Activity Report 2012

Project-Team GALLIUM

Programming languages, types, compilation and proofs

RESEARCH CENTER
Paris - Rocquencourt

THEME
Programs, Verification and Proofs

Table of contents

Project-Team GALLIUM

Keywords: Programming Languages, Functional Programming, Compilation, Type Systems, Safety, Proofs Of Programs

Creation of the Project-Team: May 01, 2006 .

1. Members

Research Scientists
Xavier Leroy [Team leader, DR Inria]
Damien Doligez [CR Inria]
François Pottier [DR Inria, HdR]
Didier Rémy [Deputy team leader, DR Inria, HdR]
Na Xu [CR Inria, until August 2012]

Engineers
Xavier Clerc [IR Inria, SED, 30% part time]
Valentin Robert [Research programmer, January–September 2012]

PhD Students
Julien Cretin [AMX grant, U. Paris Diderot]
Jacques-Henri Jourdan [ENS Paris student, since September 2012]
Jonathan Protzenko [CORDI-S grant, U. Paris Diderot]
Gabriel Scherer [AMN grant, U. Paris Diderot]

Post-Doctoral Fellows
Thomas Braibant [since September 2012]
Thibaut Balabonski [since October 2012]

Administrative Assistants
Stéphanie Chaix [Temporary personnel, until September 2012]
Virginie Collette [AI Inria, since October 2012]

Others
Raphaël Proust [graduate intern, ENS Cachan, April–August 2012]
Joseph Tassarotti [undergraduate intern, Harvard University, June–August 2012]

2. Overall Objectives

2.1. Introduction

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.

2.2. Highlights of the Year

Xavier Leroy was awarded the 2012 Microsoft Research Verified Software Milestone Award in recognition of his work on the CompCert C verified compiler.

3. Scientific Foundations

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 [36]. Through collaborations, we also gained experience with several domain-specific languages based on a functional core, including XML processing (XDuce, CDuce), reactive functional programming, distributed programming (JoCaml), and hardware modeling (ReFLect).

3.2. Type systems

Type systems [49] 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 [46], [42], 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 [53], 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 [50].

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. Finally, we also participate in the FoCaLiZe project, which designs and implements an environment for combined programming and proving [23] [52].

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 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 [48] and enforcement of data confidentiality through type-based inference of information flows and noninterference properties [51].

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. Languages such as CDuce and OCamlDuce extend these benefits to the handling of semi-structured XML data [44]. 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, USA, and Japan.

5. Software

5.1. OCaml

Participants: Damien Doligez [correspondant], Xavier Clerc [team SED], Alain Frisch [LexiFi], Jacques Garrigue [Nagoya University], Thomas Gazagnaire [OCamlPro], Fabrice Le Fessant [Inria Saclay and OCamlPro], Xavier Leroy, Luc Maranget [EPI Moscova].

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, etc) 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, compilation manager, and the Camlp4 source pre-processor.

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

5.2. CompCert C

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

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 long` and `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. 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).

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

6. New Results

6.1. Language design and type systems

6.1.1. The Mezzo programming language

Participants: Jonathan Protzenko, François Pottier.

In the past ten years, the type systems community and the separation logic community, among others, have developed highly expressive formalisms for describing ownership policies and controlling side effects in imperative programming languages. In spite of this extensive knowledge, it remains very difficult to come up with a programming language design that is simple, effective (it actually controls side effects!) and expressive (it does not force programmers to alter the design of their data structures and algorithms).

The Mezzo programming language, formerly known as HaMLet, aims to bring new answers to these questions.

We have come up with a solid design for the programming language: many features of the language have been reworked or consolidated this year, and we believe we strike a good balance between expressiveness and complexity. We wrote several flagship examples that illustrate the gains offered by Mezzo, as well as two (yet unpublished) papers discussing the design of the language. Jonathan Protzenko implemented a prototype type-checker; although it is not perfect yet, several non-trivial examples are successfully type-checked.

The current state of the Mezzo programming language is best described in [40]; a former version of this document can be found as [39].

François Pottier wrote a formal definition of (a slightly lower-level variant of) Mezzo, and proved that Mezzo is type-safe: that is, well-typed programs cannot crash (but they can stop abruptly if a run-time check fails). The proof, which is about 15,000 lines, has been machine-checked using Coq. A paper that describes this work is in preparation.

This work was facilitated by Pottier's experience with a similar previous proof. In particular, out of the above 15,000 lines, about 2,000 lines correspond to a re-usable library for working with de Bruijn indices, and about 3,000 lines correspond to a re-usable formalisation of "monotonic separation algebras", which help reason about resources (memory, time, knowledge, ...) and how they evolve over time. These libraries have not yet been fully documented and released; this might be done in the future.

6.1.2. Coercion abstraction

Participants: Julien Cretin, Didier Rémy.

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 studies of such systems is to make type conversions explicit as "coercions" inside terms.

Following a general approach to coercions based on System F, we introduced a language F-iota with abstraction over coercions and where all type transformations are represented as coercions. The main difficulty is dealing with coercion abstraction, as abstract coercions whose types are uninhabited cannot be erased at run-time. We proposed a restriction, called parametric F-iota, that ensures erasability of all coercions by construction. This work was presented at the POPL conference in January [22].

We extended parametric F-iota with non-interleaved positive recursive types and with erasable isomorphisms. We generalized the presentation of the language viewing coercions as conversions between typings (pairs of a typing environment and a type) rather than between types. An extended version with full proofs will be submitted for journal publication.

We also studied a more liberal version of F-iota where coercion inhabitation is no more ensured by construction (which limits expressiveness), but instead by providing coercion witnesses in source terms. This extension requires pushing abstract coercions under redexes so that they do not block the reduction. As a consequence, coercions cannot be reified in System F, and we need a direct proof of termination of iota-reduction. We completed one such proof based on reducibility candidates.

6.1.3. Ambivalent types for principal type inference with GADTs

Participants: Jacques Garrigue [Nagoya University], Didier Rémy.

Type inference for Generalized Abstract Data Types (GADTs) is always a matter of compromise because it is inherently non monotone: assuming more specific types for GADTs may ensure more invariants, which in turn may result in more general types. Moreover, even when types of GADTs parameters are explicitly given, they introduce equalities between types, which makes them inter-convertible but with a limited scope. This may then creates an ambiguity when leaving the scope of the equation: which representative should be used for the equivalent forms? Ideally, one should use a type disjunction, but this is not allowed—for good reasons. Hence, to avoid arbitrary choices, these situations must be rejected, forcing the user to add more annotations to resolve ambiguities.

We proposed a new approach to type inference with GADTs. While some uses of equations are unavoidable and create real ambiguities, others are gratuitous and create artificial ambiguities, To distinguish between the two, we introduced *ambivalent types*: a way to trace types that have been obtained by an unavoidable use of an equation. We then redefined ambiguities so that only ambivalent types become ambiguous and should be rejected or resolved by a programmer annotation.

Interestingly, the solution is fully compatible with unification-based type inference algorithms used in ML dialects. The work was presented at the ML workshop [31] and implemented in the latest version 4.00 of OCaml.

6.1.4. GADTs and Subtyping

Participants: Gabriel Scherer, Didier Rémy.

Following the addition of GADTs to the OCaml language in version 4.00 released this year, we studied the theoretical underpinnings of variance subtyping for GADTs. The question is to decide which variances should be accepted for a GADT-style type declaration that includes type equality constraints in constructor types. This question exposes a new notion of decomposability and unexpected tensions in the design of a subtyping relation. Our formalization partially reuses earlier work by François Pottier and Vincent Simonet [54]. It was presented at the ML Workshop [33]. An extended version including full proofs is available as a technical report [38] and was submitted for presentation at a conference.

6.1.5. Singleton types for code inference

Participants: Gabriel Scherer, Didier Rémy.

Inspired by tangent aspects of the PhD work of Julien Cretin, we investigated the use of 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. The preliminary results seem encouraging, both on the theoretical side (identifying general situations for type-directed programming) and the practical side (mining existing OCaml code for usage situations).

6.1.6. Programming with names and binders

Participants: Nicolas Pouillard, François Pottier.

Following Nicolas Pouillard's Ph.D. defense in January 2012 [11], Nicolas Pouillard and François Pottier produced a unified presentation of Pouillard's approach to programming with abstract syntax, in the form of a paper that was published in the Journal of Functional Programming [16].

6.1.7. A type-and-capability calculus with hidden state

Participant: François Pottier.

During the year 2010, François Pottier developed a machine-checked proof of an expressive type-and-capability system, which can be used to type-check and prove properties of imperative ML programs. The proof is carried out in Coq and takes up roughly 20,000 lines of code. In the first half of 2011, François Pottier wrote a paper that describes the system and its proof in detail. This paper was published, after a revision, in 2012 [15].

6.2. Formal verification of compilers and static analyses

6.2.1. The CompCert verified C compiler

Participants: Xavier Leroy, Sandrine Blazy [project-team Celtique], Jacques-Henri Jourdan, Valentin Robert.

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.

The two major novelties of CompCert this year are described separately: verification of floating-point arithmetic (section 6.2.2) and a posteriori validation of assembly and linking (section 6.2.3). Other improvements to CompCert include:

- The meaning of "volatile" memory accesses is now fully specified in the semantics of the CompCert C source language. Their translation to built-in function invocations, previously part of the unverified pre-front-end part of CompCert, is now proved correct.
- CompCert C now natively supports assignment between composite types (structs or unions), passing composite types by value as function parameters, and other instances of using composites as r-values, with the exception of returning composites by value from a function.
- A new pass was added to the compiler to perform inlining of functions. Its correctness proof raised interesting challenges to properly relate the (widely different) call stacks of the program before and after inlining.
- The constant propagation optimization is now able to propagate the initial values of global variables declared `const`.
- The common subexpression elimination (CSE) optimization was improved so as to eliminate more redundant memory loads.

Two versions of the CompCert development were publicly released, integrating these improvements: versions 1.10 in March and 1.11 in July. We also wrote a 50-page user's manual [37] and a technical report on the CompCert memory model [35].

In parallel, we continued our collaboration with Jean Souyris, Ricardo Bedin França and Denis Favre-Felix at Airbus. They are conducting an experimental evaluation of CompCert's usability for avionics software, and studying the regulatory issues (DO-178 certification) surrounding the potential use of CompCert in this context. Preliminary results were presented at the 2012 Embedded Real-Time Software and Systems conference (ERTS'12) [29].

6.2.2. *Formalization of floating-point arithmetic in Compcert*

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

The aim of this research theme was to formalize the semantics and compilation of floating-point arithmetic in the CompCert compiler. Prior to this work, floating-point arithmetic was axiomatized in the Coq proof of CompCert, then mapped to OCaml's floating-point operations during extraction. This approach was prone to errors and fails to formally guarantee conformance to the IEEE-754 standard for floating-point arithmetic.

To remedy this situation, Jacques-Henri Jourdan replaced this axiomatization by a fully-formal Coq development, building on the Coq formalization of IEEE-754 arithmetic provided by the Flocq library. Sylvie Boldo and Guillaume Melquiond, authors of Flocq, adapted their library to the needs of this development. The new formalization of floating-point arithmetic is used throughout CompCert: to give semantics to FP computations in the source, intermediate and target (assembly) languages; to perform correct compile-time FP evaluations during constant propagation; to prove the correctness of code generation scheme for conversions between integers and FP numbers; and to parse FP literals with correct rounding.

A paper describing this work is accepted for presentation at the forthcoming ARITH 2013 conference [20].

6.2.3. *Validation of assembly and linking*

Participants: Valentin Robert, Xavier Leroy.

Valentin Robert designed and implemented a validation tool for the assembly and linking phases of the CompCert C compiler. These passes are not formally verified and call into off-the-shelf assemblers and linkers. The `cchecklink` tool of Valentin Robert improves the confidence that end-users can have in these passes by validating *a posteriori* their operation. The tool takes as inputs the PowerPC/ELF executable produced by the linker, as well as the abstract syntax trees for assembly files produced by the formally-verified part of CompCert. It then proceeds to establish a correspondence between the two sets of inputs, via a thorough structural analysis on the ELF executable, light disassembling of the machine code, expansion of CompCert's macro-asm instructions, and propagation of constraints over symbolic names. The tool produces detailed diagnostics if any discrepancies are found.

6.2.4. Improving CompCert's reusability for verification tools

Participants: Xavier Leroy, Jacques-Henri Jourdan, Andrew Appel [Princeton University], Sandrine Blazy [project-team Celtique], David Pichardie [project-team Celtique].

Several ongoing projects focus on proving the soundness of verification tools that reuse parts of the CompCert development, namely some of the intermediate languages, their formal semantics, and the CompCert passes that produce these intermediate forms. This is the case for the Verasco ANR project, which focuses on the proof of a static analyzer based on abstract interpretation, and for the Verified Software Toolchain (VST) project, led by Andrew Appel at Princeton University, which develops a concurrent separation logic embedded in Coq. However, the CompCert intermediate languages, currently designed to fit the needs of a compiler, are not perfectly suited to static analysis and deductive verification.

To improve the reusability of CompCert's Clight language in the Verasco and VST projects, Xavier Leroy is currently revising the CompCert C front-end passes so that function-local C variables whose address is never taken are pulled out of memory and replaced by nonadressable temporary variables. The resulting Clight intermediate form is much easier to analyze or prove correct, as temporary variables cannot suffer from aliasing problems.

Likewise, Sandrine Blazy, Jacques-Henri Jourdan, Xavier Leroy and David Pichardie designed a variant of CompCert's RTL intermediate language, called CFG. Like RTL, CFG represents the flow of control by a graph; unlike RTL, CFG is independent of the target processor, and supports complex expressions instead of 3-address code. These features of CFG make it a better target for static analysis, both non-relational (e.g. David Pichardie's certified interval analysis) and relational. Jacques-Henri Jourdan implemented and proved correct a compilation pass that produces CFG code from the Cminor intermediate language of CompCert.

6.2.5. Formal verification of hardware synthesis

Participants: Thomas Braibant, Adam Chlipala [MIT].

Verification of hardware designs has been thoroughly investigated, and yet, obtaining provably correct hardware of significant complexity is usually considered challenging and time-consuming. Hardware synthesis aims to raise the level of description of circuits, reducing the effort necessary to produce them.

This yields two opportunities for formal verification: a first option is to verify (part of) the hardware compiler; a second option is to study to what extent these higher-level design are amenable to formal proof.

During a visit at MIT, Thomas Braibant worked on the implementation and proof of correctness of a prototype hardware compiler in Coq, under Adam Chlipala's supervision. This compiler produces descriptions of circuits in RTL style from a high-level description language inspired by BlueSpec. After joining Gallium, Thomas Braibant continued working part time on this subject, finishing the proof of this compiler, and implementing a few hardware designs of mild complexity. This work was presented at the 2012 Coq Workshop [30] and will be submitted to a conference in 2013.

6.2.6. A formally-verified alias analysis

Participants: Valentin Robert, Xavier Leroy.

Valentin Robert improved the verified static analysis for pointers and non-aliasing that he initiated in 2011 during his Master's internship supervised by Xavier Leroy. This alias analysis is intraprocedural and flow-sensitive, and follows the "points-to" approach of Andersen [41]. An originality of this alias analysis is that it is conducted over the RTL intermediate language of the CompCert compiler: since RTL is essentially untyped, the traditional approaches to field sensitivity do not apply, and are replaced by a simple but effective tracking of the numerical offsets of pointers with respect to their base memory blocks. The soundness of this alias analysis is proved against the operational semantics of RTL using the Coq proof assistant and techniques inspired from abstract interpretation. A paper describing the analysis and its soundness proof was presented at the CPP 2012 conference [26].

6.3. The OCaml language and system

6.3.1. The OCaml system

Participants: Xavier Clerc [team SED], Damien Doligez, Alain Frisch [Lexifi SAS], Jacques Garrigue [University of Nagoya], Fabrice Le Fessant [Inria Saclay and OCamlPro start-up company], Jacques Le Normand [Lexifi SAS], Xavier Leroy.

This year, we released versions 4.00.0 and 4.00.1 of the OCaml system. Version 4.00.0 (released in July) is a major release that fixes about 150 reported bugs and 4 unreported bugs, and introduces 57 new features suggested by users. Version 4.00.1 (released in October) is a bug-fix release that fixes 3 major and 20 minor bugs. Damien Doligez acted as release manager for both versions.

The major innovation in OCaml 4.00 is support for generalized algebraic datatypes (GADTs). These non-uniform datatype definitions enable programmers to express some invariants over data structures, and the OCaml type-checker to enforce these invariants. They also support interesting ways of reflecting types into run-time values. GADTs are found in proof assistants such as Coq and in functional languages such as Agda and Haskell. Their integration in OCaml raised delicate issues of partial type inference and principality of inferred types, to which Jacques Garrigue and Jacques Le Normand provided original solutions [45].

Other features of this release include:

- Lightweight notations to facilitate the use of first-class modules.
- Better reporting of type errors.
- Improvements in native-code generation.
- Performance and security improvements in the hashing primitive and hash tables.
- New warnings for unused code (variables, record fields, etc.)
- A new back-end for the ARM architecture.

6.3.2. Namespaces for OCaml

Participants: Gabriel Scherer, Didier Rémy, Fabrice Le Fessant [Inria Saclay].

As part of an ongoing discussion among members of the OCaml Consortium, we investigated the formal aspects of "namespaces" and their putative status in the OCaml language. Namespaces aim at providing OCaml programmers with efficient ways to manage and structure the names of compilation units, in contrast with the flat, global space of compilation units provided today in OCaml. This formalization provides scientific support to ongoing design and engineering discussions. It was presented at the December 2011 IFIP 2.8 working group on functional programming, and at the December 2012 meeting of the OCaml Consortium.

6.4. Software specification and verification

6.4.1. Tools for TLA+

Participants: Damien Doligez, 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 [47], and to build tools for writing TLA+ specifications and mechanically checking the corresponding formal proofs.

This year, the TLA+ project released two new versions (in January and in November) of the TLA+ tools: the GUI-based TLA Toolbox and the TLA+ Proof System, an environment for writing and checking TLA+ proofs. This environment is described in a paper presented at the 2012 symposium on Formal Methods [21]. The January release (version 1.0 of TLAPS and 1.4.1 of Toolbox) added support for back-ends based on SMT provers (CVC3, Z3, Yices, VeriT), which dramatically extends the range of proof obligations that the system can discharge automatically. The November release includes many bug-fixes and performance improvements.

We have also improved the theoretical design of the proof language with respect to temporal properties. This design will be implemented in TLAPS in the near future.

Web site: http://tlaplus.net/

6.4.2. The Zenon automatic theorem prover

Participants: Damien Doligez, David Delahaye [CNAM], Mélanie Jacquel [CNAM].

Damien Doligez continued the development of Zenon, a tableau-based prover for first-order logic with equality and theory-specific extensions. Version 0.7.1 of Zenon was released in May.

David Delahaye and Mélanie Jacquel designed and implemented (with some help from Damien Doligez) an extension of Zenon called SuperZenon, based on the Superdeduction framework of Brauner, Houtmann, and Kirchner [43].

Both Zenon and SuperZenon entered the CASC theorem-proving contest, where, as expected, SuperZenon did much better than Zenon.

6.4.3. Hybrid contract checking via symbolic simplification

Participant: Na Xu.

Program errors are hard to detect or prove absent. Allowing programmers to write formal and precise specifications, especially in the form of contracts, is one popular approach to program verification and error discovery. Na Xu formalizes and implements a hybrid contract checker for a pure subset of OCaml. The key technique we use is symbolic simplification, which makes integrating static and dynamic contract checking easy and effective. This technique statically verifies that a function satisfies its contract or blames the function violating the contract. When a contract satisfaction is undecidable, it leaves residual code for dynamic contract checking.

A paper describing this result is published in the proceeding of the PEPM'2012 conference [27]. An extended version of this paper will appear in the journal Higher-Order and Symbolic Computation. Na Xu implemented this approach in a prototype based on the OCaml 3.12.1 compiler and experimented with nontrivial examples such as sorting algorithms and balancing AVL trees (see http://gallium.inria.fr/~naxu/research/hcc.html).

6.4.4. Probabilistic contracts for component-based design

Participants: Na Xu, Gregor Goessler [project-team POPART], Alain Girault [project-team POPART].

We define a framework of probabilistic contracts for constructing component-based embedded systems, based on the formalism of discrete-time Interactive Markov Chains. A contract specifies the assumptions a component makes on its context and the guarantees it provides. Probabilistic transitions represent allowed uncertainty in the component behavior, for instance, to model internal choice or reliability. Action transitions are used to model non-deterministic behavior and communication between components. An interaction model specifies how components interact with each other.

We provide the ingredients for a component-based design flow, including (1) contract satisfaction and refinement, (2) parallel composition of contracts over disjoint, interacting components, and (3) conjunction of contracts describing different requirements over the same component. Compositional design is enabled by congruence of refinement. A paper describing the details of this result is published in the journal Formal Methods in System Design [14].

7. Bilateral Contracts and Grants with Industry

7.1. The Caml Consortium

Participants: Xavier Leroy [correspondant], Xavier Clerc, Damien Doligez, Didier Rémy.

The Caml Consortium is a formal structure where industrial and academic users of Caml can support the development of the language and associated tools, express their specific needs, and contribute to the long-term stability of Caml. Membership fees are used to fund specific developments targeted towards industrial users. Members of the Consortium automatically benefit from very liberal licensing conditions on the OCaml system, allowing for instance the OCaml compiler to be embedded within proprietary applications.

The Consortium currently has 12 member companies: CEA, Citrix, Dassault Aviation, Dassault Systèmes, Esterel Technologies, Jane Street, LexiFi, Microsoft, MLstate, Mylife.com, OCamlPro, and 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. ADN4SE (FSN)

Participant: Damien Doligez.

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. BWare (ANR)

Participant: Damien Doligez.

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.

8.1.3. CEEC (FSN)

Participants: Thomas Braibant, 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.

8.1.4. LaFoSec

Participant: Damien Doligez.

The LaFoSec study, commissioned by ANSSI, aims at studying the security properties of functional languages, and especially of OCaml. The study is done by a consortium led by the SafeRiver company. Last year, it produced more than 600 pages of documents, including recommendations for security-aware development in OCaml.

The study continued this year with the production of a prototype of a secure XML/XSD validator following these recommendations, and a security evaluation of the prototype by an independent company.

Most of these documents will be made available in 2013 on the ANSSI Web site (http://ssi.gouv.fr/).

8.1.5. Paral-ITP (ANR)

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.

8.1.6. U3CAT (ANR)

Participant: Xavier Leroy.

The "U3CAT" project (2009-2012) ended in August 2012. It was coordinated by Virgile Prevosto at CEA LIST and funded by the *Arpège* programme of *Agence Nationale de la Recherche*. This action focused on program verification tools for critical embedded C codes. We were involved in this project on issues related to memory models [35] and formal semantics for the C language, at the interface between compilers and verification tools.

8.1.7. Verasco (ANR)

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.

8.2. International Research Visitors

8.2.1. Visits of International Scientists

Gabriel Dos Reis, assistant professor at Texas A&M University, visited the Gallium team in July 2012, to work on the formal semantics of the C and C++ languages.

8.2.1.1. Internships

Joseph Tassarotti, undergraduate student at Harvard University, did an internship at Gallium from June to August 2012. He worked on register allocation and instruction scheduling for the CompCert verified compiler.

9. Dissemination

9.1. Scientific Animation

9.1.1. Conference organization

Didier Rémy co-chaired the OCaml Users and Developers Workshop (OUD 2012), affiliated with ICFP 2012, which took place in Copenhagen, Denmark in September 2012.

Didier Rémy is organizing the next meeting of IFIP working group 2.8 "Functional programming", which will take place in Aussois, France, in October 2013.

9.1.2. Editorial boards

Xavier Leroy was co-editor in chief of the Journal of Functional Programming until March 2012, when he stepped down at the end of his 5-year term.

Xavier Leroy is a member of the editorial boards of Journal of Automated Reasoning, Journal of Functional Programming, and Journal of Formalized Reasoning.

9.1.3. Program committees

Damien Doligez was a member of the program committee of the International Workshop on the TLA+ Method and Tools 2012 (TLA 2012), a satellite event of Formal Methods 2012 (FM 2012)

Xavier Leroy was a member of the program committees of the ACM symposium on Principles of Programming Languages (POPL 2013) and of the European Symposium on Programming (ESOP 2013). He was a member of the external review committee for the ACM conference on Programming Languages Design and Implementation (PLDI 2013).

François Pottier was a member of the program committees of the ACM workshops on ML (ML 2012), on Higher-Order Programming with Effects (HOPE 2012), and on Programming Languages meets Program Verification (PLPV 2013). He was a member of the external review committee for the ACM symposium on Principles of Programming Languages (POPL 2013).

Na Xu was a member of the program committee of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM 2013).

9.1.4. Steering committees

Xavier Leroy is a member of the steering committees for the Certified Programs and Proofs (CPP) conference and for the ACM PLPV workshop.

François Pottier is a member of the steering committee for the ACM TLDI workshop.

Didier Rémy is a member of the steering committee of the OCaml Users and Implementors Workshop.

9.2. Teaching - Supervision - Juries

9.2.1. Teaching

Licence: "Algorithmique et programmation" (INF431), 13h30, L3, École Polytechnique, France. Lecturer: François Pottier (*professeur chargé de cours*).

Licence: "Algorithmique et programmation", 40h, L3, École Polytechnique, France. Teaching assistant: Jonathan Protzenko.

Licence: "Bases de données", 26h, L1, U. Paris Diderot, France. Teaching assistant: Julien Cretin.

Licence: "Principe de fonctionnement des machines binaires", 33h, L1, U. Paris Diderot, France. Teaching assistant: Julien Cretin.

Licence: “Travaux dirigés de Caml Light”, 36h, L1 (classes préparatoires MPSI), Lycée Louis-le-Grand, France. Teaching assistant: Gabriel Scherer.

Master: “Functional programming and type systems”, 30h, M2, MPRI master (U. Paris Diderot and ENS Paris and ENS Cachan and Polytechnique), France. Lecturers: Xavier Leroy (12h) and Didier Rémy (18h).

Master: “Modal Web”, 36 hours, M1, École Polytechnique, France. Teaching assistant: Jonathan Protzenko.

Doctorat: “Proving a compiler: mechanized verification of program transformations and static analyses”, 7h, Oregon Programming Languages Summer School, USA. Lecturer: Xavier Leroy.

9.2.2. Supervision

PhD: Nicolas Pouillard, “A unifying approach to safe programming with first-order syntax with binders”, U. Paris Diderot, defended January 13th, 2012, supervised by François Pottier.

PhD: Tahina Ramananandro, “Mechanized formal semantics and verified compilation for C++ objects”, U. Paris Diderot, defended January 10th, 2012, supervised by Xavier Leroy.

PhD in progress: Julien Cretin, “Coercions in typed languages”, since December 2010, supervised by Didier Rémy.

PhD in progress: Alexandre Pilkiewicz (currently employed by Google in New York), “Verifying polyhedral optimizations”, U. Paris Diderot, since December 2008, supervised by François Pottier

PhD in progress: Jonathan Protzenko, “Fine-grained static control of side effects”, U. Paris Diderot, since September 2010, supervised by François Pottier.

PhD in progress: Gabriel Scherer, “Modules and mixins”, U. Paris Diderot, since October 2011, supervised by Didier Rémy.

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

9.2.3. Juries

Xavier Leroy was reviewer (*rapporteur*) for the Ph.D. of Robert Dockins (Princeton University, August 2012), Antonis Stampoulis (Yale University, October 2012), and Delphine Demange (ENS Cachan Bretagne, October 2012). He was a member of the Ph.D. jury of Ricardo Bedin França (U. Toulouse, April 2012). Xavier Leroy chaired the Habilitation jury of Fabrice Rastello, ENS Lyon, December 2012.

François Pottier was an external examiner for the Habilitation defense of Etienne Lozes (École Normale Supérieure de Cachan, July 3, 2012) and for the Ph.D. defense of Jérémy Planul (École Polytechnique, February 8, 2012), Thibaut Balabonski (Université Paris Diderot, November 16, 2012), and Antoine Madet (Université Paris Diderot, December 6, 2012).

9.3. Popularization

Jacques-Henri Jourdan participated in the organization of the *Castor* computer science contest (http://castor-informatique.fr/). This contest aims at making computer science more popular in French high schools and junior high schools. It attracted over 90,000 participants.

Xavier Leroy gave a popular science talk on critical avionics software at the December 2012 “Jam session” meeting of Inria Alumni.

Jonathan Protzenko, along with two other PhD students, is curating the Junior Seminar in Rocquencourt, where he coaches other PhD students into presenting their work in front of a friendly audience.

Gabriel Scherer set up a research blog for the Gallium project-team, to highlight small chunks of work of the team members in an informal and popularized style. For example, Valentin Robert, Thomas Braibant and Jacques-Henri Jourdan described various aspects of their interaction with the Coq proof assistant, some of which eventually resulted in small improvements integrated by the Coq development team. Articles are published approximately once a week, and the blog gets on average 500 visits a day.

10. Bibliography

Major publications by the team in recent years

[1] A. CHARGUÉRAUD, F. POTTIER. *Functional Translation of a Calculus of Capabilities*, in "Proceedings of the 13th International Conference on Functional Programming (ICFP'08)", ACM Press, September 2008, p. 213–224, http://doi.acm.org/10.1145/1411204.1411235.

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

[3] D. LE BOTLAN, D. RÉMY. *Recasting MLF*, in "Information and Computation", 2009, vol. 207, n^{o} 6, p. 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, p. 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, p. 107–115, http://doi.acm.org/10.1145/1538788.1538814.

[6] B. MONTAGU, D. RÉMY. *Modeling Abstract Types in Modules with Open Existential Types*, in "Proceedings of the 36th ACM Symposium on Principles of Programming Languages (POPL'09)", ACM Press, January 2009, p. 354-365, http://doi.acm.org/10.1145/1480881.1480926.

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

[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, p. 389–489.

[9] N. POUILLARD, F. POTTIER. *A fresh look at programming with names and binders*, in "Proceedings of the 15th International Conference on Functional Programming (ICFP 2010)", ACM Press, 2010, p. 217–228, http://doi.acm.org/10.1145/1863543.1863575.

[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, p. 83–92, http://doi.acm.org/10.1145/1706299.1706311.

Publications of the year

Doctoral Dissertations and Habilitation Theses

[11] N. POUILLARD. *Une approche unifiante pour programmer sûrement avec de la syntaxe du premier ordre contenant des lieurs / Namely, Painless: A unifying approach to safe programming with first-order syntax with binders*, Université Paris Diderot (Paris 7), January 2012, http://tel.archives-ouvertes.fr/tel-00759059.

[12] T. RAMANANANDRO. *Machine-checked Formal Semantics and Verified Compilation for C++ Objects*, Université Paris Diderot (Paris 7), January 2012, http://tel.archives-ouvertes.fr/tel-00769044.

Articles in International Peer-Reviewed Journals

[13] A. W. APPEL, R. DOCKINS, X. LEROY. *A list-machine benchmark for mechanized metatheory*, in "Journal of Automated Reasoning", 2012, vol. 49, n° 3, p. 453–491, http://dx.doi.org/10.1007/s10817-011-9226-1.

[14] G. GÖSSLER, D. N. XU, A. GIRAULT. *Probabilistic contracts for component-based design*, in "Formal Methods in System Design", 2012, vol. 41, n° 2, p. 211–231, http://dx.doi.org/10.1007/s10703-012-0162-4.

[15] F. POTTIER. *Syntactic soundness proof of a type-and-capability system with hidden state*, in "Journal of Functional Programming", 2013, vol. 23, n° 1, p. 38–144, to appear, http://dx.doi.org/10.1017/S0956796812000366.

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

[17] D. RÉMY, B. YAKOBOWSKI. *A Church-Style Intermediate Language for MLF*, in "Theoretical Computer Science", 2012, vol. 435, n° 1, p. 77–105, http://dx.doi.org/10.1016/j.tcs.2012.02.026.

[18] J. SCHWINGHAMMER, L. BIRKEDAL, F. POTTIER, B. REUS, K. STØVRING, H. YANG. *A step-indexed Kripke Model of Hidden State*, in "Mathematical Structures in Computer Science", 2013, vol. 23, n° 1, p. 1–54, to appear, http://dx.doi.org/10.1017/S0960129512000035.

Invited Conferences

[19] X. LEROY. *Mechanized Semantics for Compiler Verification*, in "Programming Languages and Systems, 10th Asian Symposium, APLAS 2012", R. JHALA, A. IGARASHI (editors), Lecture Notes in Computer Science, Springer, 2012, vol. 7705, p. 386–388, Abstract of invited talk, http://dx.doi.org/10.1007/978-3-642-35182-2_27.

International Conferences with Proceedings

[20] S. BOLDO, J.-H. JOURDAN, X. LEROY, G. MELQUIOND. *A Formally-Verified C Compiler Supporting Floating-Point Arithmetic*, in "IEEE Symposium on Computer Arithmetic, ARITH 2013", IEEE Computer Society Press, 2013, to appear, http://hal.inria.fr/hal-00743090.

[21] 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, p. 147-154, http://dx.doi.org/10.1007/978-3-642-32759-9_14.

[22] 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, p. 361–372, http://dx.doi.org/10.1145/2103656.2103699.

[23] D. DOLIGEZ, M. JAUME, R. RIOBOO. *Development of secured systems by mixing programs, specifications and proofs in an object-oriented programming environment: a case study within the FoCaLiZe environment*, in

"Proceedings of the 7th Workshop on Programming Languages and Analysis for Security (PLAS'12)", ACM Press, 2012, p. 9:1–9:12, http://doi.acm.org/10.1145/2336717.2336726.

[24] 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, p. 397–416, http://dx.doi.org/10.1007/978-3-642-28869-2_20.

[25] T. RAMANANANDRO, G. DOS REIS, X. LEROY. *A Mechanized Semantics for C++ Object Construction and Destruction, with Applications to Resource Management*, in "Proceedings of the 39th ACM Symposium on Principles of Programming Languages (POPL'12)", ACM Press, 2012, p. 521–532, http://dx.doi.org/10.1145/2103656.2103718.

[26] 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, p. 11-26, http://dx.doi.org/10.1007/978-3-642-35308-6_5.

[27] D. N. XU. *Hybrid contract checking via symbolic simplification*, in "Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM'12)", ACM Press, 2012, p. 107–116, http://dx.doi.org/10.1145/2103746.2103767.

[28] B. YORGEY, S. WEIRICH, J. CRETIN, JOSÉ PEDRO. MAGALHÃES, S. PEYTON JONES, D. VYTINIOTIS. *Giving Haskell a Promotion*, in "The Seventh ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI'12)", ACM Press, 2012, p. 67–78, http://dx.doi.org/10.1145/2103786.2103795.

Conferences without Proceedings

[29] R. BEDIN FRANÇA, S. BLAZY, D. FAVRE-FELIX, X. LEROY, M. PANTEL, J. SOUYRIS. *Formally verified optimizing compilation in ACG-based flight control software*, in "Embedded Real Time Software and Systems (ERTS2 2012)", AAAF, SEE, 2012, http://hal.inria.fr/hal-00653367/.

[30] T. BRAIBANT, A. CHLIPALA. *Formal verification of hardware synthesis*, in "The Coq Workshop", 2012, http://gallium.inria.fr/~braibant/data/braibant-talk-coq-workshop-2012.pdf.

[31] J. GARRIGUE, D. RÉMY. *Tracing ambiguity in GADT type inference*, in "ACM SIGPLAN Workshop on ML", 2012, http://www.lexifi.com/ml2012/full7.pdf.

[32] J. PROTZENKO, F. POTTIER. *Programming with permissions: the Mezzo language*, in "ACM SIGPLAN Workshop on ML", 2012, http://www.lexifi.com/ml2012/full1.pdf.

[33] G. SCHERER, D. RÉMY. *GADTs meet subtyping*, in "ACM SIGPLAN Workshop on ML", 2012, http://www.lexifi.com/ml2012/full14.pdf.

Research Reports

[34] G. GÖSSLER, D. N. XU, A. GIRAULT. *Probabilistic Contracts for Component-based Design*, Inria, July 2012, n^{o} RR-7328, http://hal.inria.fr/hal-00715750.

[35] X. LEROY, A. W. APPEL, S. BLAZY, G. STEWART. *The CompCert Memory Model, Version 2*, Inria, June 2012, n^{o} RR-7987, http://hal.inria.fr/hal-00703441.

[36] X. LEROY, D. DOLIGEZ, J. GARRIGUE, D. RÉMY, J. VOUILLON. *The Objective Caml system, documentation and user's manual – release 4.00*, Inria, July 2012, http://caml.inria.fr/pub/docs/manual-ocaml-4.00/.

[37] X. LEROY. *The CompCert C verified compiler, documentation and user's manual*, Inria, July 2012, http://compcert.inria.fr/man/.

[38] G. SCHERER, D. RÉMY. *GADT meet Subtyping*, Inria, October 2012, n° RR-8114, http://hal.inria.fr/hal-00744292.

Other Publications

[39] F. POTTIER, J. PROTZENKO. *An introduction to Mezzo*, September 2012, Unpublished draft, http://gallium.inria.fr/~fpottier/publis/mezzo-tutorial.pdf.

[40] F. POTTIER, J. PROTZENKO. *Programming with permissions in Mezzo*, October 2012, Submitted for publication, http://gallium.inria.fr/~fpottier/publis/pottier-protzenko-mezzo.pdf.

References in notes

[41] L. O. ANDERSEN. *Program Analysis and Specialization for the C Programming Language*, DIKU, University of Copenhagen, 1994.

[42] V. BENZAKEN, G. CASTAGNA, A. FRISCH. *CDuce: an XML-centric general-purpose language*, in "Int. Conf. on Functional programming (ICFP'03)", ACM Press, 2003, p. 51–63.

[43] P. BRAUNER, C. HOUTMANN, C. KIRCHNER. *Principles of Superdeduction*, in "22nd IEEE Symposium on Logic in Computer Science (LICS 2007)", IEEE Computer Society Press, 2007, p. 41-50, http://hal.inria.fr/inria-00133557.

[44] A. FRISCH. *OCaml + XDuce*, in "Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming", ACM Press, September 2006, p. 192–200, http://doi.acm.org/10.1145/1159803.1159829.

[45] J. GARRIGUE, J. LE NORMAND. *Adding GADTs to OCaml: the direct approach*, in "ACM SIGPLAN Workshop on ML", ACM Press, 2011.

[46] H. HOSOYA, B. C. PIERCE. *XDuce: A Statically Typed XML Processing Language*, in "ACM Transactions on Internet Technology", May 2003, vol. 3, n° 2, p. 117–148.

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

[48] X. LEROY. *Java bytecode verification: algorithms and formalizations*, in "Journal of Automated Reasoning", 2003, vol. 30, n° 3–4, p. 235–269, http://gallium.inria.fr/~xleroy/publi/bytecode-verification-JAR.pdf.

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

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

[51] F. POTTIER, V. SIMONET. *Information Flow Inference for ML*, in "ACM Transactions on Programming Languages and Systems", January 2003, vol. 25, n^{o} 1, p. 117–158, http://gallium.inria.fr/~fpottier/publis/fpottier-simonet-toplas.ps.gz.

[52] V. PREVOSTO, D. DOLIGEZ. *Algorithms and Proofs Inheritance in the FOC Language*, in "Journal of Automated Reasoning", 2002, vol. 29, n^{o} 3–4, p. 337-363.

[53] 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, p. 40–53.

[54] V. SIMONET, F. POTTIER. *A Constraint-Based Approach to Guarded Algebraic Data Types*, in "ACM Transactions on Programming Languages and Systems", January 2007, vol. 29, n^{o} 1, article no. 1, http://gallium.inria.fr/~fpottier/publis/simonet-pottier-hmg-toplas.ps.gz.