Activity Report 2014

Project-Team GALLIUM

Programming languages, types, compilation and proofs

RESEARCH CENTER
Paris - Rocquencourt

THEME
Proofs and Verification

Table of contents

Project-Team GALLIUM

Keywords: Programming Languages, Functional Programming, Compilation, Type Systems, Proofs Of Programs, Static Analysis, Parallelism

Creation of the Project-Team: 2006 May 01.

1. Members

Research Scientists

Xavier Leroy [Team leader, Inria, Senior Researcher]
Umut Acar [Inria, Advanced Research position, ERC DeepSea]
Damien Doligez [Inria, Researcher, on sabbatical since Nov 2014]
Fabrice Le Fessant [Inria, Researcher]
Luc Maranget [Inria, Researcher]
François Pottier [Inria, Senior Researcher, HdR]
Mike Rainey [Inria, Starting Research position, ERC DeepSea]
Didier Rémy [Inria, Senior Researcher, HdR]

Faculty Member

Pierre Courtieu [CNAM, Associate Professor on délégation, from Sep 2014]

Engineer

Michael Laporte [Inria, until Nov 2014, granted by OSEO, project FUI Richelieu]

PhD Students

Julien Cretin [U. Paris Diderot, until Jan 2014, AMX grant]
Arthur Guillon [ENS Cachan, since Sep 2014]
Jacques-Henri Jourdan [Inria, granted by ANR, project VERASCO]
Jonathan Protzenko [Inria, until Sep 2014]
Gabriel Scherer [U. Paris Diderot, AMN grant]
Thomas Williams [ENS Paris, since Sep 2014]

Post-Doctoral Fellows

Thibaut Balabonski [Inria, until Feb 2014]
Thomas Braibant [Inria, until Mar 2014, granted by CDC, project FSN CEEC]
Pierre-Évariste Dagand [Inria, until Sep 2014]
Jael Kriener [Inria, until Sep 2014, granted by CDC, project FSN ADN4SE]
Maxime Dénès [Inria, since Oct 2014, granted by CDC, project FSN CEEC]

Visiting Scientist

Sigurd Schneider [Ph.D. student, Saarlandes U., from Mar 2014 until May 2014]

Administrative Assistant

Cindy Crossouard [Inria]

Others

Pierrick Couderc [intern, from Apr 2014 to Sep 2014]
Jacques-Pascal Deplaix [intern, EPITECH, until May 2014]
Ali-Firat Kilic [intern, from Jun 2014 until Sep 2014]
Le Thanh Dung Nguyen [intern, ENS Paris, from Oct 2014]
Julien Sagot [intern, from Jun 2014 until Sep 2014]
Thomas Williams [intern, from Apr 2014 to Sep 2014]

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 Caml 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 accesses, etc) to the 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 around 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 [48]. 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 [65] 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 analyze 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 [59], [53], 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 [69], 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 [66].

Finally, the notion of "coercion polymorphism" proposed by Cretin and Rémy [28] 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 typechecker? 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 typechecking 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 designs 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 refer collectively 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 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 Caml 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 Caml 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 references, 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 Caml 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 [62] and enforcement of data confidentiality through type-based inference of information flows and noninterference properties [67].

4.3. Processing of complex structured data

Like most functional languages, Caml 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, Caml 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 Caml 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 Caml 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 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. New Software and Platforms

5.1. OCaml

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

OCaml, formerly known as Objective Caml, is our flagship implementation of the Caml language. From a language standpoint, it extends the core Caml language with a fully-fledged object and class layer, as well as a powerful module system, all joined together by a sound, polymorphic type system featuring type inference. The OCaml system is an industrial-strength implementation of this language, featuring a high-performance native-code compiler for several processor architectures (IA32, AMD64, PowerPC, ARM, ARM64) as well as a bytecode compiler and interactive loop for quick development and portability. The OCaml distribution includes a standard library and a number of programming tools: replay debugger, lexer and parser generators, documentation generator, and compilation manager.

Web site: http://caml.inria.fr/

5.2. CompCert C

Participants: Xavier Leroy [correspondant], Sandrine Blazy [EPI Celtique], Jacques-Henri Jourdan.

The CompCert C verified compiler is a compiler for a large subset of the C programming language that generates code for the PowerPC, ARM and x86 processors. The distinguishing feature of Compcert is that it has been formally verified using the Coq proof assistant: the generated assembly code is formally guaranteed to behave as prescribed by the semantics of the source C code. The subset of C supported is quite large, including all C types except `long double`, all C operators, almost all control structures (the only exception is unstructured `switch`), and the full power of functions (including function pointers and recursive functions but not variadic functions). The generated PowerPC code runs 2–3 times faster than that generated by GCC without optimizations, and only 7% (resp. 12%) slower than GCC at optimization level 1 (resp. 2).

Web site: http://compcert.inria.fr/

5.3. The diy tool suite

Participants: Luc Maranget [correspondant], Jade Alglave [Microsoft Research, Cambridge], Jacques-Pascal Deplaix, Susmit Sarkar [University of St Andrews], Peter Sewell [University of Cambridge].

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

Web site: http://diy.inria.fr/

5.4. Zenon

Participant: Damien Doligez.

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).

Web site: http://zenon-prover.org/

6. New Results

6.1. Formal verification of compilers and static analyzers

6.1.1. Formal verification of static analyzers based on abstract interpretation

Participants: Jacques-Henri Jourdan, Xavier Leroy, Sandrine Blazy [EPI Celtique], Vincent Laporte [EPI Celtique], David Pichardie [EPI Celtique], Sylvain Boulmé [Grenoble INP, VERIMAG], Alexis Fouilhe [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 codes.

This year, Jacques-Henri Jourdan continued the developpment of this static analyzer. He finished the proof of correctness of the abstract interpreter, using an axiomatic semantics for the C#minor intermediate language to decompose this proof in two manageable halves. He improved the precision and performance of the abstract iterator and of numerical abstract domains. He designed and verified a symbolic domain that helps analyzing sequential Boolean operators such as `&&` and `||` that are encoded as Boolean variables and conditional constructs in the C#minor intermediate language. As a more flexible alternative to reduced products of domains, Jacques-Henri Jourdan designed, implemented and proved correct a communication system between numerical abstract domains, based on communication channels and inspired by Astrée [56].

In parallel, IRISA and VERIMAG, our academic partners on the Verasco project, contributed a verified abstract domain for memory states and pointer values (Vincent Laporte, Sandrine Blazy, and David Pichardie) and a polyhedric abstract domain for linear numerical inequalities (Alexis Fouilhe, Sylvain Boulmé, Michaël Périn) that uses validation a posteriori. Those various components were brought together by Jacques-Henri Jourdan and Vincent Laporte, resulting in an executable static analyzer.

The overall architecture and specification of Verasco is described in a paper [29] accepted for presentation at the forthcoming POPL 2015 conference.

6.1.2. The CompCert formally-verified compiler

Participants: Xavier Leroy, Jacques-Henri Jourdan.

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 [5]. This compiler comprises a back-end part, translating the Cminor intermediate language to PowerPC assembly and reusable for source languages other than C [4], and a front-end translating the CompCert C subset of C to Cminor. The compiler is mostly written within the specification language of the Coq proof assistant, from which Coq's extraction facility generates executable Caml 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 parser, previously compiled to unverified OCaml code, was replaced by a parser compiled to Coq code and validated *a posteriori* by a validator written and proved sound in Coq. This validation step, performed when the CompCert compiler is compiled, provides a formal proof that the parser recognizes exactly the language described by the source grammar. This approach builds on the earlier work by Jacques-Henri Jourdan, François Pottier and Xavier Leroy on verified validation of $LR(1)$ parsers [60]. Jacques-Henri Jourdan succeeded in scaling this approach all the way up to the full ISO C99 grammar plus some extensions.

- Two new static analyses, value analysis and neededness analysis, were added to the CompCert back-end. As described in section 6.1.3 below, the results of these analyses enable more aggressive optimizations over the RTL intermediate form.
- As part of the work on formalizing floating-point arithmetic (see section 6.1.4 below), the semantics and compilation of floating-point arithmetic in CompCert was revised to handle single-precision floating-point numbers as first-class values, instead of systematically converting them to double precision before arithmetic. This increases the efficiency and compactness of the code generated for applications that make heavy use of single precision.
- Previously, the CompCert back-end compiler was assuming a partitioned register set from the target architecture, where integer registers always contain 32-bit integers or pointers, and floating-point registers always contain double-precision FP numbers. This convention on register uses simplified the verification of CompCert, but became untenable with the introduction of single-precision FP numbers as first-class values: FP registers can now hold either single- or double-precision FP numbers. Xavier Leroy rearchitected the register allocator and the stack materialization passes of CompCert, along with their soundness proofs, to lift this limitation on register uses. Besides mixtures of single- and double-precision FP numbers, this new architecture makes it possible to support future target processors with a unified register set, such as the SPE variant of PowerPC.
- We added support for several features of ISO C99 that were not handled previously: designated initializers, compound literals, `switch` statements where the `default` case is not the last case, `switch` statements over arguments of 64-bit integer type, and incomplete arrays as the last member of a `struct`. Also, variable-argument functions and the `<stdarg.h>` standard include are now optionally supported, but their implementation is neither specified nor verified.
- The ARM back-end was extended with support for the EABI-HF calling conventions (passing FP arguments and results in FP registers instead of integer registers) and with generation of Thumb2 instructions. Thumb2 is an alternate instruction set and instruction encoding for the ARM architecture that results in more compact machine code (up to 30% reduction in code size on our tests).

We released three versions of CompCert, integrating these enhancements: version 2.2 in February 2014, version 2.3 in April, and version 2.4 in September.

In June 2014, Inria signed a licence agreement with AbsInt Angewandte Informatik GmbH, a software publisher based in Saarbrucken, Germany, to market and provide support for the CompCert formally-verified C compiler. AbsInt will extend CompCert to improve its usability in the critical embedded software market, and also provide long-term maintenance as required in this market.

6.1.3. Value analysis and neededness analysis in CompCert

Participant: Xavier Leroy.

Xavier Leroy designed, implemented, and proved sound two new static analyses over the RTL intermediate representation of CompCert. Both analyses are of the intraprocedural dataflow kind.

- Value analysis is a forward analysis that tracks points-to information for pointers, constantness information for integer and FP numbers, and variation intervals for integer numbers, using intervals of the form $[0, 2^n)$ and $[-2^n, 2^n)$. This value analysis extends and generalizes CompCert's earlier constant analysis as well as the points-to analysis of Robert and Leroy [68]. In particular, it tracks both the values of variables and the contents of memory locations, and it can take advantage of points-to information to show that function-local memory does not escape the scope of the function.
- Neededness analysis is a backward analysis that tracks which memory locations and which bits of the values of integer variables may be used later in a function, and which memory locations and integer bits are "dead", i.e. never used later. This analysis extends CompCert's earlier liveness analysis to memory locations and to individual bits of integer values.

Compared with the static analyses developed as part of Verasco (section 6.1.1), value analysis is much less precise: every function is analyzed independently of its call sites, relations between variables are not tracked, and even interval analysis is coarser (owing to CompCert's lack of support for widened fixpoint iteration). However, CompCert's static analyses are much cheaper than Verasco's, and scale well to large source codes, making it possible to perform them at every compilation run.

Xavier Leroy then modified CompCert's back-end optimizations to take advantage of the results of the two new static analyses, thus improving performance of the generated code:

- Common subexpression elimination (CSE) takes advantage of non-aliasing information provided by value analysis to eliminate redundant memory loads more aggressively.
- Many more integer casts (type conversions) and bit masking operations are discovered to be redundant and eliminated.
- Memory stores and block copy operations that become useless after constant propagation and CSE can now be eliminated entirely.

6.1.4. Verified compilation of floating-point arithmetic

Participants: Sylvie Boldo [EPI Toccata], Jacques-Henri Jourdan, Xavier Leroy, Guillaume Melquiond [EPI Toccata].

In 2012, we replaced the axiomatization of floating-point numbers and arithmetic operations used in early versions of CompCert by a fully-formal Coq development, building on the Coq formalization of IEEE-754 arithmetic provided by the Flocq library of Sylvie Boldo and Guillaume Melquiond. This verification of FP arithmetic and of its compilation was further improved in 2013 with respect to the treatment of "Not a Number" special values.

This year, Guillaume Melquiond improved the algorithmic efficiency of some of the executable FP operations provided by Flocq. Xavier Leroy generalized the theorems over FP arithmetic used in CompCert's soundness proof so that these theorems apply both to single- and double-precision FP numbers. Jacques-Henri Jourdan and Xavier Leroy proved additional theorems concerning conversions between integers and FP numbers.

A journal paper describing this 3-year work on correct compilation of floating-point arithmetic was accepted for publication at Journal of Automated Reasoning [14].

6.1.5. Verified JIT compilation of Coq

Participants: Maxime Dénès, Xavier Leroy.

Evaluation of terms from Gallina, the functional language embedded within Coq, plays a crucial role in the performance of proof checking or execution of verified programs, and the trust one can put in them. Today, Coq provides various evaluation mechanisms, some internal, in the kernel, others external, via extraction to OCaml or Haskell. However, we believe that the specific performance trade-offs and the delicate issues of trust are still calling for a better, more adapted, treatment.

That is why we started in October this year the Coqonut project, whose objective is to develop and formally verify an efficient, compiled implementation of Coq reductions. As a first step, we wrote an unverified prototype in OCaml producing x86-64 machine code using a monadic intermediate form. We started to port it to Coq and to specify the semantics of the source, target and intermediate languages.

6.2. Language design and type systems

6.2.1. The Mezzo programming language

Participants: Thibaut Balabonski, 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.

In 2013 and early 2014, Thibaut Balabonski and François Pottier re-worked the machine-checked proof of type soundness for Mezzo. They developed a version of the proof which includes concurrency and dynamically-allocated locks, and showed that well-typed programs do not crash and are data-race free. This work was presented by François Pottier at FLOPS 2014 [24]. The proof was then extended with a novel and simplified account of adoption and abandon, a mechanism for combining the static ownership discipline with runtime ownership tests. A comprehensive paper, which contains both a tutorial introduction to Mezzo and a description of its formal definition and proof, was submitted to TOPLAS.

Minor modifications were carried out by Jonathan Protzenko in the implementation. A version of Mezzo that runs in a Web browser was developed and uploaded online, so that curious readers can play with the language without installing the software locally.

Jonathan Protzenko wrote his Ph.D. dissertation [12], which describes the design of Mezzo and the implementation of the Mezzo type-checker. He defended on September 29, 2014.

Web site: http://protz.github.io/mezzo/

6.2.2. *System F with coercion constraints*

Participants: Julien Cretin [Trust In Soft], Didier Rémy, Gabriel Scherer.

Expressive type systems often allow non trivial conversions between types, which may lead to complex, challenging, and sometimes ad hoc type systems. Such examples are the extension of System F with type equalities to model GADTs and type families of Haskell, or the extension of System F with explicit contracts. A useful technique to simplify the meta-theoretical study of such systems is to view type conversions as *coercions* inside terms.

Following a general approach based on System F, Julien Cretin and Didier Rémy earlier introduced a language of *explicit coercions* enabling abstraction over coercions and viewing all type transformations as explicit coercions [57]. To ensure that coercions are erasable, *i.e.*, that they only decorate terms without altering their reduction, they are restricted to those that are parametric in either their domain or codomain. Despite this restriction, this language already subsumed many extensions of System F, including bounded polymorphism, instance-bounded polymorphism, and η-conversions—but not subtyping constraints.

To lift this restriction, Julien Crétin and Didier Rémy proposed a new approach were coercions are left implicit. Technically, we extended System F with a rich language of propositions containing a first-order logic, a coinduction mechanism, consistency assertions, and coercions (which are thus just a particular form of propositions); we then introduced a type-level language using kinds to classify types, and constrained kinds to restrict kinds to types satisfying a proposition. Abstraction over types of a constrained kind amounts to abstraction over arbitrary propositions, including coercions.

By default, type abstraction should be erasable, which is the case when kinds of abstract type variables are inhabited—we say that such abstractions are consistent. Still, inconsistent type abstractions are also useful, for instance, to model GADTs. We provide them as a different construct, since they are not erasable, as they must delay reduction of subterms that depend on them. This introduces a form of weak reduction in a language with full reduction, which is a known source of difficulties: although the language remains sound, we loose the subject reduction property. This work has been described in [28] and is part of Julien Cretin's PhD dissertation [11] defended in January 2014; a simpler, core subset is also described in [45].

Recently, Gabriel Scherer and Didier Rémy introduced *assumption hiding* [32], [50] to restore confluence when mixing full and weak reductions and provide a continuum between consistent and inconsistent abstraction. Assumption hiding allows a 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.

6.2.3. *Singleton types for code inference*

Participants: Gabriel Scherer, Didier Rémy.

We continued working on singleton types for code inference. If we can prove that a type contains, in a suitably restricted pure lambda-calculus, a unique inhabitant modulo program equivalence, the compiler can infer the code of this inhabitant. This opens the way to type-directed description of boilerplate code, through type inference of finer-grained type annotations. A decision algorithm for the simply-typed lambda-calculus is still work-in-progress. We presented at the TYPES'14 conference [42] our general approach to such decision procedures, and obtained an independent counting result for intuitionistic logic [52] that demonstrates the finiteness of the search space.

6.2.4. Generic programming with ornaments

Participants: Pierre-Évariste Dagand, Didier Rémy, Thomas Williams.

Since their first introduction in ML, datatypes have evolved: besides providing an organizing *structure* for computation, they are now offering more *control* over what is a valid result. GADTs, which are now part of the OCaml language, offer such a mechanism: ML programmers can express fine-grained, logical invariants of their datastructures. Programmers thus strive to express the correctness of their programs in the types: a well-typed program is correct by construction. However, these carefully crafted datatypes are a threat to any library design: the same data-*structure* is used for many logically incompatible purposes. To address this issue, McBride developed *ornaments*. It defines conditions under which a new datatype definition can be described as an ornament of another one, typically when they both share the same inductive definition scheme. For example, lists can be described as the ornament of the Church encoding of natural numbers. Once a close correspondence between a datatype and its ornament has been established, certain kinds of operations on the original datatype can be automatically lifted to its ornament.

To account for whole-program transformations, we developed a type-theoretic presentation of *functional ornament* [17] as a generalization of ornaments to functions. This work built up on a type-theoretic *universe of datatypes*, a first-class description of inductive types within the type theory itself. Such a presentation allowed us to analyze and compute over datatypes in a transparent manner. Upon this foundation, we formalized the concept of functional ornament by another type-theoretic universe construction. Based on this universe, we established the connection between a base function (such as addition and subtraction) and its ornamented version (such as, respectively, the concatenation of lists and the deletion of a prefix). We also provided support for driving the computer into semi-automatically lifting programs: we showed how addition over natural numbers could be incrementally evolved into concatenation of lists.

Besides the theoretical aspects, we have also tackled the practical question of offering ornaments in an ML setting [33]. Our goal was to extend core ML with support for ornaments so as to enable semi-automatic program transformation and fully-automatic code refactoring. We thus treated the purely syntactic aspects, providing a concrete syntax for describing ornaments of datatypes and specifying the lifting of functions. Such lifting specifications allow the user to declaratively instruct the system to, for example, lift addition of numbers to concatenation of lists. We gave an algorithm that, given a lifting specification, performs the actual program transformation from the bare types to the desired, ornamented types. This work has been evaluated by a prototype implementation in which we demonstrated a few typical use-cases for the semi-automatic lifting of programs.

Having demonstrated the benefits of ornaments in ML, it has been tempting to offer ornaments as first-class citizens in a programming language. Doing so, we wished to rationalize the lifting of programs as an elaboration process within a well-defined, formal system. To describe the liftings, one would like to specify only the local transformations that are applied to the original program. Designing such a language of *patches* and formalizing its elaboration has been the focus of our recent efforts.

6.2.5. Constraints as computations

Participant: François Pottier.

Hindley-Milner type inference–the problem of determining whether an ML program is well-typed–is well-understood, and can be elegantly explained and implemented in two phases, namely constraint generation and constraint solving. In contrast, elaboration–the task of constructing an explicitly-typed representation of the program–seems to have received relatively little attention in the literature, and did not until now enjoy a modular constraint-based presentation. François Pottier proposed such a presentation, which views constraints as computations and equips them with the structure of an applicative functor. This work was presented as a "functional pearl" at ICFP 2014 [31]. The code, in the form of a re-usable library, is available online.

6.2.6. Equivalence and normalization of lambda-terms with sums

Participants: Gabriel Scherer, Guillaume Munch-Maccagnoni [Université 13, LIPN lab].

Determining uniqueness of inhabitants requires a good understanding of program equivalence in presence of sum types. In yet-unpublished work, Gabriel Scherer worked on the correspondence between two existing normalization techniques, one coming from the focusing community [54] and the other using direct lambda-term rewriting [63]. A collaboration with Guillaume Munch-Maccagnoni has also started this year, whose purpose is to present normalization procedures for sums using System L, a rich, untyped syntax of terms (or abstract machines) for the sequent calculus.

6.2.7. Computational interpretation of realizability

Participants: Pierre-Évariste Dagand, Gabriel Scherer.

We are trying to better understand the computational behavior of semantic normalization techniques such as a realizability and logical relation models. As a very first step, we inspected the computational meaning of a normalization proof by realizability for the simply-typed lambda-calculus. It corresponds to an evaluation function; the evaluation order for each logical connective is determined by the definition of the sets of truth and value witnesses. This preliminary work is to be presented at JFLA 2015 [35].

6.3. Shared-memory parallelism

6.3.1. Algorithms and data structures for parallel computing

Participants: Umut Acar, Arthur Charguéraud [EPI Toccata], Mike Rainey.

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 computations 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 two major results this year.

The first result is a sequence data structure that provides amortized constant-time access at the two ends, and logarithmic time concatenation and splitting at arbitrary positions. These operations are essential for programming efficient computation in the fork-join model. Compared with prior work, this novel sequence data structure achieves excellent constant factors, allowing it to be used as a replacement for traditional, non-splittable sequence data structures. This data structure, called *chunked sequence* due to its use of chunks (fixed-capacity arrays), has been implemented both in C++ and in OCaml, and shown competitive with state-of-the art sequence data structures that do not support split and concatenation operations. This work is described in a paper published at ESA [22].

A second main result is the development of fast and robust parallel graph traversal algorithms, more precisely for parallel BFS and parallel DFS. The new algorithms leverage the aforementioned sequence data structure for representing the set of edges remaining to be visited. In particular, it uses the split operation for balancing the edges among the several processors involved in the computation. Compared with prior work, these new algorithms are designed to be efficient not just for particular classes of graphs, but for all input graphs. This work has not yet been published, however it is described in details in a technical report [46].

6.3.2. Weak memory models

Participants: Luc Maranget, Jacques-Pascal Deplaix, Jade Alglave [University College London, then Microsoft Research, Cambridge].

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 execution of its constituting 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.

In the last few years, Luc Maranget took part in an international research effort to define the semantics of the computers of the multi-core era. This research effort relies both on formal methods for defining the models and on intensive experiments for validating the models. Joint work with, amongst others, Jade Alglave (now at Microsoft Research, Cambridge), Peter Sewell (University of Cambridge) and Susmit Sarkar (University of St. Andrews) achieved several significant results, including two semantics for the IBM Power and ARM memory models: one of the operational kind [70] and the other of the axiomatic kind [64]. In particular, Luc Maranget is the main developer of the **diy** tool suite (see section 5.3). Luc Maranget also performs most of the experiments involved.

In 2014 we produced a new model for Power/ARM. The new model is simpler than the previous ones, in the sense that it is based on fewer mathematical objects and can be simulated more efficiently than the previous models. The new **herd** simulator (part of **diy** tool suite) is in fact a generic simulator, whose central component is an interpreter for a domain-specific language. More precisely, memory models are described in a simple language that defines relations by means of a few operators such as concatenation, transitive closure, fixpoint, etc., and performs validity checks on relations such as acyclicity. The Power/ARM model consists of about 50 lines of this specific language. This work, with additional material, including in-depth testing of ARM devices and data-mining of potential concurrency bugs in a huge code base, was published in the journal *Transaction on Programming Languages and Systems* [13] and selected for presentation at the PLDI conference [23]. Luc Maranget gave this presentation.

In the same research theme, Luc Maranget supervised the internship of Jacques-Pascal Deplaix (EPITECH), from Oct. 2013 to May 2014. Jacques-Pascal extended **litmus**, our tool to run tests on hardware. **litmus** now accepts test written in C; we can now perform the conformance testing of C compilers and machines with respect to the C11/C++11 standard. Namely, Mark Batty (University of Cambridge), under the supervision of Jade Alglave, wrote a **herd** model for this standard. The new **litmus** also proves useful to run tests that exploit some machine idiosyncrasies, when our **litmus** assembly implementation does not handle them.

As a part of the **litmus** infrastructure, Luc Maranget designed a synchronisation barrier primitive by simplifying the sense synchronisation barrier published by Maurice Herlily and Nir Shavit in their textbook [58]. He co-authored a JFLA article [34], that presents this primitive and proves it correct automatically by the means of the **cubicle** tool developed under the supervision of Sylvain Conchon (team Toccata, Inria Saclay).

6.4. The OCaml language and system

6.4.1. 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 [OCaml Labs, Cambridge University], Jeremy Yallop [OCaml Labs, Cambridge University].

This year, we released versions 4.02.0 and 4.02.1 of the OCaml system. Release 4.02.0 is a major release that fixes about 60 bugs and introduces 70 new features suggested by users. Damien Doligez acted as release manager for both versions.

OCaml 4.02.0 introduces a large number of major innovations:

- Extension points: a uniform syntax for adding attributes and extensions in OCaml source code: most external preprocessors can now extend the language without need to extend the syntax and reimplement the parser.
- Improvements to the module system: generative functors and module aliases facilitate the efficient handling of large code bases.
- Separation between text-like read-only strings and array-like read-write byte sequences. This makes OCaml programs safer and clearer.
- An extension to the pattern-matching syntax to catch exceptions gives a short, readable way to write some important code patterns.
- Extensible open datatypes generalize the exception type and make its features available for general programming.
- Several important optimizations were added or enhanced: constant propagation, common subexpression elimination, dead code elimination, optimization of pattern-matching on strings.
- A code generator for the new 64-bit ARM architecture "AArch64".
- A safer and faster implementation of the printf function, based on the GADT feature introduced in OCaml 4.00.0.

This version has also seen a reduction in size: the Camlp4 and Labltk parts of the system are now independent systems. This makes them free to evolve on their own release schedules, and to widen their contributor communities beyond the core OCaml team.

OCaml 4.02.1 fixes a few bugs introduced in 4.02.0, along with 25 older bugs.

In parallel, we designed and experimented with several new features that are candidates for inclusion in the next major releases of OCaml:

- Ephemerons: a more powerful version of weak pointers.
- A parallel extension of the runtime system and associated language features that will let multi-threaded OCaml programs run in parallel on several CPU cores.
- Modular implicits: a typeclass-like extension that will make is easy to write generic code (*e.g.* print functions, comparison predicates, overloaded arithmetic operators, etc).
- "Inlined" records as constructor arguments, which will let the programmer select a packed representation for important data structures.
- Major improvements to the inlining optimization pass.
- Support for debugging native-code OCaml programs with GDB.

6.4.2. Namespaces for OCaml

Participants: Fabrice Le Fessant, Pierrick Couderc.

With the growth of the OCaml community and the ease of sharing code through OPAM, the new OCaml package manager, OCaml projects are using more and more external libraries. As a consequence, conflicts between module names of different libraries are now more likely for big projects, and the need for switching from the current flat namespace to a hierarchical namespace is now real.

We experimented with a prototype of OCaml where the namespaces used by a module are explicitely written in the OCaml module source header, to generate the environment in which the source is typed and compiled [39]. Namespaces are mapped on directories on the disk. This mechanism complements the recent addition of module aliases to OCaml, by providing extensibility at the namespace level, whereas it is absent at the module level, and solves also the problem of exact dependency analysis (the previous tool used for that purpose, ocamldep, provides only an approximation of the dependencies, computed on the syntax tree).

6.4.3. Memory profiling OCaml application

Participants: Fabrice Le Fessant, Çagdas Bozman [ENSTA ParisTech], Grégoire Henry [OCamlPro], Michel Mauny [ENSTA ParisTech].

Most modern languages make use of automatic memory management to discharge the programmer from the burden of allocating and releasing the chunks of memory used by the software. 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.

We have experimented with three tools to profile the memory usage of real OCaml applications. The first tool 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. A second tool updates counters at every garbage collection event, it complements the first tool by providing insight on the behavior of the minor heap, and the values that are promoted or not to the major heap. Finally, a third tool samples allocations and saves stacks of function calls at these samples.

These tools have been used on real applications (Alt-Ergo, an SMT solver, or Cumulus, an Ocsigen website), and allowed us to track down and fix memory problems with these applications, such as useless copies of data structures and memory leaks.

6.4.4. OPAM, the OCaml package manager

Participants: Fabrice Le Fessant, Roberto Di Cosmo [IRILL], Louis Gesbert [OCamlPro].

With the growth of the OCaml community, the need for sharing libraries between users has lead to the development of a new package manager, called OPAM. OPAM is based on Dose, a library developed by the Mancoosi team at IRILL, to provide a unified format, CUDF, to query external dependency solvers. The specific needs of OPAM have driven interesting research and improvements on the Dose library, that have consequently opened new opportunities for improvements in OPAM, for the benefit of both software.

We have for example experimented with the design of a specific language [37] to describe optimization criteria, when managing OPAM packages. Indeed, depending on the actions (installation, upgrade, removal), the user might want to reach very different configurations, requiring an expressive power that go far beyond what traditional package managers can express in their configuration options. For example, during installation, the user would probably see as little compilation as possible, whereas upgrading is supposed to move the configuration to the most up-to-date state, with as much compilation as needed.

We have also proposed a new paradigm: multi-switch constraints, to model *switches* used in OPAM to handle different versions of OCaml on the same computer [41]. We proposed this new paradigm as a way to solve multiple problems (cross-compilation, multi-switch packages, per-switch repositories and application-specific switches). However, we expect this new paradigm to challenge the scalability of the current CUDF solvers used by OPAM, and to require important changes and optimization in the Dose library.

6.5. Software specification and verification

6.5.1. Tools for TLA+

Participants: Damien Doligez, Jael Kriener, Leslie Lamport [Microsoft Research], Stephan Merz [EPI VeriDis], Tomer Libal [Microsoft Research-Inria Joint Centre], Hernán Vanzetto [Microsoft Research-Inria Joint Centre].

Damien Doligez is head of the “Tools for Proofs” team in the Microsoft-Inria Joint Centre. The aim of this team is to extend the TLA+ language with a formal language for hierarchical proofs, formalizing the ideas in [61], and to build tools for writing TLA+ specifications and mechanically checking the corresponding formal proofs.

This year, we released two versions of the TLA+ Proof System (TLAPS), the part of the TLA+ tools that handles mechanical checking of TLA+ proofs. This environment is described in [55].

These versions add the propositional temporal logic prover LS4 as a back-end, which allows TLAPS to deal with propositional temporal formulas. This relies on a technique called *coalescing* [40], which allows users to prove arbitrary safety properties, as well as some liveness properties, by translating them into the back-end prover's logic without increasing the complexity of the formulas.

Jael Kriener started a post-doc contract in December 2013, funded by the ADN4SE contract, and left in September 2014. She worked on the theory of temporal proofs in TLA+ and, in collaboration with CEA, on proving some properties of the PharOS real-time operating system.

Web sites:
http://research.microsoft.com/users/lamport/tla/tla.html
http://tla.msr-inria.inria.fr/tlaps

6.5.2. *The Zenon automatic theorem prover*

Participants: Damien Doligez, David Delahaye [CNAM], Pierre Halmagrand [Equipe DEDUCTEAM], Guillaume Bury [Equipe DEDUCTEAM], Olivier Hermant [Mines ParisTech].

Damien Doligez continued the development of Zenon, a tableau-based prover for first-order logic with equality and theory-specific extensions.

Pierre Halmagrand continued his thesis work, funded by ANR BWare, on integrating Deduction Modulo in Zenon, with emphasis on making it efficient for dealing with B set theory.

Guillaume Bury did an internship, also funded by ANR BWare. He implemented an extension of Zenon, based on the simplex method, to deal with arithmetic formulas.

6.5.3. *Well-typed generic fuzzing for module interfaces*

Participants: Thomas Braibant, Jonathan Protzenko, Gabriel Scherer.

Property-based testing generates arbitrary instances of inputs to check a given correctness predicate/property. Thomas Braibant proposed that, instead of a random generation function defined from the internals of one's data-structure, one could use the user-exposed interface to generate instances by composition of API calls. GADTs let us reflect/reify a typed API, and program a type-respecting exploration/testing directly in the host language. We developed a prototype library, Articheck, to experiment with this idea. This work was presented at the ML Workshop [38].

6.5.4. *Depth-First Search and Strong Connectivity in Coq*

Participant: François Pottier.

In 2002, Ingo Wegener published a short paper which sketches a proof of Kosaraju's linear-time algorithm for computing the strongly connected components of a directed graph. At the same time, Wegener's paper helps explain why the algorithm works, which, from a pedagogical standpoint, makes it quite valuable. In 2013 and 2014, François Pottier produced a machine-checked version of Wegener's proof, and wrote a precise informal account of it, which will be presented at JFLA 2015 [36].

6.5.5. *Implementing hash-consed structures in Coq*

Participants: Thomas Braibant, Jacques-Henri Jourdan, David Monniaux [CNRS, VERIMAG].

Hash-consing is a programming technique used to implement maximal sharing of immutable values in memory, keeping a single copy of semantically equivalent objects. Hash-consed data-structures give a unique identifier to each object, allowing fast hashing and comparisons of objects. This may lead to major improvements in execution time by itself, but it also makes it possible to do efficient memoization of computations.

Hash-consing and memoization are examples of imperative techniques that are of prime importance for performance, but are not easy to implement and prove correct using the purely functional language of a proof assistant such as Coq.

We published an article in Journal of Automated Reasoning [15], explaining our work on this subject during the last 3 years. We gave four different approaches for implementing hash-consed data-structures in Coq. Then, we performed an in-depth comparative study of how our "design patterns" for certified hash-consing fare on two real-scale examples: BDDs and lambda-terms.

7. Bilateral Contracts and Grants with Industry

7.1. Bilateral Contracts with Industry

7.1.1. The Caml Consortium

Participants: Xavier Leroy [correspondant], 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 11 member companies:

- CEA
- Citrix
- Dassault Aviation
- Dassault Systèmes
- Esterel Technologies
- Jane Street
- LexiFi
- Microsoft
- Multitudine
- 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. Partnerships and Cooperations

8.1. National Initiatives

8.1.1. ANR projects

8.1.1.1. 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.

8.1.1.2. Paral-ITP

Participant: Damien Doligez.

The "Paral-ITP" project (2011-2014) is coordinated by Burkhart Wolff at Université Paris Sud and funded by the *Ingénierie Numérique et Sécurité* programme of *Agence Nationale de la Recherche*. The objective of Paral-ITP is to investigate the parallelization of interactive theorem provers such as Coq and Isabelle.

8.1.1.3. Verasco

Participants: Jacques-Henri Jourdan, Xavier Leroy.

The "Verasco" project (2012-2015) 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-year project is to develop and formally verify a static analyzer based on abstract interpretation, and interface it with the CompCert C verified compiler.

8.1.2. FSN projects

8.1.2.1. ADN4SE

Participants: Damien Doligez, Jael Kriener.

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.

8.1.2.2. CEEC

Participants: Thomas Braibant, Maxime Dénès, Xavier Leroy.

The "CEEC" project (2011-2014) 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.

8.1.3. FUI projects

8.1.3.1. Richelieu

Participants: Michael Laporte, Fabrice Le Fessant.

The "Richelieu" project (2012-2014) is funded by the *Fonds unique interministériel* (FUI). It involves Scilab Enterprises, U. Pierre et Marie Curie, Dassault Aviation, ArcelorMittal, CNES, Silkan, OCamlPro, and Inria. The objective of the project is to improve the performance of scientific programming languages such as Scilab's through the use of VMKit and LLVM.

8.2. European Initiatives

8.2.1. FP7 & H2020 Projects

8.2.1.1. DEEPSEA

Type: FP7

Defi: NC

Instrument: ERC Starting Grant

Objectif: NC

Duration: June 2013 - May 2018

Coordinator: Umut Acar

Partner: Inria

Inria contact: Umut Acar

Abstract: the objective of project DEEPSEA is to develop abstractions, algorithms and languages for parallelism and dynamic parallelism, with applications to problems on large data sets.

8.3. International Initiatives

8.3.1. Inria International Partners

8.3.1.1. 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.

8.4. International Research Visitors

8.4.1. Visits of International Scientists

8.4.1.1. Internships

Sigurd Schneider, Ph.D. student at Saarlandes University in Saarbrucken, visited Gallium from Mar 2014 to May 2014. As part of his Ph.D., Sigurd Schneider develops an intermediate representation that unifies static single assignment form (SSA) and functional intermediate representations. During his internship, he considered the addition of GC support to this intermediate representation. He also developed a program logic to verify the correctness of a class of optimizations, including constant subexpression elimination (CSE) and global value numbering.

8.4.1.2. Research stays abroad

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

9. Dissemination

9.1. Promoting Scientific Activities

9.1.1. Scientific events organisation

9.1.1.1. Member of the organizing committee

Maxime Dénès was member of the organizing committee for the First International Workshop on Coq for Programming Languages (CoqPL 2015), co-located with POPL'15 in Mumbai, India.

9.1.2. Scientific events selection

9.1.2.1. Chair of the conference program committee

Xavier Leroy co-chaired the program committee of the ACM SIGPLAN 2015 conference on Certified Proofs and Programs (CPP'15), which will take place in January 2015 in Mumbai, India, colocated with POPL'15.

9.1.2.2. Member of the conference program committee

- **ALENEX 2015** (Algorithm Engineering & Experiments): Umut Acar was a member of the program committee.
- **ECOOP 2014** (European Conference on Object-Oriented Programming): Mike Rainey was a member of the artefact evaluation committee.
- **ESOP 2015** (European Symposium on Programming): Umut Acar was a member of the program committee.
- **Eurosys 2015** (European Conference on Systems Research and Development): Pierre-Évariste Dagand was a member of the shadow program committee.
- **POPL 2015** (42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages): Umut Acar and François Pottier were members of the program committee; Xavier Leroy was a member of the external review committee.
- **CoqPL 2015** (Workshop on Coq for Programming Languages): Maxime Dénès and Xavier Leroy were members of the program committee.
- **MLW 2015** (ACM SIGPLAN ML Family Workshop): Didier Rémy was a member of the program committee.
- **WGP 2014** (Workshop on Generic Programming): Pierre-Évariste Dagand was a member of the program committee.

9.1.2.3. Reviewer

In 2014, members of Gallium reviewed approximately 95 papers submitted to international conferences and 15 papers submitted to international workshops.

9.1.3. Journals

9.1.3.1. Member of the editorial board

- **Communications of the ACM**: Xavier Leroy is on the editorial board for the Research Highlights column.
- **Journal of Automated Reasoning**: Xavier Leroy is a member of the editorial board.
- **Journal of Functional Programming**: François Pottier is a member of the editorial board.
- **Journal of Formalized Reasoning**: Xavier Leroy is a member of the editorial board.

9.1.3.2. Reviewer

In 2014, members of Gallium reviewed 4 journal submissions.

9.2. Teaching - Supervision - Juries

9.2.1. Teaching

Licence: Umut Acar and Mike Rainey, "Introduction to parallel computing with PASL", 15h, L2, University of Puerto Rico, USA.

Licence: Jacques-Henri Jourdan, "Langages de programmation et compilation" 46h, L3, École Normale Supérieure, France.

Licence: François Pottier, "Algorithmique et programmation" (INF431), 13h30, L3, École Polytechnique, France.

Licence: Gabriel Scherer, “IP1: Introduction to computer science and programming using Java”, 42h, L1, University Paris Diderot, France.

Licence: Gabriel Scherer, “IP1: Introduction to computer science and programming using C”, 42h, L3, University Paris Diderot, France.

Master: Xavier Leroy and Didier Rémy, “Functional programming languages”, 15h + 20h, M2R MPRI, University Paris Diderot/ENS Paris/ENS Cachan/Polytechnique, France.

Master: Luc Maranget, “Semantics, languages and algorithms for multi-core programming”, 9h, M2 MPRI, University Paris Diderot/ENS Paris/ENS Cachan/Polytechnique, France.

Master: François Pottier, “Compilation” (INF564), 13h30, M1, École Polytechnique, France.

9.2.2. Supervision

PhD: Çagdas Bozman, “Profilage mémoire d’applications OCaml”, Ecole Polytechnique, defended on 16 Dec 2014, supervised by Michel Mauny (ENSTA), Pierre Chambart (OCamlPro), and Fabrice Le Fessant (Inria).

PhD: Julien Cretin, “Erasable coercions: a unified approach to type systems” [11], École Polytechnique, defended on 30 Jan 2014, supervised by Didier Rémy.

PhD: Jonathan Protzenko, “Mezzo: a typed language for safe effectful concurrent programs” [12], U. Paris Diderot, defended on 29 Sep 2014, supervised by François Pottier.

PhD in progress: Arthur Guillon, “Concurrent realizability and program verification for weak memory computers”, U. Paris Diderot, started September 2014, interrupted January 2015, supervised by Luc Maranget and Jade Alglave (Microsoft Research Cambridge).

PhD in progress: Pierre Halmagrand, “Déduction Automatique Modulo”, CNAM, since September 2013, supervised by David Delahaye (CNAM), Damien Doligez (Inria), and Olivier Hermant (Mines ParisTech).

PhD in progress: Gabriel Scherer, “Term inference: proof search for singleton inhabitants”, U. Paris Diderot, since October 2011, supervised by Didier Rémy.

PhD in progress: Jacques-Henri Jourdan, “Formal verification of a static analyzer for critical embedded software”, U. Paris Diderot, since September 2012, supervised by Xavier Leroy.

Pre-PhD ENS year: Thomas Williams, “Putting ornaments into practice”, since September 2014, supervised by Didier Rémy.

9.2.3. Juries

Damien Doligez was a member of the Ph.D. jury of Çagdas Bozman, ENSTA ParisTech, Palaiseau, December 2014.

Xavier Leroy chaired the PhD committee of André Maroneze, University Rennes 1, June 2014.

Xavier Leroy was a member of the committees for the PhD awards of ACM SIGPLAN and of the GDR GPL (*Génie de la Programmation et du Logiciel*).

9.2.4. Collective responsibilities

Damien Doligez chaired the *Commission des actions de développement technologiques* of Inria Paris-Rocquencourt until Oct 2014.

Xavier Leroy is *vice-président du comité des projets* of Inria Paris-Rocquencourt and appointed member of Inria’s *Commission d’Évaluation*. He participated in the following Inria hiring and promotion committees: *jury d’admissibilité CR2 Saclay*; *jury d’admissibilité DR2*; *promotions CR1*; *promotions DR1 and DR0*.

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.

Didier Rémy is deputy scientific director of Inria, in charge of the fields Algorithmics, Programming, Software and Architecture.

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.

9.3. Popularization

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

Jacques-Henri Jourdan manned a stand at “Salon Culture & Jeux Mathématiques”, in Paris.

Since 2012, the Gallium team publishes a research blog at http://gallium.inria.fr/blog/, edited by Gabriel Scherer. This blog continued its activity in 2014, with 23 posts from 5 different authors.

10. Bibliography

Major publications by the team in recent years

[1] 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

[2] J. CRETIN, D. RÉMY. *On the Power of Coercion Abstraction*, in "Proceedings of the 39th ACM Symposium on Principles of Programming Languages (POPL'12)", ACM Press, 2012, pp. 361–372, http://dx.doi.org/10.1145/2103656.2103699

[3] 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

[4] 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

[5] 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

[6] 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

[7] 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

[8] F. POTTIER, D. RÉMY. *The Essence of ML Type Inference*, in "Advanced Topics in Types and Programming Languages", B. C. PIERCE (editor), MIT Press, 2005, chap. 10, pp. 389–489

[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

Doctoral Dissertations and Habilitation Theses

[11] J. CRETIN. *Erasable coercions: a unified approach to type systems*, Université Paris-Diderot - Paris VII, January 2014, https://tel.archives-ouvertes.fr/tel-00940511

[12] J. PROTZENKO. *Mezzo: a typed language for safe effectful concurrent programs*, Université Paris Diderot - Paris 7, September 2014, https://hal.inria.fr/tel-01086106

Articles in International Peer-Reviewed Journals

[13] J. ALGLAVE, L. MARANGET, M. TAUTSCHNIG. *Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory*, in "ACM Trans. On Programming Languages and Systems (TOPLAS)", June 2014, vol. 36, n° 2, pp. 7:1–7:74 [*DOI :* 10.1145/2627752], https://hal.inria.fr/hal-01081364

[14] 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

[15] T. BRAIBANT, J.-H. JOURDAN, D. MONNIAUX. *Implementing and reasoning about hash-consed data structures in Coq*, in "Journal of Automated Reasoning", 2014, vol. 53, n° 3, pp. 271-304 [*DOI :* 10.1007/s10817-014-9306-0], https://hal.inria.fr/hal-00881085

[16] Y. CHEN, J. DUNFIELD, A. HAMMER, U. A. ACAR. *Implicit self-adjusting computation for purely functional programs*, in "Journal of Functional Programming", January 2014, vol. 24, n° 1, pp. 56-112, https://hal.inria.fr/hal-01100346

[17] P.-É. DAGAND, C. MCBRIDE. *Transporting functions across ornaments*, in "Journal of Functional Programming", May 2014, vol. 24, n° 2-3, 67 p. [*DOI :* 10.1017/S0956796814000069], https://hal.inria.fr/hal-00922581

Invited Conferences

[18] X. LEROY. *Compiler verification for fun and profit*, in "FMCAD 2014 - Formal Methods in Computer-Aided Design", Lausanne, Switzerland, K. CLAESSEN, V. KUNCAK (editors), FMCAD Inc, October 2014, 9 p. , https://hal.inria.fr/hal-01076547

[19] X. LEROY. *Formal proofs of code generation and verification tools*, in "SEFM 2014 - 12th International Conference Software Engineering and Formal Methods", Grenoble, France, D. GIANNAKOPOULOU, G. SALAÜN (editors), Lecture Notes in Computer Science, Springer, September 2014, vol. 8702, pp. 1-4 [*DOI :* 10.1007/978-3-319-10431-7_1], https://hal.inria.fr/hal-01059423

[20] X. LEROY. *Formal verification of a static analyzer: abstract interpretation in type theory*, in "Types - The 2014 Types Meeting", Paris, France, May 2014, https://hal.inria.fr/hal-00983847

[21] X. LEROY. *Proof assistants in computer science research*, in "IHP thematic trimester on Semantics of proofs and certified mathematics", Paris, France, Institut Henri Poincaré, April 2014, https://hal.inria.fr/hal-00983850

International Conferences with Proceedings

[22] U. A. ACAR, A. CHARGUÉRAUD, M. RAINEY. *Theory and Practice of Chunked Sequences*, in "European Symposium on Algorithms", Wrocław, Poland, A. SCHULZ, D. WAGNER (editors), Lecture Notes in Computer Science, Springer Berlin Heidelberg, September 2014, n^{o} 8737, pp. 25 - 36 [*DOI :* 10.1007/978-3-662-44777-2_3], https://hal.inria.fr/hal-01087245

[23] J. ALGLAVE, L. MARANGET, M. TAUTSCHNIG. *Herding cats: Modelling, simulation, testing, and data-mining for weak memory*, in "PLDI '14: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation", Edinburg, United Kingdom, ACM, June 2014, 40 p. [*DOI :* 10.1145/2594291.2594347], https://hal.inria.fr/hal-01081413

[24] T. BALABONSKI, F. POTTIER, J. PROTZENKO. *Type Soundness and Race Freedom for Mezzo*, in "FLOPS 2014: 12th International Symposium on Functional and Logic Programming", Kanazawa, Japan, LNCS, Springer, June 2014, vol. 8475, pp. 253 - 269 [*DOI :* 10.1007/978-3-319-07151-0_16], https://hal.inria.fr/hal-01081194

[25] P. BHATOTIA, U. A. ACAR, P. JUNQUEIRA, R. RODRIGUES. *Slider: Incremental Sliding Window Analytics*, in "Middleware 2014: Proceedings of the 15th International Middleware Conference", Bordeaux, France, December 2014 [*DOI :* 10.1145/2663165.2663334], https://hal.inria.fr/hal-01100350

[26] Y. CHEN, U. A. ACAR, K. TANGWONGSAN. *Functional Programming for Dynamic and Large Data with Self-Adjusting Computation*, in "ICFP 2014: 19th ACM SIGPLAN International Conference on Functional Programming", Gothenburg, Sweden, September 2014 [*DOI :* 10.1145/2628136.2628150], https://hal.inria.fr/hal-01100337

[27] J. CHENEY, A. AMAL, U. A. ACAR. *Database Queries that Explain their Work*, in "PPDP 2014: 16th International Symposium on Principles and Practice of Declarative Programming", Canterbury, United Kingdom, September 2014 [*DOI :* 10.1145/2643135.2643143], https://hal.inria.fr/hal-01100324

[28] J. CRETIN, D. RÉMY. *System F with Coercion Constraints*, in "CSL-LICS 2014: Joint Meeting of the Annual Conference on Computer Science Logic and the Annual Symposium on Logic in Computer Science", Vienna, Austria, T. A. HENZINGER, D. MILLER (editors), ACM, July 2014, 34 p. [*DOI :* 10.1145/2603088.2603128], https://hal.inria.fr/hal-01093239

[29] 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

[30] R. KREBBERS, X. LEROY, F. WIEDIJK. *Formal C semantics: CompCert and the C standard*, in "ITP 2014: Fifth conference on Interactive Theorem Proving", Vienna, Austria, Lecture Notes in Computer Science, Springer, July 2014, vol. 8558, pp. 543-548 [*DOI :* 10.1007/978-3-319-08970-6_36], https://hal.inria.fr/hal-00981212

[31] F. POTTIER. *Hindley-Milner Elaboration in Applicative Style*, in "ICFP 2014: 19th ACM SIGPLAN International Conference on Functional Programming", Goteborg, Sweden, ACM, September 2014 [*DOI :* 10.1145/2628136.2628145], https://hal.inria.fr/hal-01081233

[32] 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

[33] T. WILLIAMS, P.-É. DAGAND, D. RÉMY. *Ornaments in practice*, in "WGP 2014: ACM workshop on Generic programming", Gothenburg, Sweden, August 2014 [*DOI :* 10.1145/2633628.2633631], http://hal.upmc.fr/hal-01081547

National Conferences with Proceedings

[34] S. CONCHON, L. MARANGET, A. MEBSOUT, D. DECLERCK. *Vérification de programmes C concurrents avec Cubicle : Enfoncer les barrières*, in "JFLA", Fréjus, France, January 2014, https://hal.inria.fr/hal-01088655

[35] 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

[36] 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

Conferences without Proceedings

[37] P. ABATE, R. DI COSMO, L. GESBERT, F. LE FESSANT, S. ZACCHIROLI. *Using Preferences to Tame your Package Manager*, in "OCaml 2014", Goteborg, Sweden, September 2014, https://hal.inria.fr/hal-01091177

[38] T. BRAIBANT, J. PROTZENKO, G. SCHERER. *Well-typed generic smart-fuzzing for APIs*, in "ML'14 - ACM SIGPLAN ML Family Workshop", Göteborg, Sweden, August 2014, https://hal.inria.fr/hal-01094006

[39] P. COUDERC, B. CANOU, P. CHAMBART, F. LE FESSANT. *A Proposal for Non-Intrusive Namespaces in OCaml*, in "OCaml 2014", Goteborg, Sweden, September 2014, https://hal.inria.fr/hal-01091173

[40] D. DOLIGEZ, J. KRIENER, L. LAMPORT, T. LIBAL, S. MERZ. *Coalescing: Syntactic Abstraction for Reasoning in First-Order Modal Logics*, in "ARQNL 2014 - Automated Reasoning in Quantified Non-Classical Logics", Vienna, Austria, July 2014, https://hal.inria.fr/hal-01063512

[41] F. LE FESSANT. *A Case for Multi-Switch Constraints in OPAM*, in "OCaml 2014", goteborg, Sweden, September 2014, https://hal.inria.fr/hal-01091175

[42] G. SCHERER, D. RÉMY. *Deciding unique inhabitants with sums (work in progress)*, in "TYPES", Paris, France, May 2014, https://hal.inria.fr/hal-01094127

Scientific Books (or Scientific Book chapters)

[43] X. LEROY, A. TIU. *CPP '15: Proceedings of the 2015 Conference on Certified Programs and Proofs*, ACM, January 2015, 184 p. , https://hal.inria.fr/hal-01101937

[44] X. LEROY, A. W. APPEL, S. BLAZY, G. STEWART. *The CompCert memory model*, in "Program Logics for Certified Compilers", A. W. APPEL (editor), Cambridge University Press, April 2014, pp. 237-271, https://hal.inria.fr/hal-00905435

[45] D. RÉMY, J. CRETIN. *From Amber to Coercion Constraints*, in "Essays for the Luca Cardelli Fest", M. ABADI, P. GARDNER, A. D. GORDON, R. MARDARE (editors), TechReport, Microsoft Research, September 2014, n^{o} MSR-TR-2014-104, https://hal.inria.fr/hal-01093216

Research Reports

[46] U. A. ACAR, A. CHARGUÉRAUD, M. RAINEY. *Data Structures and Algorithms for Robust and Fast Parallel Graph Search*, Inria, December 2014, https://hal.inria.fr/hal-01089125

[47] J. CRETIN, D. RÉMY. *System F with Coercion Constraints*, January 2014, n^{o} RR-8456, 36 p. , https://hal.inria.fr/hal-00934408

[48] X. LEROY, D. DOLIGEZ, A. FRISCH, J. GARRIGUE, D. RÉMY, J. VOUILLON. *The OCaml system release 4.02: Documentation and user's manual*, Inria, September 2014, https://hal.inria.fr/hal-00930213

[49] X. LEROY. *The CompCert C verified compiler: Documentation and user's manual*, Inria, September 2014, https://hal.inria.fr/hal-01091802

[50] G. SCHERER, D. RÉMY. *Full reduction in the face of absurdity*, Inria, December 2014, https://hal.inria.fr/hal-01093910

Other Publications

[51] G. CANO, C. COHEN, M. DÉNÈS, A. MÖRTBERG, V. SILES. *Formalized Linear Algebra over Elementary Divisor Rings in Coq*, November 2014, https://hal.inria.fr/hal-01081908

[52] G. SCHERER. *2-or-more approximation for intuitionistic logic*, November 2014, https://hal.inria.fr/hal-01094120

References in notes

[53] V. BENZAKEN, G. CASTAGNA, A. FRISCH. *CDuce: an XML-centric general-purpose language*, in "ICFP'03: International Conference on Functional Programming", ACM Press, 2003, pp. 51–63

[54] K. CHAUDHURI, D. MILLER, A. SAURIN. *Canonical Sequent Proofs via Multi-Focusing*, in "TCS 2008 – Fifth IFIP International Conference On Theoretical Computer Science", G. AUSIELLO, J. KARHUMÄKI, G. MAURI, C. L. ONG (editors), IFIP, Springer, 2008, vol. 273, pp. 383–396, http://dx.doi.org/10.1007/978-0-387-09680-3_26

[55] 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

[56] P. COUSOT, R. COUSOT, J. FERET, L. MAUBORGNE, A. MINÉ, D. MONNIAUX, X. RIVAL. *Combination of Abstractions in the Astrée Static Analyzer*, in "ASIAN 2006: 11th Asian Computing Science Conference", Lecture Notes in Computer Science, Springer, 2006, vol. 4435, pp. 272-300

[57] J. CRETIN, D. RÉMY. *On the Power of Coercion Abstraction*, in "Proceedings of the 39th ACM Symposium on Principles of Programming Languages (POPL'12)", ACM Press, 2012, pp. 361–372, http://dx.doi.org/10.1145/2103656.2103699

[58] M. HERLIHY, N. SHAVIT. *The Art of Multiprocessor Programming*, Morgan Kaufmann Publishers Inc.San Francisco, CA, USA, 2008

[59] H. HOSOYA, B. C. PIERCE. *XDuce: A Statically Typed XML Processing Language*, in "ACM Transactions on Internet Technology", May 2003, vol. 3, n^{o} 2, pp. 117–148

[60] J.-H. JOURDAN, F. POTTIER, X. LEROY. *Validating LR(1) Parsers*, in "Programming Languages and Systems – 21st European Symposium on Programming, ESOP 2012", H. SEIDL (editor), Lecture Notes in Computer Science, Springer, 2012, vol. 7211, pp. 397–416, http://dx.doi.org/10.1007/978-3-642-28869-2_20

[61] 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

[62] 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

[63] S. LINDLEY. *Extensional Rewriting with Sums*, in "TLCA 2007 – Typed Lambda Calculi and Applications, 8th International Conference", Springer, 2007, pp. 255–271, http://dx.doi.org/10.1007/978-3-540-73228-0_19

[64] S. MADOR-HAIM, L. MARANGET, S. SARKAR, K. MEMARIAN, J. ALGLAVE, S. OWENS, R. ALUR, M. MARTIN, P. SEWELL, D. WILLIAMS. *An Axiomatic Memory Model for Power Multiprocessors*, in "CAV 2012: Computer Aided Verification, 24th International Conference", Lecture Notes in Computer Science, Springer, 2012, vol. 7358, pp. 495-512

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

[66] F. POTTIER. *Simplifying subtyping constraints: a theory*, in "Information and Computation", 2001, vol. 170, n^{o} 2, pp. 153–183

[67] 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

[68] V. ROBERT, X. LEROY. *A Formally-Verified Alias Analysis*, in "Certified Programs and Proofs – Second International Conference, CPP 2012", C. HAWBLITZEL, D. MILLER (editors), Lecture Notes in Computer Science, Springer, 2012, vol. 7679, pp. 11-26, http://dx.doi.org/10.1007/978-3-642-35308-6_5

[69] 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

[70] S. SARKAR, P. SEWELL, J. ALGLAVE, L. MARANGET, D. WILLIAMS. *Understanding Power multiprocessors*, in "PLDI 2011: 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation", ACM, 2011, pp. 175-186