ("Seven") provide a new license (Cecill-B), a simplified build process and numerous features, including a simple macro system that connects the C input language and LISA annotations.

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

6.3. Menhir

Participants: François Pottier [**contact**], Yann Régis-Gianas [Université Paris Diderot].

Menhir is a LR(1) parser generator for the OCaml programming language. That is, Menhir compiles LR(1) grammar specifications down to OCaml code.

- URL: http://gallium.inria.fr/~fpottier/menhir/

6.4. OCaml

Participants: Damien Doligez [**contact**], Alain Frisch [LexiFi], Jacques Garrigue [Nagoya University], Fabrice Le Fessant, Xavier Leroy, Luc Maranget, Gabriel Scherer, Mark Shinwell [Jane Street], Leo White [Jane Street], Jeremy Yallop [OCaml Labs, Cambridge University].

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

- URL: http://ocaml.org/

6.5. OPAM Builder

Participant: Fabrice Le Fessant.

OPAM Builder checks in real time the installability on a computer of all packages after any modification of the OPAM repository. To achieve this result, it uses smart mechanisms to compute incremental differences between package updates, to be able to reuse cached compilations, and go down from quadratic complexity to linear complexity.

- URL: http://github.com/OCamlPro/opam-builder

6.6. PASL

Participants: Michael Rainey [**contact**], Arthur Charguéraud, Umut Acar.

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.

- URL: http://deepsea.inria.fr/pasl/

6.7. TLAPS

Participants: Damien Doligez [**contact**], Stefan Merz [team Veridis], Martin Riener [team Veridis].

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 independent proof steps. 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+ 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.

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

6.8. Zenon

Participants: Damien Doligez [**contact**], Guillaume Bury [CNAM], David Delahaye [CNAM], Pierre Halmagrand [team Deducteam], Olivier Hermant [MINES ParisTech].

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

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

7. New Results

7.1. Formal verification of compilers and static analyzers

7.1.1. The CompCert formally-verified compiler

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

In the context of our work on compiler verification (§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 PowerPC, ARM, and x86 architectures [7]. This compiler comprises a back-end, which translates the Cminor intermediate language to PowerPC assembly, and is reusable for source languages other than C [6]; and a front-end, which translates the CompCert C subset of C to Cminor. The compiler is mostly written within the specification language of the Coq proof assistant, out of which Coq's extraction facility generates executable OCaml code. The compiler comes with a 50000-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, the CompCert C compiler was improved in several directions:

- The proof of semantic preservation was extended to account for separate compilation and linking. (See section 7.1.2.)
- Support for 64-bit target processors was added, while keeping the original support for 32-bit processors. The x86 code generator, initially 32-bit only, was extended to handle x86 64-bit as well.
- The generation of DWARF debugging information in `-g` mode, developed last year for PowerPC, is now available for ARM and x86 as well.
- The semantics of conversions from pointer types to the `_Bool` type is fully defined again. (It was made temporarily undefined while addressing issues with comparisons between the null pointer and out-of-bound pointers.)
- More features of ISO C 2011 are supported, such as the `_Noreturn` attribute, or anonymous members of struct and union types.
- As a result of his research on implementing a correct parser for the C language (§7.1.5), Jacques-Henri Jourdan improved the implementation of the parser.

Version 2.7 of CompCert was released in June 2016, incorporating most of these enhancements, with the exception of 64-bit processor support and anonymous members, which will be released Q1 2017.

7.1.2. Separate compilation and linking in CompCert

Participants: Xavier Leroy, Chung-Kil Hur [KAIST, Seoul], Jeehoon Kang [KAIST, Seoul].

Separate compilation (of multiple C source files into multiple object files, followed by linking of the object files to produce the final executable program) has been supported for a long time by the CompCert implementation, but it was not accounted for by CompCert's correctness proof. That proof established semantic preservation in the case of a single, monolithic C source file which is compiled at once to produce the final executable, but not in the more general case of separate compilation and linking.

Version 2.7 of CompCert, released this year, extends the proof of semantic preservation in order to account for separate compilation and linking. It follows the approach described by Kang, Kim, Hur, Dreyer and Vafeiadis in their POPL 2016 paper [47] and prototyped by Kang on CompCert 2.4. In this approach, the proof considers a set of C compilation units, separately compiled to assembly then linked, and shows that the resulting assembly program preserves the semantics of the C program that would be obtained by syntactic linking of the source C compilation units. The simplicity of this approach follows from the fact that semantic preservation is still shown between whole programs (after linking); there is no need to give semantics to individual compilation units. Xavier Leroy integrated the approach of Kang *et al.* into the CompCert development, and extended it to several new optimization passes that were not present in Kang's prototype implementation.

7.1.3. Separation logic assertions for compiler verification

Participants: Xavier Leroy, Timothy Bourke [EPI Parkas], Lélio Brun [EPI Parkas], Maxime Dénès [EPI Marelle].

Separation logic is a powerful tool to reason about imperative programs. It is a Hoare-style program logic where preconditions and postconditions are assertions about the contents of mutable state. Those assertions are built in a compositional manner using a separating conjunction operator.

While effective to prove the correctness of a given program, separation logic and program logics in general are less effective to prove the correctness of a compiler or of a program transformation, in particular because it is difficult to show preservation of termination. The alternative approach that we investigated this year consists in using the assertion language of separation logic, and in particular its separating conjunction, in the context of a conventional, CompCert-style proof of semantic preservation based on simulation diagrams. Assertions from separation logic make it possible to state the invariant that relates the memory states of the program before and after the transformation in a compositional manner, simplifying the proof that this invariant is preserved through execution steps.

This approach was developed and experimentally evaluated in in three case studies.

The first case study was part of project CEEC and consisted in verifying a code generator from a domain-specific, purely-functional intermediate language down to the Clight language of CompCert. Xavier Leroy and Maxime Dénès used ad-hoc separation logic assertions to describe the memory states of the generated Clight programs, and in particular the use of pointers to return multiple function results via "out" parameters.

The second case study was a complete rewrite of the Stacking pass of the CompCert back-end and of its correctness proof, as part of the new support for 64-bit architectures (§7.1.2). For this new proof, Xavier Leroy reused and improved the separation logic assertions of the previous project, using a shallow embedding into Coq instead of a deep embedding. Separating conjunctions are used to specify the layout and current contents of the stack frames for every compiled function, in a way that accommodates 32- and 64-bit registers and pointer values equally well.

The third use takes place in the context of the verified Lustre-to-C compiler in development at team Parkas (see their activity report). The final pass of this compiler translates a simple object-oriented intermediate language, Obc, to CompCert's Clight. Timothy Bourke and Lélio Brun used the separation logic assertions from the second project to specify and reason about the Clight memory layout of the Obc nested objects. Timothy Bourke and Xavier Leroy also extended the separation logic with a "magic wand" operator. A paper on this compiler verification project is under review.

7.1.4. Formal verification of static analyzers based on abstract interpretation

Participants: Jacques-Henri Jourdan, Xavier Leroy, Sandrine Blazy [team Celtique], David Pichardie [team Celtique], Sylvain Boulmé [Grenoble INP, VERIMAG], Alexis Fouilhé [Université Joseph Fourier de Grenoble, VERIMAG], Michaël Périn [Université Joseph Fourier de Grenoble, VERIMAG].

In the context of the Verasco ANR project, we are investigating the formal specification and verification in Coq of a realistic static analyzer based on abstract interpretation. This static analyzer handles a large subset of the C language (the same subset as the CompCert compiler, minus recursion and dynamic allocation); supports a combination of abstract domains, including relational domains; and should produce usable alarms. The long-term goal is to obtain a static analyzer that can be used to prove safety properties of real-world embedded C code.

This year, Jacques-Henri Jourdan published in his PhD thesis [11] an in-depth description of the mode of operation of the current version of the Verasco static analyzer. He also presented at the NSAD workshop [24] the new algorithms used in Verasco for the abstract domain of Octagons that he developed in 2015.

7.1.5. Correct parsing of C using LR(1)

Participants: Jacques-Henri Jourdan, François Pottier.

The C programming language cannot be parsed directly using LR technology. Indeed, the grammar described in the C standard exhibits ambiguities which are addressed in English prose. On the implementation side, it is known from the folklore that one can in fact use an LALR(1) parser to parse C, provided one sets up a so-called "lexer hack" to perform on-the-fly disambiguation of tokens, guided by the current state of the parser.

However, Jacques-Henri Jourdan and François Pottier found that a correct implementation of the "lexer hack" is, surprisingly, difficult. To clarify this situation, they implemented a reference C11 parser using Menhir. They invented new techniques that improve and simplify the "lexer hack", so as to write correct yet reasonably simple C11 parsers. They also created a test suite of C programs that exhibit particularly challenging corner cases. This work is described in a paper that is currently under review.

7.1.6. A SPARK front-end for CompCert

Participants: Pierre Courtieu, Zhi Zang [Kansas University].

SPARK is a language, and a platform, dedicated to developing and verifying critical software. It is a subset of the Ada language. It shares with Ada a strict typing discipline and gives strict guarantees in terms of safety. SPARK goes one step further by disallowing certain "dangerous" features, that is, those that are too difficult to statically analyze (aliasing, references, etc). Given its dedication to safety critical software, we think that the SPARK platform can benefit from a certified compiler. We are working on adding a SPARK front-end to the CompCert verified compiler.

Defining a semantics for SPARK in Coq is previous joint work with Zhi Zang. The current front-end is based on this semantics. The compiler has been written and tested and the proofs of correctness are nearing completion.

7.2. Language design and type systems

7.2.1. Types with unique inhabitants for code inference

Participants: Gabriel Scherer [Northeastern University], Didier Rémy.

Some programming language features (coercions, type-classes, implicits) rely on inferring a part of the code that is determined by its usage context. In order to better understand the theoretical underpinnings of this mechanism, we ask: when is it the case that there is a unique program that could have been guessed, or in other words, that all possible guesses result in equivalent program fragments? Which types have a unique inhabitant?

To approach the question of uniqueness, we build on work in proof theory on canonical representations of proofs. Using the proofs-as-programs correspondence, we adapt the logical technique of focusing to obtain canonical program representations.

In the setting of simply-typed lambda-calculus with sums, equipped with the strong $\beta\eta$-equivalence, we show that uniqueness is decidable. We present a saturating focused logic that introduces irreducible cuts on positive types "as soon as possible". Goal-directed proof search in this logic gives an effective algorithm that returns either zero, one or two distinct inhabitants for any given type.

This work, which was previously presented at a conference [56] and was the main part of Scherer's PhD dissertation [12], has been submitted for journal publication.

7.2.2. 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 to describe some changes in data type definitions that preserve their recursive structure, reorganizing, adding, or dropping some pieces of data. After a new data structure has been described as an ornament of an older one, some functions operating on the bare structure can be partially or sometimes totally lifted into functions operating on the ornamented structure.

We have continued working on the decomposition of the algorithm in several steps. Using ornament inference, we first elaborate an ML program 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. We studied the semantics of this intermediate language and used them to prove the correctness of the lifting, using logical relations techniques. A paper describing this process was submitted to PLDI.

On the practical side, we updated our prototype implementation to match our theoretical presentation: we create the generic program, then instantiate it. We then simplify the resulting term so that it remains readable to the programmer, and output an ML program. In the case of refactoring (the representation of a data type is modified without adding any data), the transformation is still fully automatic.

7.3. Shared-memory parallelism

7.3.1. Weak memory models

Participants: Luc Maranget, Jade Alglave [University College London–Microsoft Research, UK], Patrick Cousot [New York University], Andrea Parri [Sant'Anna School of Advanced Studies, Pisa, Italy].

Modern multi-core and multi-processor computers do not follow the intuitive "Sequential Consistency" model that would define a concurrent execution as the interleaving of the executions of its constituent threads and that would command instantaneous writes to the shared memory. This situation is due both to in-core optimisations such as speculative and out-of-order execution of instructions, and to the presence of sophisticated (and cooperating) caching devices between processors and memory. Luc Maranget took 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 focus on devices.

More precisely, in 2016, Luc Maranget pursued his collaboration with Jade Alglave and Patrick Cousot to extend "Cats", a domain-specific language for defining and executing weak memory models. Last year, a long article that presents a precise semantics for "Cats" and a study and formalisation of the HSA memory model was submitted. (The Heterogeneous System Architecture foundation is an industry standards body targeting heterogeneous computing devices.) As this article was rejected, a new paper, focused on the "Cats" semantics, was submitted this year, while the definition of the HSA memory model was made available on the web site of the HSA foundation (http://www.hsafoundation.com/standards/).

This year, our team hosted Andrea Parri, a Ph.D. student (supervised by Mauro Marinoni at Sant'Anna School of Advanced Studies, Pisa, Italy), for six months. Luc Maranget and Andrea Parri collaborated with Paul McKenney (IBM), Alan Stern (Harvard University) and Jade Alglave on the definition of a memory model for the Linux kernel. A preliminary version of this work was presented by Paul McKenney at the *2016 Linux Conference Europe*. While invited at the Dagstuhl seminar "*Concurrency with Weak Memory Models...*", Luc Maranget demonstrated the Diy toolsuite and the "Cats" language. It is worth noting that Cats models are being used independently of us by other researchers, most notably by Yatin Manerkar and Caroline J. Trippel (Princeton University) who discovered an anomaly in the published compilation scheme of the C11 language down to the Power architecture.

Luc Maranget also co-authored a paper that will be presented at POPL 2017 [23]. This work describes memory-model-aware "mixed-size" semantics for the ARMv8 architecture and for the C11 and Sequential Consistency models. A mixed-size semantics accounts for the behaviour of systems that access memory at different granularity levels (bytes, words, etc.) This is joint work with many researchers, including Shaked Flur and other members of Peter Sewell's team (University of Cambridge) as well as Mark Batty (University of Kent).

7.3.2. Algorithms and data structures for parallel computing

Participants: Umut Acar, Vitalii Aksenov, Arthur Charguéraud, Adrien Guatto, Michael Rainey, Filip Sieczkowski.

The ERC Deepsea project, with principal investigator Umut Acar, started in June 2013 and is hosted by the Gallium team. This project aims at developing techniques for parallel and self-adjusting computation in the context of shared-memory multiprocessors (i.e., multicore platforms). The project is continuing work that began at Max Planck Institute for Software Systems between 2010 and 2013. As part of this project, we are developing a C++ library, called PASL, for programming parallel computations at a high level of abstraction. We use this library to evaluate new algorithms and data structures. We obtained four main results this year.

Our first result is a calculus for parallel computing on hardware shared-memory computers such as modern multicores. Many languages for writing parallel programs have been developed. These languages offer several distinct abstractions for parallelism, such as fork-join, async-finish, futures, etc. While they may seem similar, these abstractions lead to different semantics, language design and implementation decisions. In this project, we consider the question of whether it would be possible to unify these approaches to parallelism. To this end, we propose a calculus, called the *DAG-calculus*, which can encode existing approaches to parallelism based on fork-join, async-finish, and futures, and possibly others. We have shown that the approach is realistic by presenting an implementation in C++ and by performing an empirical evaluation. This work was presented at ICFP 2016 [18].

Our second result is a concurrent data structure that may be used to efficiently determine when a concurrently-updated counter reaches the value zero. Our data structure extends an existing data structure called SNZI [44]. While the latter imposes a fixed number of threads, our structure is able to dynamically grow in response to the increasing degree of concurrency in the system. We use our dynamic non-zero indicator data structure to derive an efficient runtime representation of async/finish programs. The async/finish paradigm for expressing parallelism is one that, in the past decade, has become a part of many research-language implementations (e.g. X10) and is now gaining traction in a number of mainstream languages, most notably Java. The implementation of async/finish is challenging because the finish-block mechanism permits, and even encourages, computations in which a large number of threads are required to synchronize on shared barriers, and this number is not

statically known. We present an implementation of async/finish and prove that, in a model that takes contention into account, the cost of synchronization of the async-ed threads is amortized constant time, regardless of the number of threads. We also present experimental evaluation suggesting that the approach performs well in practice. This work has been accepted for publication at PPoPP [17].

Our third result is an extended, polished presentation of our prior work on granularity control for parallel algorithms using user-provided complexity functions. Granularity control denotes the problem of controlling the size of parallel threads created in implicitly parallel programs. If small threads are executed in parallel, the overheads due to thread creation can overwhelm the benefits of parallelism. If large threads are executed sequentially, processors may spin idle. In our work, we show that, if we have an oracle able to approximately predict the execution time of every sub-task, then there exists a strategy that delivers provably good performance. Moreover, we present empirical results showing that, for simple recursive divide-and-conquer programs, we are able to implement such an oracle simply by requiring the user to annotate functions with their asymptotic complexity. The idea is to estimate the constant factors that apply by conducting measures at runtime. This work is described in depth in an article published in the Journal of Functional Programming (JFP) [13].

Our fourth result is an extension of our aforementioned granularity control approach, with three major additions. First, we have developed an algorithm that ensures convergence of the estimators associated with the constant factors for all fork-join programs, and not just for a small class of programs. Second, we have built a theoretical analysis establishing bounds for the overall overheads of the convergence phase. Third, we have developed a C++ implementation accompanied with an extensive experimental study covering several benchmarks from the Problem Based Benchmark Suite (PBBS), a collection of high-quality parallel algorithms that delivers state-of-the-art performance. Even though our approach does not leverage a specific compiler and does not require any magic constant to be hard-coded in the source programs, our code either matches or exceeds the performance of the authors' original, hand-tuned codes. An article describing this work is in preparation.

7.4. The OCaml language and system

7.4.1. OCaml

Participants: Damien Doligez, Alain Frisch [Lexifi SAS], Jacques Garrigue [University of Nagoya], Sébastien Hinderer, Fabrice Le Fessant, Xavier Leroy, Luc Maranget, Gabriel Scherer, Mark Shinwell [Jane Street], Leo White [Jane Street], Jeremy Yallop [OCaml Labs, Cambridge University].

This year, we released versions 4.03.0 and 4.04.0 of the OCaml system. These are major releases that introduce a large number of new features. The most important features are:

- A new optimization subsystem called *flambda*, which does inlining and specialization of functions as well as static allocation of some data structures, etc.
- *ephemerons*: a generalization of weak pointers that is better suited for memoization of mutually-recursive functions.
- A fine-grained memory profiler to help programmers understand the allocation behavior of their programs.
- *unboxed types*: a user-controlled optimized representation for some simple data types.

7.4.2. Infrastructure for OCaml

Participant: Sébastien Hinderer.

Sébastien Hinderer worked on improving the test infrastructure of the OCaml compiler. These tests aim at verifying that the compiler works as expected. Currently, they are driven by a set of Makefiles which are hard to maintain and extend and make it difficult to add new tests. Sébastien developed the `ocamltest` driver, which parses test descriptions written in a domain-specific language and runs the appropriate tests.

Sébastien Hinderer also worked on merging the Makefiles used for building the compiler under Unix and Windows. The existence of separate sets of Makefiles, which is the result of a long development history, makes it especially hard to maintain and extend the compiler's build system. Sébastien worked on eliminating this redundancy, so that a single build system can be used on every platform. This is a prerequisite for using the GNU `autoconf` tools and for building easy-to-use cross-compilers for OCaml. A cross-compiler is required, for instance, to build iOS apps using OCaml.

7.4.3. Continuous integration of OCaml packages

Participant: Fabrice Le Fessant.

OPAM is a repository of OCaml source packages. It is now advertised as the official way of installing the OCaml distribution. To maintain a high level of quality for the thousands of source packages distributed in the repository, it is crucial to provide feedback to the developers on the impact of their modifications to the repository, in real-time, despite the high churn and the cascading costs of package recompilations.

We have designed and prototyped a simple modular architecture for a service that monitors the OPAM repository, and triggers recompilation of packages that are impacted by the latest modifications to the repository, for all major and minor OCaml versions since 3.12.1. Previous attempts to design such a system have failed to scale, although they targeted cloud systems of thousands of virtual machines. On the contrary, the new prototype has been deployed on a single quadcore server, and has been able to follow the OPAM repository for eight months, providing feedback in almost real-time. To achieve such a result, it uses many optimizations and caching techniques, to make recompilations as incremental as possible [37].

7.4.4. Global analyses of OCaml programs

Participants: Thomas Blanc [ENSTA-ParisTech & OCamlPro], Pierre Chambart [OCamlPro], Vincent Laviron [OCamlPro], Fabrice Le Fessant, Michel Mauny.

Exception handling in OCaml can be used for managing and reporting errors, as well as to express complex control flow constructs. As such, exceptions can be the source of errors, when, for instance, a function that may raise an exception is called in a context where this exception cannot be handled. In such situations, the program may fail unexpectedly, and the source of the error can be difficult to identify.

This work aims at performing global static analyses of OCaml programs using abstract interpretation techniques, with a particular focus on the detection of uncaught exceptions. Starting from one of the OCaml intermediate languages, we produce a hypergraph that represents the program to be analyzed. Each node of this hypergraph is a program state and each edge is an operation. Operations that may or may not raise an exception (such as function calls) have one or two successors. A fixpoint iteration is then performed on the graph, where function application edges are dynamically replaced by the corresponding subgraphs. In essence, environment information is propagated through the graph, adding at each node a superset of all possible values of each variable, until no additional information can be found. A description of the framework was presented at the 2015 OCaml workshop. We expect concrete results as well as Thomas Blanc's thesis manuscript during 2017.

7.4.5. Type-checking the OCaml intermediate languages

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

This work aims at propagating type information through the intermediate languages used by the OCaml compiler. We started by the design and implementation of a consistency checker of the type-annotated abstract syntax trees (TASTs) produced by the OCaml compiler. It appears that, 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).

7.4.6. Optimizing OCaml for satisfiability problems

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

This work aims at improving the performance of the Alt-Ergo SMT solver, implemented in OCaml. For safety reasons, the implementation of Alt-Ergo uses as much as possible a functional programming style and persistent data structures, which are sometimes less efficient that the imperative style and mutable data structures. We would like to first obtain a better understanding of the OCaml memory and cache behavior, so as to understand where efficiency could be gained, and then design dedicated data structures (for instance, semi-persistent data structures) and compare their efficiency to the current ones. This work is still at a preliminary stage: we have selected benchmarks and profiled their execution in order to discover sources of inefficiency.

7.4.7. Type compatibility checking for dynamically loaded OCaml data

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

The SecurOCaml project (FUI 18) aims at enhancing the OCaml language and environment in order to make it more suitable for building secure applications, following recommendations published by the French ANSSI in 2013. Michel Mauny and Florent Balistrieri (ENSTA-ParisTech) represent ENSTA-Paristech in this project for the two-year period 2016-2017.

The goal of this first year was to design and produce an effective OCaml implementation that checks whether a memory graph – typically the result obtained by un-marshalling some data – is compatible with a given OCaml type, following the algorithm designed by Henry *et al.* in 2012. As the algorithm needs a runtime representation of OCaml types, Florent Balestrieri implemented a library for generic programming in OCaml [21]. 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. The algorithm handles sharing and polymorphism, but currently supports neither functional values nor existential types.

7.4.8. Pattern matching

Participants: Luc Maranget, Gabriel Scherer [Northeastern University, Boston], Thomas Réfis [Jane Street LLC].

A new pattern matching diagnostic message, which should help OCaml programmers to detect rare but vicious programming errors, was integrated in the yearly release of the OCaml compiler, and was presented at the OCaml Users and Developers Workshop [39].

7.4.9. Error diagnosis in Menhir parsers

Participant: François Pottier.

In 2015, François Pottier proposed a reachability algorithm for LR automata, which he implemented in the Menhir parser generator. He applied this approach to the C grammar in the front-end of the CompCert compiler, therefore allowing CompCert to produce better syntax error messages. This work has been presented at the conferences JFLA 2016 [31] and CC 2016 [26].

7.5. Software specification and verification

7.5.1. Step-indexing in program logics

Participant: Filip Sieczkowski.

Filip Sieczkowski pursued a line of work focused on techniques for formal reasoning about programs, in joint work with Lars Birkedal (Aarhus University) and Kasper Svendsen (Cambridge University). A modern and successful approach to grounding programs logics is to rely on so-called step-indexed models. Filip and his co-authors solved a problem that arises in most step-indexed models, due to a tight coupling between the unfoldings of a recursive domain equation and evaluation steps. Their approach is based on the use of transfinite step-indexing. This work appeared at ESOP 2016 [29].

7.5.2. 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 [48], and to build tools for writing TLA+ specifications and mechanically checking the proofs.

Our rewrite of the TLAPS tools is almost done and we hope to do a first release in the first quarter of 2017.

7.5.3. Hash tables and iterators: a case study in program verification

Participant: François Pottier.

In the setting of the Vocal ANR project, François Pottier developed the the specification and proof of an (imperative, sequential) hash table implementation, as found in the module `Hashtbl` of OCaml's standard library. This data structure supports the usual dictionary operations (insertion, lookup, and so on), as well as iteration via folds and iterators. The code was verified using higher-order separation logic, embedded in Coq, via Charguéraud's CFML tool and library. This work was presented at CPP 2017 [27]. It can be viewed as a case study that should help prepare the way for verifying other modules in the Vocal library.

7.5.4. Read-only permissions in separation logic

Participants: Arthur Charguéraud, François Pottier.

Separation Logic, as currently implemented in Charguéraud's CFML tool and library, imposes a simple ownership discipline on mutable heap-allocated data structures: a thread either has full read-write access to a data structure, or has no access at all. This implies, for instance, that two threads cannot temporarily share read-only access to a data structure. There exist more flexible disciplines in the literature, such as "fractional permissions" and "share algebras", but they are much more complex.

In the setting of the Vocal ANR project, Arthur Charguéraud and François Pottier noted that it would be desirable to define an extension of Separation Logic that allows temporary shared read-only access, yet remains very simple. They proposed a general mechanism for temporarily converting any assertion (or "permission") to a read-only form. The metatheory of this proposal has been verified in Coq. This work will be presented at ESOP 2017 [42].

Charguéraud and Pottier believe that this mechanism should allow more concise specifications and proofs. This remains to be confirmed, in future work, via an implementation in CFML and case studies in the Vocal project.

7.5.5. Formal reasoning about asymptotic complexity

Participants: Armaël Guéneau, Arthur Charguéraud, François Pottier.

Armaël Guéneau started his Ph.D. at Gallium in September 2016, supervised by Arthur Charguéraud and François Pottier. In the line of his previous M2 internship at Gallium, he continued his work on asymptotic reasoning in Coq. The challenge is to give a formal definition of the well-known big-O notation, covering both single-variable and multiple-variable scenarios, to establish its fundamental properties, and to define tactics that make asymptotic reasoning as convenient in Coq as it seemingly is on paper. The ultimate goal is to apply these techniques to machine-checked proofs of the asymptotic time complexity of programs.

7.5.6. Certified distributed algorithms for autonomous mobile robots

Participant: Pierre Courtieu.

The variety and complexity of the tasks that can be performed by autonomous robots are increasing. Many applications envision groups of mobile robots that self-organise and cooperate toward the resolution of common objectives, in the absence of any central coordinating authority.

Pierre Courtieu is elaborating a verification platform, based on Coq, for distributed algorithms for autonomous robots. (This is joint work with Xavier Urbain, Sebastien Tixeuil and Lionel Rieg.) As part of this effort, Pierre Courtieu designed and verified a protocol for mobile robots that achieves the "gathering" task in all cases where it has not been proved impossible [34], [35].

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, 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 14 member companies:

- Aesthetic Integration
- Ahrefs
- Bloomberg
- CEA
- Citrix
- Dassault Aviation
- Esterel Technologies
- Facebook
- Jane Street
- Kernelyze
- 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. 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 is 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.

9. Partnerships and Cooperations

9.1. National Initiatives

9.1.1. ANR projects

9.1.1.1. BWare

Participants: Damien Doligez, Fabrice Le Fessant.

The "BWare" project (2012–2016) is coordinated by David Delahaye at Conservatoire National des Arts et Métiers and funded by the *Ingénierie Numérique et Sécurité* programme of *Agence Nationale de la Recherche*. BWare is an industrial research project that aims to provide a mechanized framework to support the automated verification of proof obligations coming from the development of industrial applications using the B method and requiring high guarantees of confidence.

9.1.1.2. Verasco

Participants: Jacques-Henri Jourdan, Xavier Leroy.

The "Verasco" project (2012–2016) is coordinated by Xavier Leroy and funded by the *Ingéniérie Numérique et Sécurité* programme of *Agence Nationale de la Recherche*. The objective of this 4.5-year project is to develop and formally verify a static analyzer based on abstract interpretation, and interface it with the CompCert C verified compiler.

9.1.1.3. Vocal

Participants: Xavier Leroy, François Pottier.

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

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. FSN projects

9.1.2.1. ADN4SE

Participants: Damien Doligez, Martin Riener.

The "ADN4SE" project (2012–2016) is coordinated by the Sherpa Engineering company and funded by the *Briques Génériques du Logiciel Embarqué* programme of *Fonds national pour la Société Numérique*. The aim of this project is to develop a process and a set of tools to support the rapid development of embedded software with strong safety constraints. Gallium is involved in this project to provide tools and help for the formal verification in TLA+ of some important aspects of the PharOS real-time kernel, on which the whole project is based.

9.1.3. FUI Projects

9.1.3.1. 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, Michael Rainey, Filip Sieczkowski.

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. Inria International Partners

9.3.1.1. 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 Organisation

10.1.1.1. Member of the Organizing Committees

Michel Mauny is a member of the steering committee of the OCaml workshop.

Didier Rémy was a member of the steering committee of the OCaml workshop until September 2017. He is a member of the steering committee of the ML Family workshop.

10.1.2. Scientific Events Selection

10.1.2.1. Member of the Conference Program Committees

Xavier Leroy was a member of the program committees of the Compiler Construction conference (CC 2016), of the conference on Interactive Theorem Proving (ITP 2016), and on the external review committee of the symposium on Principles of Programming Languages (POPL 2017).

François Pottier was a member of the program committees of the conferences Journées Francophones des Langages Applicatifs (JFLA 2017) and Compiler Construction (CC 2017).

10.1.2.2. Reviewer

In 2016, the members of Gallium reviewed at least 30 conference submissions.

10.1.3. Journal

10.1.3.1. Member of the Editorial Boards

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

François Pottier is an editor for the Journal of Functional Programming.

10.1.4. Invited Talks

Xavier Leroy was an invited speaker at the ICALP conference (Rome, July 2016).

10.1.5. Research Administration

Xavier Leroy is *délégué scientifique adjoint* of Inria Paris and appointed member of Inria's *Commission d'Évaluation*. He participated in the following Inria hiring and promotion committees: *jury d'admissibilité DR2*, *promotions CR1*, and *promotions DR1*.

Xavier Leroy was a member of the hiring committee for a professor position at Université de Lorraine.

Xavier Leroy was a member of the HCERES evaluation panel for the LORIA laboratory.

François Pottier is a member of the *Commission de Développement Technologique* and (as of January 2016) chairs the *Comité de Suivi Doctoral* of Inria Paris.

Didier Rémy is *Deputy Scientific Director* (ADS) in change of *Algorithmics, Programming, Software and Architecture*.

10.2. Teaching - Supervision - Juries

10.2.1. Teaching

Master: Xavier Leroy and Didier Rémy, "Functional programming languages", 15+18h, M2 (MPRI), Université Paris Diderot, France.

Master: Luc Maranget, "Semantics, languages and algorithms for multi-core programming", 13.5h, M2 (MPRI), Université Paris Diderot, France.

Master: "Principles of Programming Languages", 32h, M1, ENSTA-ParisTech, France.

Licence: François Pottier, "Programmation avancée" (INF441), 20h, L3, École Polytechnique, France.

Master: François Pottier, "Compilation" (INF564), 20h, M1, École Polytechnique, France.

Licence: Michael Rainey and Umut Acar, "Theory and practice of parallel computing" (part of a longer course entitled 15-210, "Parallel and Sequential Data Structures and Algorithms"), 9h, L3, Carnegie Mellon University, USA.

Michel Mauny has been a Professor at ENSTA-ParisTech from August 1st, 2005 to July 31st, 2016. While at ENSTA-ParisTech, Michel Mauny was in charge of the specialization "Architecture and Security of Information Systems" (MSc. 2nd year).

François Pottier has been a Professeur Chargé de Cours at École Polytechnique from September 1st, 2004 to August 31st, 2016.

Didier Rémy is Inria's delegate in the pedagogical team of the MPRI.

Fabrice Le Fessant has been involved in the second edition of the OCaml MOOC on the FUN platform, in coordination with the OCamlPro team in charge of the development of the exercise platform [33].

10.2.2. Supervision

M2 (Master Pro): Jacques-Pascal Deplaix, Epitech, supervised by François Pottier.

M2 (MPRI): Ambroise Lafont, École Polytechnique, supervised by Xavier Leroy.

PhD: Pierre Halmagrand, "Automated Deduction and Proof Certification for the B Method" [45], Conservatoire National des Arts et Métiers, defended December 10, 2016, supervised by David Delahaye, Damien Doligez and Olivier Hermant.

PhD: Jacques-Henri Jourdan, "Verasco: a formally verified C static analyzer" [11], Université Paris Diderot, defended May 2016, supervised by Xavier Leroy.

PhD: Gabriel Scherer, "Which types have a unique inhabitant?" [12], Université Paris Diderot, defended March 2016, supervised by Didier Rémy.

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: Thomas Blanc (ENSTA-ParisTech & OCamlPro), "Analyses de programmes complets, application à OCaml", Université Paris-Saclay, since February 2014, supervised by Michel Mauny and Pierre Chambart (OCamlPro).

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

10.2.3. Juries

François Pottier was a reviewer for the Ph.D. thesis of Benoît Vaugon, Université Paris-Saclay, March 2016. He was a reviewer for the Habilitation of Damien Pous, ENS Lyon, September 2016. He was a member of the jury for the Ph.D. thesis of Léon Gondelman, Université Paris-Saclay, December 2016.

Xavier Leroy was on the Ph.D. committee of Pierre Wilke, Université Rennes 1, November 2016.

Didier Rémy was chair of the Ph.D. committee of Raphaël Cauderlier, Conservatoire National des Arts et Métiers (CNAM), October 2016.

10.3. Popularization

Xavier Leroy gave a popularization talk on formal methods at the plenary days of Inria's DGD-T (may 2016) and another on critical avionics software for first-year students at École Polytechnique (june 2016).

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° 2, article no 7 p. , http://dx.doi.org/10.1145/2627752

[2] 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

[3] 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

[4] 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

[5] 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

[6] 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

[7] 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

[8] F. POTTIER. *Hiding local state in direct style: a higher-order anti-frame rule*, in "Proceedings of the 23rd Annual IEEE Symposium on Logic In Computer Science (LICS'08)", IEEE Computer Society Press, June 2008, pp. 331-340, http://dx.doi.org/10.1109/LICS.2008.16

[9] F. POTTIER, J. PROTZENKO. *Programming with permissions in Mezzo*, in "Proceedings of the 18th International Conference on Functional Programming (ICFP 2013)", ACM Press, 2013, pp. 173–184, http://dx.doi.org/10.1145/2500365.2500598

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

Publications of the year

Doctoral Dissertations and Habilitation Theses

[11] J.-H. JOURDAN. *Verasco: a Formally Verified C Static Analyzer*, Universite Paris Diderot-Paris VII, May 2016, https://hal.archives-ouvertes.fr/tel-01327023

[12] G. SCHERER. *Which types have a unique inhabitant?: Focusing on pure program equivalence*, Université Paris-Diderot, March 2016, https://hal.inria.fr/tel-01309712

Articles in International Peer-Reviewed Journals

[13] 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

[14] 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 (TOPLAS)", August 2016, vol. 38, n° 4, 94 p. [*DOI :* 10.1145/2837022], https://hal.inria.fr/hal-01246534

[15] M.-K. RIVIERE, J.-H. JOURDAN, S. ZOHAR. *dfcomb: An R-package for phase I/II trials of drug combinations*, in "Computer Methods and Programs in Biomedicine", 2016, vol. 125, pp. 117–133 [*DOI :* 10.1016/J.CMPB.2015.10.018], http://hal.upmc.fr/hal-01297367

[16] M.-K. RIVIERE, Y. YUAN, J.-H. JOURDAN, F. DUBOIS, S. ZOHAR. *Phase I/II dose-finding design for molecularly targeted agent: Plateau determination using adaptive randomization*, in "Statistical Methods in Medical Research", March 2016 [*DOI :* 10.1177/0962280216631763], http://hal.upmc.fr/hal-01298681

International Conferences with Proceedings

[17] 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

[18] U. A. ACAR, A. CHARGUÉRAUD, M. RAINEY, F. SIECZKOWSKI. *Dag-calculus: a calculus for parallel computation*, in "Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP)", Nara, Japan, September 2016, pp. 18 - 32 [*DOI :* 10.1145/2951913.2951946], https://hal.inria.fr/hal-01409022

[19] D. AHMAN, C. HRIŢCU, K. MAILLARD, G. MARTÍNEZ, G. PLOTKIN, J. PROTZENKO, A. RASTOGI, N. SWAMY. *Dijkstra Monads for Free*, in "44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)", Unknown, Unknown or Invalid Region, ACM, 2017, pp. 515-529, https://hal.archives-ouvertes.fr/hal-01424794

[20] S. AZAIEZ, D. DOLIGEZ, M. LEMERRE, T. LIBAL, S. MERZ. *Proving Determinacy of the PharOS Real-Time Operating System*, in "Abstract State Machines, Alloy, B, TLA, VDM, and Z - 5th International Conference, ABZ 2016", Linz, Austria, M. J. BUTLER, K.-D. SCHEWE, A. MASHKOOR, M. BIRÓ (editors), LNCS - Lecture Notes in Computer Science, Springer, May 2016, vol. 9675, pp. 70-85 [*DOI :* 10.1007/978-3-319-33600-8_4], https://hal.inria.fr/hal-01322335

[21] 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

[22] S. FLUR, K. E. GRAY, C. PULTE, S. SARKAR, A. SEZGIN, L. MARANGET, W. DEACON, P. SEWELL. *Modelling the ARMv8 Architecture, Operationally: Concurrency and ISA*, in "Principles of Programming Languages 2016 (POPL 2016)", Saint Petersburg, United States, January 2016, https://hal.inria.fr/hal-01244776

[23] 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

[24] J.-H. JOURDAN. *Sparsity Preserving Algorithms for Octagons*, in "NSAD 2016 - Numerical and symbolic abstract domains workshop", Edinburgh, United Kingdom, I. MASTROENI (editor), Elsevier, September 2016, 14 p. , https://hal.inria.fr/hal-01406795

[25] 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, Proceedings of the Twenty-fifth Safety-Critical Systems Symposium, February 2017, https://hal.inria.fr/hal-01399482

[26] F. POTTIER. *Reachability and Error Diagnosis in LR(1) Parsers*, in "CC 2016 - 25th International Conference on Compiler Construction", Barcelone, Spain, Proceedings of the 25th International Conference on Compiler Construction (CC 2016), March 2016, 11 p. [*DOI :* 10.1145/2892208.2892224], https://hal.inria.fr/hal-01417004

[27] 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

[28] R. A. RAGHUNATHAN, S. A. MULLER, U. A. ACAR, G. A. BLELLOCH. *Hierarchical Memory Management for Parallel Programs*, in "Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming", Nara, Japan, September 2016 [*DOI :* 10.1145/3022670.2951935], https://hal.inria.fr/hal-01416237

[29] K. SVENDSEN, F. SIECZKOWSKI, L. BIRKEDAL. *Transfinite Step-Indexing: Decoupling Concrete and Logical Steps*, in "25th European Symposium on Programming Languages and Systems", Eindhoven, Netherlands, December 2016, vol. 9632, pp. 727 - 751 [*DOI :* 10.1007/978-3-662-49498-1_28], https://hal.inria.fr/hal-01408649

[30] B. VAUGON, M. MAUNY. *A Type Inference System Based on Saturation of Subtyping Constraints*, in "Trends in Functional Programming", College Park (MD), United States, June 2016, https://hal.inria.fr/hal-01413043

National Conferences with Proceedings

[31] F. POTTIER. *Reachability and error diagnosis in LR(1) automata*, in "Journées Francophones des Langages Applicatifs", Saint-Malo, France, January 2016, https://hal.inria.fr/hal-01248101

Conferences without Proceedings

[32] Ç. BOZMAN, T. HUFFSCHMITT, M. LAPORTE, F. LE FESSANT. *ocp-lint, A Plugin-based Style-Checker with Semantic Patches*, in "OCaml Users and Developers Workshop 2016", Nara, Japan, September 2016, https://hal.inria.fr/hal-01352013

[33] B. CANOU, G. HENRY, Ç. BOZMAN, F. LE FESSANT. *Learn OCaml, An Online Learning Center for OCaml*, in "OCaml Users and Developers Workshop 2016", Nara, Japan, September 2016, https://hal.inria.fr/hal-01352015

[34] P. COURTIEU, L. RIEG, S. TIXEUIL, X. URBAIN. *A Certified Universal Gathering Algorithm for Oblivious Mobile Robots*, in "Distributed Computing (DISC)", Paris, France, September 2016, http://hal.upmc.fr/hal-01349061

[35] P. COURTIEU, L. RIEG, S. TIXEUIL, X. URBAIN. *Certified Universal Gathering in R2 for Oblivious Mobile Robots*, in "ACM Conference on Principles of Distributed Computing (PODC)", Chicago, United States, ACM, July 2016, http://hal.upmc.fr/hal-01349084

[36] J.-H. JOURDAN. *Statistically profiling memory in OCaml*, in "OCaml 2016", Nara, Japan, September 2016, https://hal.inria.fr/hal-01406809

[37] F. LE FESSANT. *OPAM-builder: Continuous Monitoring of OPAM Repositories*, in "OCaml Users and Developers Workshop 2016", Nara, Japan, September 2016, https://hal.inria.fr/hal-01352008

[38] X. LEROY, S. BLAZY, D. KÄSTNER, B. SCHOMMER, M. PISTER, C. FERDINAND. *CompCert - A Formally Verified Optimizing Compiler*, in "ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress", Toulouse, France, SEE, January 2016, https://hal.inria.fr/hal-01238879

[39] G. SCHERER, L. MARANGET, T. RÉFIS. *Ambiguous pattern variables*, in "OCaml 2016: The OCaml Users and Developers Workshop", Nara, Japan, September 2016, 2 p. , https://hal.inria.fr/hal-01413241

Research Reports

[40] X. LEROY, D. DOLIGEZ, A. FRISCH, J. GARRIGUE, D. RÉMY, J. VOUILLON. *The OCaml system release 4.04: Documentation and user's manual*, Inria, November 2016, https://hal.inria.fr/hal-00930213

[41] X. LEROY. *The CompCert C verified compiler: Documentation and user's manual: Version 2.7*, Inria, June 2016, https://hal.inria.fr/hal-01091802

Other Publications

[42] A. CHARGUÉRAUD, F. POTTIER. *Temporary Read-Only Permissions for Separation Logic*, October 2016, working paper or preprint, https://hal.inria.fr/hal-01408657

References in notes

[43] 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

[44] 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", 2007, pp. 13–22, http://dl.acm.org/citation.cfm?id=1281106

[45] P. HALMAGRAND. *Automated Deduction and Proof Certification for the B Method*, Conservatoire National des Arts et Métiers, December 2016

[46] 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

[47] J. KANG, Y. KIM, C. HUR, D. DREYER, V. VAFEIADIS. *Lightweight verification of separate compilation*, in "Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages", 2016, pp. 178–190, http://doi.acm.org/10.1145/2837614.2837642

[48] 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

[49] 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/

[50] 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

[51] X. LEROY. *Formal certification of a compiler back-end, or: programming a compiler with a proof assistant*, in "33rd ACM symposium on Principles of Programming Languages", ACM Press, 2006, pp. 42–54, http://doi.acm.org/10.1145/1111037.1111042

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

[53] 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

[54] 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

[55] 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

[56] G. SCHERER, D. RÉMY. *Which simple types have a unique inhabitant?*, in "ICFP'15: 20th International Conference on Functional Programming", ACM Press, 2015, pp. 243–255, http://dx.doi.org/10.1145/2784731.2784757