Activity Report 2015

Project-Team GALLIUM

Programming languages, types, compilation and proofs

RESEARCH CENTER
Paris - Rocquencourt

THEME
Proofs and Verification

Table of contents

1. Members **1**
2. Overall Objectives **2**
3. Research Program **2**
3.1. Programming languages: design, formalization, implementation 2
3.2. Type systems 3
3.2.1. Type systems and language design. 3
3.2.2. Polymorphism in type systems. 3
3.2.3. Type inference. 4
3.3. Compilation 4
3.4. Interface with formal methods 5
3.4.1. Software-proof codesign 5
3.4.2. Mechanized specifications and proofs for programming languages components 5
4. Application Domains **5**
4.1. High-assurance software 5
4.2. Software security 6
4.3. Processing of complex structured data 6
4.4. Rapid development 6
4.5. Teaching programming 6
5. Highlights of the Year **7**
6. New Software and Platforms **7**
6.1. CompCert 7
6.2. Diy 7
6.3. Menhir 7
6.4. OCaml 7
6.5. PASL 8
6.6. Zenon 8
7. New Results **8**
7.1. Formal verification of compilers and static analyzers 8
7.1.1. The CompCert formally-verified compiler 8
7.1.2. Formal verification of static analyzers based on abstract interpretation 9
7.1.3. A SPARK Front-end for CompCert 9
7.1.4. Verified JIT compilation of Coq 10
7.2. Language design and type systems 10
7.2.1. Full reduction in the presence of inconsistent assumptions 10
7.2.2. Equivalence and normalization of lambda-terms with sums 10
7.2.3. Types with unique inhabitants for code inference 10
7.2.4. Refactoring with ornaments in ML 10
7.2.5. The Mezzo programming language 11
7.3. Shared-memory parallelism 11
7.3.1. Weak memory models 11
7.3.2. Algorithms and data structures for parallel computing 12
7.4. The OCaml language and system 12
7.4.1. The OCaml system 12
7.4.2. Memory profiling OCaml applications 13
7.4.3. Advanced development tools for OCaml 13
7.4.4. Error diagnosis in Menhir parsers 14
7.4.5. Improvements to Menhir 14
7.5. Software specification and verification 14
7.5.1. Machine-checked proofs of programs, including time complexity 14

7.5.2. Verified property-based random testing 15
7.5.3. Tools for TLA+ 15
7.5.4. Certified distributed algorithms for autonomous mobile robots 15
7.5.5. Contributions to ProofGeneral, an IDE for Coq 16
8. Bilateral Contracts and Grants with Industry **16**
8.1.1. The Caml Consortium 16
8.1.2. Scientific Advisory for OCamlPro 16
9. Partnerships and Cooperations **17**
9.1. National Initiatives 17
9.1.1. ANR projects 17
9.1.1.1. BWare 17
9.1.1.2. Verasco 17
9.1.1.3. Vocal 17
9.1.2. FSN projects 17
9.1.2.1. ADN4SE 17
9.1.2.2. CEEC 18
9.1.3. FUI Projects 18
9.2. European Initiatives 18
9.2.1. FP7 & H2020 Projects 18
9.2.2. ITEA3 Projects 18
9.3. International Initiatives 18
9.4. International Research Visitors 18
10. Dissemination **19**
10.1. Promoting Scientific Activities 19
10.1.1. Scientific events organisation 19
10.1.2. Scientific events selection 19
10.1.2.1. Chair of conference program committees 19
10.1.2.2. Member of conference program committees 19
10.1.2.3. Reviewer 19
10.1.3. Journal 19
10.1.3.1. Member of editorial boards 19
10.1.3.2. Reviewer 19
10.1.4. Leadership within the scientific community 19
10.1.5. Research administration 19
10.2. Teaching - Supervision - Juries 20
10.2.1. Teaching 20
10.2.2. Supervision 20
10.2.3. Juries 20
10.3. Popularization 21
11. Bibliography **21**

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
2.1.11. - Proof languages
2.1.3. - Functional programming
2.1.6. - Concurrent programming
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
2.4.2. - Verification
2.4.3. - Proofs
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
9.4.1. - Computer science

1. Members

Research Scientists

Xavier Leroy [Team leader, Inria, Senior Researcher]
Umut Acar [Carnegie Mellon University, Advanced Research position]
Damien Doligez [Inria, Researcher]
Fabrice Le Fessant [Inria, Researcher]
Luc Maranget [Inria, Researcher]
François Pottier [Inria, Senior Researcher, HdR]
Mike Rainey [Inria, Starting Research position]
Didier Rémy [Inria, Senior Researcher, HdR]

Faculty Member

Pierre Courtieu [CNAM, Associate Professor on délégation]

PhD Students

Vitalii Aksenov [Inria, from Sep 2015]
Arthur Guillon [ENS Cachan, until Jan 2015]
Jacques-Henri Jourdan [Inria, granted by ANR VERASCO project]
Gabriel Scherer [ENS Paris and Inria]
Thomas Williams [ENS Paris]

Post-Doctoral Fellows

Maxime Dénès [Inria, until Sep 2015]
Pietro Abate [Inria]
Filip Sieczkowski [Inria]

Administrative Assistant
Cindy Crossouard [Inria]

Others
Keryan Didier [Inria, Student Intern, from Apr 2015 to Aug 2015]
Benjamin Farinier [Inria, Student Intern, until Aug 2015]
Armaël Guéneau [ENS Lyon, Student Intern, until Jul 2015]

2. Overall Objectives

2.1. Overall Objectives

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 [44]. 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 [42], [39], 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[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 [45] 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

In 2015, Xavier Leroy was appointed Fellow of the ACM "for contributions to safe, high-performance functional programming languages and compilers, and to compiler verification".

Xavier Leroy will receive the 2016 Royal Society Milner Award.

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], Keryan Didier.

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 specification, 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.

- 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. PASL

Participants: Mike 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/

6.6. 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://opam.ocaml.org/packages/zenon/zenon.0.8.0/

7. New Results

7.1. Formal verification of compilers and static analyzers

7.1.1. The CompCert formally-verified compiler

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

In the context of our work on compiler verification (see section 3.3.1), since 2005 we have been developing and formally verifying a moderately-optimizing compiler for a large subset of the C programming language, generating assembly code for the PowerPC, ARM, and x86 architectures [6]. This compiler comprises a back-end, which translates the Cminor intermediate language to PowerPC assembly and is reusable for source languages other than C [5], 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, from 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, we improved the CompCert C compiler in several directions:

- The generation of debugging information in DWARF format was implemented by Bernhard Schommer at AbsInt. Consequently, CompCert-compiled programs can now be debugged using standard debuggers. Xavier Leroy extended the back-end compilation passes and their proofs to propagate debugging information throughout the compilation pipeline.
- The CompCert formal semantics was made more precise in order to increase confidence. We tightened the semantics of pointer comparisons against the null pointer. We formalized the distinction between public and private (`static`) global definitions, and used it to prove the correctness of the "Unusedglob" pass that removes unreferenced private definitions.
- The calling conventions used to pass function arguments and results of `struct` and `union` types were revised in order to comply with the Application Binary Interfaces of the target platforms.
- We added partial support for extended inline assembly, an extension of the C language popularized by the GCC compiler and often used in low-level code.
- Detailed explanations of syntax errors are now produced. This usability feature builds on François Pottier's work on error reporting in LR parsers (see section 7.4.4).
- The PowerPC back-end was extended to support the PowerPC 64-bit extensions and the Freescale E5500 variant.

We released two versions of CompCert, integrating these enhancements: version 2.5 in June and version 2.6 in December. This is the public version of CompCert, available for evaluation and research purposes. In parallel, our industrial partner, AbsInt Angewandte Informatik GmbH, sells a commercial version of CompCert with long-term maintenance.

7.1.2. Formal verification of static analyzers based on abstract interpretation

Participants: Jacques-Henri Jourdan, Xavier Leroy, Sandrine Blazy [team Celtique], Vincent Laporte [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 ANR Verasco 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. The overall architecture and specification of Verasco is described in a paper that was presented at POPL 2015 [19].

This year, Jacques-Henri Jourdan continued the development of this static analyzer, with two goals. First, Jacques-Henri Jourdan improved the precision and analysis time of the existing abstract domains. The existing communication system between domains was instantiated to the cooperation between the abstract domain of intervals and the abstract domain of congruences. Second, Jacques-Henri Jourdan implemented and formalized in our static analyzer the Octagon abstract domain of Miné [46]. This led to new results in the theory behind this abstract domain, allowing Jourdan to use sparse data structures for representing octagons.

7.1.3. 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 from Kansas University. The current front-end is based on this semantics. The compiler has been written and tested, and the proofs of correctness are currently under way.

7.1.4. Verified JIT compilation of Coq

Participants: Maxime Dénès, Xavier Leroy.

Last year, we started the Coqonut project, whose objective is to develop and formally verify an efficient, compiled implementation of Coq's reduction. This year, we made progress on this verification effort:

- We ported our OCaml prototype to Coq and started its verification, notably of the first phase of the compiler which involves uncurrying, using untyped step-indexed logical relations.
- We adapted (part of) the Coq x86 macro assembler by Andrew Kennedy, Nick Benton, Jonas B. Jensen and Pierre-Evariste Dagand to x86-64. This macro assembler framework is used in Coqonut's backend to generate assembly or machine code.

7.2. Language design and type systems

7.2.1. Full reduction in the presence of inconsistent assumptions

Participants: Didier Rémy, Gabriel Scherer.

Gabriel Scherer and Didier Rémy continued their work on assumption hiding and presented it at ESOP 2015 [22]. This work aims at restoring confluence when mixing full and weak reduction and providing a continuum between consistent and inconsistent abstraction. Assumption hiding supports fine-grained control of dependencies between computations and the logical hypotheses they depend on. Although studied for a language of coercions, the solution is more general and should be applicable to any language with abstraction over propositions that are left implicit, either for the user's convenience in a surface language or because they have been erased prior to computation in an internal language.

7.2.2. Equivalence and normalization of lambda-terms with sums

Participants: Gabriel Scherer, Guillaume Munch-Maccagnoni [Université Paris-Diderot, laboratoire PPS].

Gabriel Scherer presented at TLCA 2015 his work on understanding equivalence of sum types using the proof-theoretical technique of focusing [24]. Independently, his collaboration with Guillaume Munch-Maccagnoni resulted in a presentation of sum equivalence using an abstract machine calculus [33]. This approach allows for a more concise and cleaner definition of the equivalence relation, and a finer-grained understanding of the role of purity assumptions in the program equivalence relation.

7.2.3. Types with unique inhabitants for code inference

Participants: Gabriel Scherer, Didier Rémy.

Gabriel Scherer and Didier Rémy presented at ICFP 2015 [23] an algorithm to decide whether a type has a unique inhabitant in the simply-typed lambda-calculus with sum types. This algorithm comes along with a prototype implementation. This minimal setting is not representative of the expressiveness of realistic programming languages, but already covers a first few interesting code inference scenarios for polymorphic libraries in functional languages with prenex polymorphism: for instance, we can infer the "bind" function of the exception monad.

7.2.4. 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 previously described an algorithm to perform this lifting in ML. This description was informal. This year, we improved this algorithm by decomposing it in several steps and we formalized it. 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 also studied the properties of lifting, particularly the preservation of complexity and effects, with the aim of characterizing more precisely the syntactic liftings that can be produced by our algorithm.

On the practical side, our prototype ornamentation tool has been improved with an implementation of ornament inference. The generalized program gives a description of all possible extension points that must be filled by providing patches. In practice, a few heuristics are enough to automate most of the patching work. The rest can be filled interactively by the programmer. In the case of refactoring (the representation of a data type is modified without adding any data), the transformation is fully automatic.

7.2.5. The Mezzo programming language

Participants: Thibaut Balabonski [Université Paris Sud], François Pottier, Jonathan Protzenko.

Mezzo is a programming language proposal whose untyped foundation is very much like OCaml (i.e., it is equipped with higher-order functions, algebraic data structures, mutable state, and shared-memory concurrency) and whose type system offers flexible means of describing ownership policies and controlling side effects.

A comprehensive paper, which contains both a tutorial introduction to Mezzo and a description of its formal definition and proof, was submitted to TOPLAS in 2014. This year, after a round of reviewing, it was revised and accepted for publication [11]. A reflection on the design of Mezzo was presented at SNAPL 2015 [21].

7.3. Shared-memory parallelism

7.3.1. Weak memory models

Participants: Luc Maranget, Jade Alglave [Microsoft Research, Cambridge], Patrick Cousot [New York University], Keryan Didier.

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 2015, Luc Maranget collaborated with Jade Alglave and Patrick Cousot to extend "Cats", a domain-specific language for defining and executing weak memory models. A precise semantics for "Cats" is the core of a submitted journal article that also includes a study and formalisation of the HSA memory model — the Heterogeneous System Architecture foundation is an industry standards body targeting heterogeneous computing devices (see http://www.hsafoundation.com/). The new extensions of the Cats language have been integrated in the released version of the **diy** tool suite (see section 6.2).

Luc Maranget also co-authored a paper that will be presented at POPL 2016 [18]. This work describes an operational semantics for the new generation ARM processors. It is joint work with many researchers, including S. Flur and other members of P. Sewell's team (University of Cambridge) and W. Deacon (ARM Ltd).

During his M2 internship, supervised by Luc Maranget, Keryan Didier significantly improved the **diy** tool suite, in particular by writing front-ends for ARMv8 and for a subset of the C language. Keryan Didier also wrote a new (as yet unreleased) tool to translate between various input languages, in particular from machine assemblers to generic assembler and back.

7.3.2. Algorithms and data structures for parallel computing

Participants: Umut Acar, Vitalii Aksenov, Arthur Charguéraud, Mike 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 three major results this year.

Our result on the development of fast and robust parallel graph traversal algorithms based on depth-first-search has been presented at the ACM/IEEE Conference on High Performance Computing [15]. This algorithm leverages a new sequence data structure for representing the set of edges remaining to be visited. In particular, it uses a balanced split operation for partitioning the edges of a graph among the processors involved in the computation. Compared with prior work, the new algorithm is designed to be efficient not just for particular classes of graphs, but for all input graphs.

Our second 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 has been submitted for publication.

Our third result concerns the development of parallel dynamic algorithms. This year, we started developing a parallel dynamic algorithm for tree computations. The algorithm is dynamic in the sense that it admits changes to the underlying tree in the form of insertions and deletions of edges and vertices and updates the computation by doing total work that is linear in the size of the changes, but only logarithmic in the size of the tree. The algorithm is parallel in the sense that the updates take place in parallel. Parallel algorithms have been studied extensively in the past, but few of these are dynamic. Similarly, dynamic algorithms have also been studied extensively in the past, but few of these are parallel. Our work thus explores what in retrospect seems like an obvious gap in the literature. A paper describing this work is in preparation.

7.4. The OCaml language and system

7.4.1. The OCaml system

Participants: Damien Doligez, Alain Frisch [Lexifi SAS], Jacques Garrigue [University of Nagoya], 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 4.02.2 and 4.02.3 of the OCaml system. These are minor releases that fix about 100 bugs and implement 12 minor new features, including support for nonrecursive type definitions and a higher-level interface with documentation generation tools.

Most of our activity was devoted to preparing the next major release of OCaml, version 4.03.0, which is expected in the first quarter of 2016. The novelties we worked on include:

- Inline record types as arguments to constructors of sum types, combining the clarity and extensibility brought by named record fields with the compact in-memory representation of unnamed constructor arguments.
- Improved redudancy and exhaustiveness checks for pattern-matching over generalized algebraic data types (GADTs) [41].

- Improved unboxing optimizations for numbers, including the ability to mark arguments and results of external C functions as unboxed.
- The garbage collector was made more incremental, so as to reduce the worst-case GC pause times.
- The native-code compiler was ported to two new architectures: PowerPC 64 bits (including IBM's new little-endian variant) and IBM zSystems.

On the organization side, we switched to Github as the central repository for the OCaml development sources. Github facilitates collaborative work among the growing community of contributors to the OCaml code base. In 2015, more than 100 contributors proposed small or large improvements to the OCaml compiler distribution.

7.4.2. Memory profiling OCaml applications

Participants: Fabrice Le Fessant, Çagdas Bozman [OCamlPro], Albin Coquereau [OCamlPro].

Most modern languages make use of automatic memory management to discharge the programmer from the burden of explicitly allocating and releasing chunks of memory. As a consequence, when an application exhibits an unexpected usage of memory, programmers need new tools to understand what is happening and how to solve such an issue. In OCaml, the compact representation of values, with almost no runtime type information, makes the design of such tools more complex.

In the past, we have experimented with different tools to profile the memory usage of real OCaml applications, in particular one that saves snapshots of the heap after every garbage collection. Snapshots can then be analysed to display the evolution of memory usage, with detailed information on the types of values, where they were allocated and from where they are still reachable.

This year, we experimented in three new directions, mostly driven by the size of the snapshots to be analysed:

- We studied several ways of displaying snapshots. Because of the large amount of information contained in a snapshot, it is hard for a typical user to find what he or she is looking for. We tried multiple filtering methods, based on graph algorithms, to remove the least significant information from the reports given to the user.
- We experimented with new algorithms to compress and analyse *huge* memory snapshots, i.e., snapshots that are too big to fit in the computer's memory. Indeed, standard analyses on snapshots bigger than the available memory are too long to run in practice because of random disk accesses. Thus, we tried several compression methods for snapshots and graph-reduced them to fit in memory, without losing any information, reaching a 50x speedup in complete analysis time.
- We implemented a new graph algorithm to merge sets of blocks in memory by the sets of roots they are reachable from. Such a computation was heretofore supposed to be untractable in practice, but could actually be computed in our case on huge compressed snapshots in reasonable time.

7.4.3. Advanced development tools for OCaml

Participants: Fabrice Le Fessant, Pierre Chambart [OCamlPro], Michael Laporte [OCamlPro].

In order to promote the use of OCaml in industrial contexts, we have worked on improving the tools that accompany OCaml:

- We developed the first prototype of a native debugger for OCaml, based on the LLDB debugging framework on top of LLVM. For that, we first generated a full OCaml binding for the LLDB library, by parsing the C++ headers of the libraries and automatically generating OCaml and C++ stubs. We were then able to use the OCaml binding to develop several tools, ranging from a simple tool that displays the internal GC information of a finished OCaml application, to an almost complete debugger, which displays OCaml values using runtime type information added for memory profiling.
- We also developed a new profiling framework for OCaml, called *operf*. The framework is composed of two tools: *operf-micro* can be used to run micro-benchmarks directly from inside modified OCaml compiler sources, while the *operf-macro* tool can be used to evaluate the impact of a new compiler optimization on a large set of OPAM packages.
- Finally, we came up with new ideas for *ocp-build*, a generic building tool with OCaml-specific support, to improve the expressiveness of its package description language and to easily describe cross-compilation of OCaml packages.

7.4.4. Error diagnosis in Menhir parsers

Participant: François Pottier.

LR parsers are powerful and efficient, but traditionally have done a poor job of explaining syntax errors. Although it is easy to report where an error was detected, it seems difficult to explain what has been understood so far and what is expected next. The OCaml and CompCert compilers, until now, have offered little information to the user beyond the traditional "syntax error" message.

In 2003, Jeffery proposed associating a fixed diagnostic message with every state of the LR automaton (therefore ignoring the automaton's stack). This simple approach may seem tempting. However, a typical automaton has hundreds or thousands of states. Not all of them can trigger an error, but it is difficult to tell which can, and which cannot. Furthermore, for certain states, it is difficult (or even impossible) to write an accurate diagnostic message, because some vital contextual information resides in the stack, which Jeffery's method cannot access.

In 2015, François Pottier proposed a reachability algorithm for LR automata, which he implemented in the Menhir parser generator (see section 6.3). This algorithm allows finding out which states can trigger an error and (therefore) require writing a diagnostic message. Furthermore, Pottier proposed two mechanisms for influencing where errors are detected. If used appropriately, these mechanisms make it easier (or possible) to write an accurate diagnostic message.

Pottier applied this approach to the C grammar in the front-end of the CompCert compiler, therefore allowing CompCert to produce better diagnostic messages when a C program is syntactically incorrect.

A short paper describing this work will be presented at JFLA 2016 [29]. A longer paper is in submission.

7.4.5. Improvements to Menhir

Participants: Frédéric Bour [independent consultant], Jacques-Henri Jourdan, François Pottier, Yann Régis-Gianas [team πr^2], Gabriel Scherer.

In 2015, The Menhir parser generator (see section 6.3) was extended with many new features, several of which originated in the Merlin IDE for OCaml and were ported back into Menhir.

- The parsers generated by Menhir are now incremental: they can be stopped and resumed at any point, at essentially no cost. This is exploited in Merlin, where the text is re-parsed after every keystroke.
- The state of the parser can be inspected by the user. This allows building custom libraries, outside Menhir, for error diagnosis, error recovery, etc. This is exploited in Merlin, where a valid abstract syntax tree is built (and passed to the OCaml type-checker) even if the text contains syntax errors.
- A reachability algorithm has been implemented (see section 7.4.4). It allows finding out which states can trigger an error and (therefore) require a diagnostic message to be written. It is accompanied with several tools that help maintain the database of diagnostic messages as the grammar evolves.
- Compatibility with `ocamlyacc` has been improved, in particular insofar as the computation of locations is concerned. This should help port the OCaml parser from `ocamlyacc` to Menhir, a transition that we envision making in the near future. This should help improve the quality of OCaml's syntax error messages.

7.5. Software specification and verification

7.5.1. Machine-checked proofs of programs, including time complexity

Participants: Arthur Charguéraud, Armaël Guéneau, François Pottier.

In a security-critical setting, it is important to prove that a program is correct, and to do so formally, that is, via a machine-checked proof. It is also important, one may argue, to prove that the program does not require more resources than expected (where a "resource" may be time, memory space, disk space, network bandwidth, etc.). Otherwise, even though the program is "correct" in theory, it may turn out to be unusable in practice.

Separation Logic, extended with the notion of a "time credit", a permission to perform one step of computation, allows reasoning about the correctness and the (amortized) time complexity of a program. Using this approach, which Charguéraud implemented in the CFML tool, Charguéraud and Pottier produced a machine-checked proof of the correctness and time complexity of a Union-Find data structure, implemented as an OCaml module. This demonstrates that this approach scales up to difficult complexity analyses and down to the level of actual executable code (as opposed to pseudo-code). This work was presented at ITP 2015 [17].

During his M2 internship, Armaël Guéneau extended this approach so as to allow working conveniently with the big-O notation. He extended the CFML library and verified the time complexity of a binary random access list data structure due to Okasaki. This work has not been published yet.

7.5.2. Verified property-based random testing

Participants: Zoe Paraskevopoulou [ENS Cachan, team Prosecco], Cătălin Hriţcu [team Prosecco], Maxime Dénès, Leonidas Lampropoulos [U. of Pennsylvania], Benjamin C. Pierce [U. of Pennsylvania].

Property-based random testing has been popularized in the functional programming community by tools like QuickCheck. Its integration with a proof assistant creates an interesting opportunity: reusable or tricky testing code can be formally verified using the proof assistant itself.

We introduced a novel methodology for formally verified property-based testing and implemented it as a foundational verification framework for QuickChick, a port of QuickCheck to Coq. Our framework enables one to verify that the executable testing code is testing the right Coq property. To make verification tractable, we provided a systematic way for reasoning about the set of outcomes a random data generator can produce with non-zero probability, while abstracting away from the actual probabilities.

We also applied this methodology to a complex case study on testing an information-flow control abstract machine, demonstrating that our verification methodology is modular and scalable and that it requires minimal changes to existing code.

Maxime Dénès more specifically contributed to the development of the QuickChick Coq plug-in, to the development of Coq libraries for reasoning on the set of outcomes of random generators and to the verification of QuickChick's combinator library.

This work was presented at ITP 2015 [20].

7.5.3. Tools for 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 [43], and to build tools for writing TLA+ specifications and mechanically checking the proofs.

This year, we released version 1.4.3 of the TLA+ Proof System (TLAPS) [40], the part of the TLA+ tools that handles mechanical checking of TLA+ proofs.

This was the last year of the ADN4SE project, which develops tools for rapid development of real-time software based on the PharOS real-time kernel developed by CEA. Within this project we built, in collaboration with CEA, a formal proof of determinacy of the message-passing subsystem of PharOS. We used this experience to improve our TLA+ tools and libraries.

We have started a rewrite of TLAPS from scratch, which will make it possible to handle all aspects of the TLA+ language, including temporal formulas and their proofs.

7.5.4. Certified distributed algorithms for autonomous mobile robots

Participants: Pierre Courtieu, Xavier Urbain [ENSIIE], Sébastien Tixeuil [U. Pierre et Marie Curie], Lionel Rieg [Collège de France].

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.

We are developing a Coq-based verification platform for distributed algorithms for autonomous robots. This year, we mechanically proved and slightly generalized a non-trivial proof of impossibility of such an algorithm under certain hypotheses [14]. We also proved several algorithms in the literature, demonstrating the viability of the platform [13].

7.5.5. Contributions to ProofGeneral, an IDE for Coq

Participant: Pierre Courtieu.

User interface is a crucial issue for theorem provers like Coq. ProofGeneral [38], an emacs-based prover interface, is widely used among Coq users. In addition to synchronizing with the evolutions of Coq itself, we contributed many improvements to ProofGeneral during the past year, among which: a better debugging mode and message printing, user assistance for naming hypotheses and indenting proof scripts, and more.

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 12 member companies:

- Aesthetic Integration
- Bloomberg
- CEA
- Citrix
- Dassault Aviation
- Dassault Systèmes
- Esterel Technologies
- Jane Street
- 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.

9.1.2.2. CEEC

Participants: Maxime Dénès, Xavier Leroy.

The "CEEC" project (2011–2015) is coordinated by the Prove & Run company and also involves Esterel Technologies and Trusted Labs. It is funded by the *Briques Génériques du Logiciel Embarqué* programme of *Fonds national pour la Société Numérique*. The CEEC project develops an environment for the development and certification of high-security software, centered on a new domain-specific language designed by Prove & Run. Our involvement in this project focuses on the formal verification of a C code generator for this domain-specific language, and its interface with the CompCert C verified compiler.

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, Mike 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.

9.4. International Research Visitors

9.4.1. Visits of International Scientists

9.4.1.1. Research stays abroad

From November 2014 to June 2015, Damien Doligez was on a sabbatical at Jane Street (New York, USA), a financial company (a member of the Caml Consortium) that invests considerable R&D in the OCaml language and system.

10. Dissemination

10.1. Promoting Scientific Activities

10.1.1. Scientific events organisation

10.1.1.1. Member of organizing committees

Didier Rémy is a member of the steering committees of the OCaml workshop and the ML Family workshop.

10.1.2. Scientific events selection

10.1.2.1. Chair of conference program committees

Arthur Charguéraud served as program committee chair for for the second International Workshop on Coq for Programming Languages (CoqPL 2016).

Damien Doligez served as program committee chair for the OCaml Users and Developers workshop (OUD 2015).

10.1.2.2. Member of conference program committees

Umut Acar was a member of the program committees of the 30th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2016) and of the workshop on Algorithms and Systems for MapReduce and Beyond (BeyondMR 2016). He was also a member of the External Review Committee of the 43rd ACM Symposium on Principles of Programming Languages (POPL 2016).

Xavier Leroy was a member of the program committees of the 1st Summit on Advances in Programming Languages (SNAPL 2015) and of the Compiler Construction conference (CC 2016).

Xavier Leroy was a member of the award committee for the 2015 ACM SIGPLAN Software System award.

Mike Rainey was a member of the program committee of the International Conference on Functional Programming (ICFP 2015).

Arthur Charguéraud and Didier Rémy were members of the program committee of the European Symposium on Programming Languages (ESOP 2016).

10.1.2.3. Reviewer

In 2015, the members of Gallium reviewed at least 80 conference submissions.

10.1.3. Journal

10.1.3.1. Member of 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 a member of the editorial board of the Journal of Functional Programming.

10.1.3.2. Reviewer

In 2015, the members of Gallium reviewed at least 4 journal submissions and 10 grant proposals.

10.1.4. Leadership within the scientific community

Pierre Courtieu and Xavier Leroy are members of the Coq steering commitee.

10.1.5. Research administration

Xavier Leroy is *délégué scientifique adjoint* of Inria Paris-Rocquencourt and an appointed member of Inria's *Commission d'Évaluation*. In 2015, he participated in the following Inria hiring and promotion committees: *jury d'admissibilité CR2 Paris-Rocquencourt* (vice-chair); *jury d'admissibilité CR2 Nancy*; and *promotions CR1*. He was a member of the hiring committee for a professor position at ENS Rennes.

Luc Maranget is an elected member of the *Comité Technique Inria*.

Luc Maranget chairs the *Commission des utilisateurs des moyens informatiques – Recherche* of Inria Paris-Rocquencourt.

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 represents Inria in the *commission des études* of the MPRI master, co-organized by U. Paris Diderot, ENS Cachan, ENS Paris, and École Polytechnique.

Didier Rémy is Inria's *Deputy Scientific Director* (ADS) in charge of *Algorithmics, Programming, Software and Architecture*.

10.2. Teaching - Supervision - Juries

10.2.1. Teaching

Master: Xavier Leroy and Didier Rémy, "Functional programming languages", 45h, 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.

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

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

Master: François Pottier, "Programming with permissions in Mezzo", 4.5h, M2, 15th SFM summer school, Bertinoro, Italy.

Licence: Mike 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.

10.2.2. Supervision

M2 (Master Pro): Keryan Didier, Université Denis Diderot, supervised by Luc Maranget.

M2 (MPRI): Benjamin Farinier, Université Paris Diderot, supervised by François Pottier.

M2 (MPRI): Armaël Guéneau, ENS Lyon, supervised by Arthur Charguéraud and 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: Jacques-Henri Jourdan, "Verasco: a formally verified C static analyzer", Université Paris Diderot, since September 2012, supervised by Xavier Leroy.

PhD in progress: Gabriel Scherer, "Which types have a unique inhabitant?", Université Paris Diderot, since October 2011, supervised by Didier Rémy.

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

10.2.3. Juries

Pierre Courtieu was a member of the Ph.D. jury of Sylvain Dailler, Université d'Orléans, December 2015.

Xavier Leroy was a member of the PhD juries of Tie Cheng, ENS Paris, September 2015; Vincent Laporte, Université Rennes 1, November 2015; and Robbert Krebbers, Radboud University, December 2015.

François Pottier was president of the Ph.D. jury of Bruno Bernardo, École Polytechnique, September 2015.

10.3. Popularization

Jacques-Henri Jourdan is involved in the organization of the Junior Seminar of Inria Paris-Rocquencourt.

Jacques-Henri Jourdan and Fabrice Le Fessant manned a stand at “Salon Culture & Jeux Mathématiques” in Paris.

Fabrice Le Fessant is one of the main organizers of the OCaml Meetup in Paris.

Xavier Leroy gave a popularization talk on critical software and its formal verification at the computer science colloquium of University Pierre et Marie Curie.

Since 2012, the Gallium team has published a research blog at http://gallium.inria.fr/blog/, edited by Gabriel Scherer. This blog continued its activity in 2015, with 10 posts from 8 different authors.

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] 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, http://dx.doi.org/10.1145/2603088.2603128

[4] 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

[5] 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

[6] 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

[7] 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

[8] 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

[9] 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

[10] J.-B. TRISTAN, X. LEROY. *A simple, verified validator for software pipelining*, in "Proceedings of the 37th ACM Symposium on Principles of Programming Languages (POPL'10)", ACM Press, 2010, pp. 83–92, http://doi.acm.org/10.1145/1706299.1706311

Publications of the year

Articles in International Peer-Reviewed Journals

[11] 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)", 2016 [*DOI :* 10.1145/2837022], https://hal.inria.fr/hal-01246534

[12] S. BOLDO, J.-H. JOURDAN, X. LEROY, G. MELQUIOND. *Verified Compilation of Floating-Point Computations*, in "Journal of Automated Reasoning", February 2015, vol. 54, n° 2, pp. 135-163 [*DOI :* 10.1007/s10817-014-9317-x], https://hal.inria.fr/hal-00862689

[13] B. BÉRARD, P. COURTIEU, L. MILLET, M. POTOP-BUTUCARU, L. RIEG, N. SZNAJDER, S. TIXEUIL, X. URBAIN. *[Invited Paper] Formal Methods for Mobile Robots: Current Results and Open Problems*, in "International Journal of Informatics Society", 2015, vol. 7, n° 3, pp. 101-114, http://hal.upmc.fr/hal-01238784

[14] P. COURTIEU, L. RIEG, S. TIXEUIL, X. URBAIN. *Impossibility of gathering, a certification*, in "Information Processing Letters", March 2015, vol. 115, n° 3, pp. 447-452 [*DOI :* 10.1016/j.ipl.2014.11.001], http://hal.upmc.fr/hal-01122869

International Conferences with Proceedings

[15] U. A. ACAR, A. CHARGUÉRAUD, M. RAINEY. *A Work-Efficient Algorithm for Parallel Unordered Depth-First Search*, in "Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis", Austin, Texas, United States, November 2015 [*DOI :* 10.1145/2807591.2807651], https://hal.inria.fr/hal-01245837

[16] P. BHATOTIA, P. FONSECA, U. A. ACAR, B. BJÖRN, R. RODRIGUES. *iThreads: A Threading Library for Parallel Incremental Computation*, in "Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems", Istanbul, Turkey, ACM, March 2015, pp. 645–659 [*DOI :* 10.1145/2694344.2694371], https://hal.inria.fr/hal-01245884

[17] A. CHARGUÉRAUD, F. POTTIER. *Machine-Checked Verification of the Correctness and Amortized Complexity of an Efficient Union-Find Implementation*, in "6th International Conference on Interactive Theorem Proving (ITP)", Nanjing, China, August 2015 [*DOI :* 10.1007/978-3-319-22102-1_9], https://hal.inria.fr/hal-01245872

[18] 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

[19] J.-H. JOURDAN, V. LAPORTE, S. BLAZY, X. LEROY, D. PICHARDIE. *A formally-verified C static analyzer*, in "POPL 2015: 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages", Mumbai, India, ACM, January 2015, pp. 247-259 [*DOI :* 10.1145/2676726.2676966], https://hal.inria.fr/hal-01078386

[20] Z. PARASKEVOPOULOU, C. HRIŢCU, M. DÉNÈS, L. LAMPROPOULOS, B. C. PIERCE. *Foundational Property-Based Testing*, in "ITP 2015 - 6th conference on Interactive Theorem Proving", Nanjing, China, Lecture Notes in Computer Science, Springer, August 2015, vol. 9236 [*DOI :* 10.1007/978-3-319-22102-1_22], https://hal.inria.fr/hal-01162898

[21] F. POTTIER, J. PROTZENKO. *A few lessons from the Mezzo project*, in "Summit oN Advances in Programming Languages (SNAPL)", Asilomar, United States, Leibniz International Proceedings in Informatics, May 2015, vol. 32 [*DOI :* 10.4230/LIPICS.SNAPL.2015.221], https://hal.inria.fr/hal-01246360

[22] G. SCHERER, D. RÉMY. *Full reduction in the face of absurdity*, in "ESOP'2015: European Conference on Programming Languages and Systems", London, United Kingdom, April 2015, https://hal.inria.fr/hal-01095390

[23] G. SCHERER, D. RÉMY. *Which simple types have a unique inhabitant?*, in "The 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)", Vancouver, Canada, August 2015, https://hal.inria.fr/hal-01235596

[24] G. SCHERER. *Multi-focusing on extensional rewriting with sums*, in "Typed Lambda Calculi and Applications", Warsaw, Poland, June 2015, https://hal.inria.fr/hal-01235372

[25] E. ÇIÇEK, D. GARG, U. ACAR. *Refinement Types for Incremental Computational Complexity*, in "24th European Symposium on Programming (ESOP)", London, United Kingdom, April 2015, vol. 9032, pp. 406-431 [*DOI :* 10.1007/978-3-662-46669-8_17], https://hal.inria.fr/hal-01245888

National Conferences with Proceedings

[26] Ç. BOZMAN, G. HENRY, M. IGUERNELALA, F. LE FESSANT, M. MAUNY. *ocp-memprof: un profileur mémoire pour OCaml*, in "Vingt-sixièmes Journées Francophones des Langages Applicatifs (JFLA 2015)", Le Val d'Ajol, France, D. BAELDE, J. ALGLAVE (editors), January 2015, https://hal.inria.fr/hal-01099134

[27] P.-É. DAGAND, G. SCHERER. *Normalization by realizability also evaluates*, in "Vingt-sixièmes Journées Francophones des Langages Applicatifs (JFLA 2015)", Le Val d'Ajol, France, D. BAELDE, J. ALGLAVE (editors), January 2015, https://hal.inria.fr/hal-01099138

[28] F. POTTIER. *Depth-First Search and Strong Connectivity in Coq*, in "Vingt-sixièmes journées francophones des langages applicatifs (JFLA 2015)", Le Val d'Ajol, France, D. BAELDE, J. ALGLAVE (editors), January 2015, https://hal.inria.fr/hal-01096354

[29] 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

[30] G. BURY, D. DELAHAYE, D. DOLIGEZ, P. HALMAGRAND, O. HERMANT. *Automated Deduction in the B Set Theory using Typed Proof Search and Deduction Modulo*, in "LPAR 20 : 20th International Conference on Logic for Programming, Artificial Intelligence and Reasoning", Suva, Fiji, November 2015, https://hal-mines-paristech.archives-ouvertes.fr/hal-01204701

[31] P. CHAMBART, M. LAPORTE, V. BERNARDOFF, F. LE FESSANT. *Operf: Benchmarking the OCaml Compiler*, in "OCaml Users and Developers Workshop", Vancouver, Canada, September 2015, https://hal.inria.fr/hal-01245844

[32] 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

[33] G. MUNCH-MACCAGNONI, G. SCHERER. *Polarised Intermediate Representation of Lambda Calculus with Sums*, in "Thirtieth Annual ACM/IEEE Symposium on Logic In Computer Science (LICS 2015)", Kyoto, Japan, July 2015, Dec. 2015: see the added footnote on page 7 [*DOI :* 10.1109/LICS.2015.22], https://hal.inria.fr/hal-01160579

Scientific Books (or Scientific Book chapters)

[34] *CPP '15: Proceedings of the 2015 Conference on Certified Programs and Proofs*, ACM, Mumbai, India, January 2015, 184 p. , https://hal.inria.fr/hal-01101937

Research Reports

[35] U. A. ACAR, A. CHARGUÉRAUD, M. RAINEY. *Fast Parallel Graph-Search with Splittable and Catenable Frontiers*, Inria, January 2015, https://hal.inria.fr/hal-01089125

[36] P. COURTIEU, L. RIEG, S. TIXEUIL, X. URBAIN. *A Certified Universal Gathering Algorithm for Oblivious Mobile Robots*, UPMC, Sorbonne Universites CNRS ; CNAM, Paris ; College de France ; Université Paris Sud, June 2015, http://hal.upmc.fr/hal-01159890

[37] X. LEROY. *The CompCert C verified compiler: Documentation and user's manual*, Inria, December 2015, https://hal.inria.fr/hal-01091802

References in notes

[38] D. ASPINALL. *Proof General: A Generic Tool for Proof Development*, in "Tools and Algorithms for the Construction and Analysis of Systems", S. GRAF, M. SCHWARTZBACH (editors), Lecture Notes in Computer Science, Springer, 2000, vol. 1785, pp. 38–43, http://dx.doi.org/10.1007/3-540-46419-0_3

[39] 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

[40] D. COUSINEAU, D. DOLIGEZ, L. LAMPORT, S. MERZ, D. RICKETTS, H. VANZETTO. *TLA + Proofs*, in "FM 2012: Formal Methods - 18th International Symposium", D. GIANNAKOPOULOU, D. MÉRY (editors), Lecture Notes in Computer Science, Springer, 2012, vol. 7436, pp. 147-154, http://dx.doi.org/10.1007/978-3-642-32759-9_14

[41] J. GARRIGUE, J. LE NORMAND. *GADTs and exhaustiveness: looking for the impossible*, in "ACM SIGPLAN ML Family Workshop", ACM, 2015, http://www.math.nagoya-u.ac.jp/~garrigue/papers/gadtspm.pdf

[42] 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

[43] 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

[44] 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/

[45] 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

[46] A. MINÉ. *Weakly relational numerical abstract domains*, École Polytechnique, December 2004, https://www-apr.lip6.fr/~mine/these/these-color.pdf

[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