Activity Report 2013

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, from Jun 2013]
- Damien Doligez [Inria, Researcher]
- Fabrice Le Fessant [Inria, Researcher, from Jan 2013]
- Luc Maranget [Inria, Researcher, from Jan 2013]
- François Pottier [Inria, Senior Researcher, HdR]
- Mike Rainey [Inria, Starting Research position, from Jun 2013]
- Didier Rémy [Inria, Senior Researcher, HdR]

Engineer
- Michael Laporte [Inria, granted by FUI Richelieu project, from Apr 2013]

PhD Students
- Julien Cretin [U. Paris Diderot, AMX grant]
- Jacques-Henri Jourdan [Inria, granted by ANR VERASCO project]
- Jonathan Protzenko [Inria]
- Gabriel Scherer [U. Paris Diderot, AMN grant]

Post-Doctoral Fellows
- Thibaut Balabonski [Inria]
- Thomas Braibant [Inria, granted by FSN CEEC]
- Pierre-Evariste Dagand [Inria, from Nov 2013]
- Jael Kriener [Inria, granted by FSN ADN4SE, from Dec 2013]
- Grégoire Henry [Inria, granted by OCamlPro, from Feb 2013 until Oct 2013]
- Luca Saiu [Inria, granted by OCamlPro, from Apr 2013 until Sep 2013]

Visiting Scientists
- Olin Shivers [on sabbatical from Northeastern U., from Jul 2013 until Dec 2013]
- Robbert Krebbers [Ph.D. student, Radboud U., from Jan 2013 until Mar 2013]

Administrative Assistant
- Virginie Collette [Inria]

Others
- Michael Arntzenius [intern, CMU, from Jun 2013 until Aug 2013]
- Jacques-Pascal Deplaix [intern, EPITECH, from Oct 2013]
- Armaël Guéneau [intern, ENS Lyon, from Jun 2013 until Jul 2013]
- Cyprien Mangin [intern, Ecole Polytechnique, from Apr 2013 until Jul 2013]

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

Didier Le Botlan (INSA Toulouse) and Didier Rémy received the ACM SIGPLAN Most Influential ICFP Paper Award for their ICFP 2003 paper, *MLF: Raising ML to the power of System F* [44].

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

3.2. Type systems

Type systems [47] are a very effective way to improve programming language reliability. By grouping the data manipulated by the program into classes called types, and ensuring that operations are never applied to types over which they are not defined (e.g. accessing an integer as if it were an array, or calling a string as if it were a function), a tremendous number of programming errors can be detected and avoided, ranging from the trivial (misspelled identifier) to the fairly subtle (violation of data structure invariants). These restrictions are also very effective at thwarting basic attacks on security vulnerabilities such as buffer overflows.

The enforcement of such typing restrictions is called type checking, and can be performed either dynamically (through run-time type tests) or statically (at compile-time, through static program analysis). We favor static type checking, as it catches bugs earlier and even in rarely-executed parts of the program, but note that not all type constraints can be checked statically if static type checking is to remain decidable (i.e. not degenerate into full program proof). Therefore, all typed languages combine static and dynamic type-checking in various proportions.

Static type checking amounts to an automatic proof of partial correctness of the programs that pass the compiler. The two key words here are *partial*, since only type safety guarantees are established, not full correctness; and *automatic*, since the proof is performed entirely by machine, without manual assistance from the programmer (beyond a few, easy type declarations in the source). Static type checking can therefore be viewed as the poor man's formal methods: the guarantees it gives are much weaker than full formal verification, but it is much more acceptable to the general population of programmers.

3.2.1. Type systems and language design.

Unlike most other uses of static program analysis, static type-checking rejects programs that it cannot 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 [41], [35], 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 [51], integrated in OCaml, extended the benefits of this approach (reusable code with no loss of typing precision) to object-oriented programming, extensible records and extensible variants. Another example is the work by Pottier on subtype polymorphism, using a constraint-based formulation of the type system [48].

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 [45] and enforcement of data confidentiality through type-based inference of information flows and noninterference properties [49].

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 [39]. 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. Software and Platforms

5.1. OCaml

Participants: Damien Doligez [correspondant], Alain Frisch [LexiFi], Jacques Garrigue [Nagoya University], Fabrice Le Fessant, Xavier Leroy, Luc Maranget.

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, 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 [University College London], 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.

The tool suite and a comprehensive documentation are available from 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).

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

5.5. JoCaml

Participant: Luc Maranget.

JoCaml is an experimental extension of OCaml that adds support for concurrent and distributed programming, following the programming model of the join-calculus.

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

6. New Results

6.1. Formal verification of compilers and static analyzers

6.1.1. The CompCert formally-verified compiler

Participants: Xavier Leroy, Jacques-Henri Jourdan, Robbert Krebbers.

In the context of our work on compiler verification (see section 3.3.1), since 2005 we have been developing and formally verifying a moderately-optimizing compiler for a large subset of the C programming language, generating assembly code for the PowerPC, ARM, and x86 architectures [6]. This compiler comprises a back-end part, translating the Cminor intermediate language to PowerPC assembly and reusable for source languages other than C [5], 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 released three versions of CompCert. Version 1.13, released in March, improves conformance with the ISO C standard by defining the semantics of comparisons involving pointers "one past" the end of an array. Such comparisons used to be undefined behaviors in earlier versions of CompCert. Robbert Krebbers formalized a reasonable interpretation of the ISO C rules concerning pointers "one past" and adapted CompCert's proofs accordingly. CompCert 1.13 also features minor performance improvements for the ARM and PowerPC back-ends, notably for parameter passing via stack locations.

Version 2.0 of CompCert, released in June, re-architects the compiler back-end around the new register allocator described in section 6.1.2. Besides improving the performance of generated code, this new allocator made it possible to add support for 64-bit integers, that is, the `long long` and `unsigned long long` data types of ISO C99. Most arithmetic operations over 64-bit integers are expanded in-line and proved correct, but a few complex operations (division, modulus, and conversions to and from floating-point numbers) are implemented as calls into library functions.

Moreover, conformance with Application Binary Interfaces was improved, especially concerning the passing of function parameters and results of type `float` (single-precision FP numbers).

Finally, CompCert 2.0 features preliminary support for debugging information. The `-g` compiler flag causes DWARF debugging information to be generated for line numbers and call stack structure. However, no information is generated yet for C type definitions and variable declarations.

Version 2.1, released in October, addresses several shortcomings of CompCert for embedded system codes, as identified by Airbus during their experimental evaluation of CompCert. In particular, CompCert 2.1 features the `_Alignas` modifier introduced in ISO C2011, to support precise control of alignment of global variables and structure fields, and uses this modifier to implement packed structures in a more robust fashion than in earlier releases. Xavier Leroy also implemented and proved correct the optimization of integer divisions by constants introduced by Granlund and Montgomery [40].

6.1.2. Register allocation with validation a posteriori

Participant: Xavier Leroy.

Register allocation (the placement of program variables in processor registers) has a tremendous impact on the performance of compiled code. However, advanced register allocation techniques are difficult to prove correct, as they involve complex algorithms and data structures. Since the beginning of the CompCert project, we chose to avoid some of these difficult proofs by performing validation *a posteriori* for part of register allocation: the IRC graph coloring algorithm invoked during register allocation is not proved correct; instead, its results are verified at every compiler run to be a correct coloring of the given interference graph, using a simple validator proved sound in Coq.

In CompCert 2.0, we push this validation-based approach further. The whole register allocator is now subject to validation a posteriori and no longer needs to be proved correct. The validator follows the algorithm invented by Rideau and Leroy [50] and further developed by Tassarotti and Leroy. It proceeds by backward dataflow analysis of symbolic equations between program variables, registers, and stack locations.

Consequently, the new register allocator for CompCert 2.0 is much more aggressive than that of CompCert 1: it features a number of optimizations that could not be proved correct in CompCert, including live-range splitting, better handling of two-address operations and other irregularities of the x86 instruction set, an improved spilling strategy, and iterating register allocation to place temporaries introduced by spilling. Moreover, the new register allocator can handle program variables of 64-bit integer types, allocating them to pairs of 32-bit registers or stack locations. The new register allocator improves the performance of generated x86 code by up to 10% on our benchmarks.

6.1.3. *Formal verification of static analyzers based on abstract interpretation*

Participants: Sandrine Blazy [EPI Celtique], Vincent Laporte [EPI Celtique], Jacques-Henri Jourdan, Xavier Leroy, David Pichardie [EPI Celtique].

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 should be able to handle the same large subset of the C language as the CompCert compiler; support a combination of abstract domains, including relational domains; and 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 worked on numerical abstract domains for the static analyzer. First, he designed, programmed and proved correct an abstraction layer that transforms any relational abstract domain for mathematical, arbitrary-precision integers into a relational abstract domain for finite-precision machine integers, taking overflow and "wrap-around" behaviors into account. This domain transformer makes it possible to design numerical domains without taking into account the finiteness of machine integers. Then, he implemented and proved sound non-relational abstract domains for intervals of integers and of floating-point numbers, supporting almost all CompCert arithmetic operations.

In collaboration with team Celtique, we studied which intermediate languages of the CompCert C compiler are suitable as source language for the static analyzer. Early work by Blazy, Laporte, Maroneze and Pichardie [36] performs abstract interpretation over the RTL intermediate language, a simple language with unstructured control (control-flow graph). However, this language is too low-level to support reporting alarms at the level of the source C program.

Later this year, we decided to use the C#minor intermediate language of CompCert as source language for analysis. This language has mostly structured control (if/then/else, C loops, and `goto`), and is much closer to the source C program. Then, Jacques-Henri Jourdan, Xavier Leroy and David Pichardie designed a generic abstract interpreter for the C#minor language, parameterized by an abstract domain of execution states, using structured fixpoint iteration for loops and a function-global iteration for `goto`. Jacques-Henri Jourdan is in the process of proving the soundness of this abstract interpreter in Coq.

6.1.4. *Formalization of floating-point arithmetic*

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

Last year, 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. A paper describing this work was presented at the ARITH 2013 conference [15].

This year, we extended this formalization of floating-point arithmetic with a more precise modeling of "Not a Number" special numbers, reflecting the signs and payloads of these numbers into their bit-level, in-memory representation. We also proved correct more algebraic identities over FP computations, such as $x/2^n = x \times 2^{-n}$ if $|n| < 1023$, as well as nontrivial implementation schemes for conversions between integer and FP numbers, whose correctness rely on subtle properties of the "round to odd" rounding mode. These extensions are described in a draft journal paper under submission [29], and integrated in version 2.1 of CompCert.

6.1.5. Formal verification of hardware synthesis

Participants: Thomas Braibant, Adam Chlipala [MIT].

Verification of hardware designs has been thoroughly investigated. 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.

Continuing work started during a visit at MIT under the supervision of Adam Chlipala, Thomas Braibant worked on the implementation and proof of correctness of a prototype hardware compiler. This compiler produces descriptions of circuits in RTL style from a high-level description language inspired by BlueSpec. Formal verification of hardware designs of mild complexity was conducted at the source level, making it possible to obtain fully certified RTL designs. A paper describing this compiler and two examples of certified designs was presented at the CAV 2013 conference [16].

6.2. Language design and type systems

6.2.1. The Mezzo programming language

Participants: Jonathan Protzenko, François Pottier, Thibaut Balabonski, Armaël Guéneau, Cyprien Mangin.

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 aims to bring new answers to these questions.

This year, we:

- made significant progress on the proof of soundness, by rewriting it in a more modular fashion;
- improved the implementation, by formalizing the algorithms and rewriting significant parts of the type-checker;
- hosted two interns who explored arithmetic reasoning and modeling of the iterator protocol, respectively;
- formalized libraries for concurrent programming in Mezzo;
- wrote both an interpreter and a compiler for the language.

A paper on Mezzo appeared in the ICFP 2013 conference [21].

During the previous year (2012), François Pottier wrote a formal definition of Mezzo, and proved that Mezzo is type-safe: that is, well-typed programs cannot crash. The proof was machine-checked using Coq. This year, Thibaut Balabonski and François Pottier extended this formalization with support for concurrency and dynamically-allocated locks, and proved that well-typed programs not only cannot crash, but also are data-race free.

The structure of the proof was re-worked so as to make it more modular. A paper, which emphasizes this modularity, has been submitted for presentation at a conference.

The new concurrent features have been integrated in the core library of Mezzo by Thibaut Balabonski. Further concurrent libraries have been included to provide more communication primitives, such as channels for message passing.

Jonathan Protzenko worked on formalizing the type-checking algorithms currently used in the Mezzo prototype compiler. This led to practical results in the form of improvements to the type-checker: we now type-check more programs, and the success of the type-checker is more predictable as well. Some soundness bugs have been identified and fixed. The design of some of the language's features has been improved as well.

The formalization of the type-checker was presented at the IFL 2013 conference, and is to appear in the post-symposium proceedings in 2014.

We set out to promote Mezzo in the wild. Protzenko packaged the software to make it available widely via OPAM, wrote a tutorial for end-users [34], communicated through blog posts about the language, and released the source code online for others to contribute.

We also spread the word about Mezzo through various seminar talks and discussions with other teams (Carnegie-Mellon university, Cambridge Computer Lab, Aarhus University, Brasilia University), and by communicating in international conferences (ICFP'13, FSFMA'13).

This year, two interns worked with us on Mezzo. Armaël Guéneau (L3; June-July 2013) and Cyprien Mangin (M1; April-July 2013) explored several experimental aspects of the language. In particular, Armaël worked on an encoding of iterators in an object-oriented style, which involves transfers of ownership and typestate changes; while Cyprien improved the treatment of arrays and implemented an experimental extension of Mezzo with arithmetic assertions. Armaël presented his work at the workshop HOPE 2013. This work is also described in a short unpublished paper [33].

6.2.2. System F with coercion constraints

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, we extended System F with a richer type-level language and a proposition language. Propositions contain a first-order logic, a coinduction mechanism, coherence assertions and coercion assertions. Types are classified by kinds and extended in order to handle lists of types. We introduce a particular kind restricting a previous kind to its types satisfying a proposition. Abstracting over such a kind means abstracting over arbitrary propositions, and thus enables coercion abstraction. Type abstraction must be coherent: the kind of the abstract type has to be inhabited by a witness type. This language, called Fcc, extends our previous language parametric F-iota and additionally subsumes Constraint ML.

We also extended Fcc with incoherent polymorphism in order to model GADTs. Unlike coercions and thus coherent polymorphism, incoherent polymorphism is not erasable. But in counterpart, incoherent abstraction does not require the kind to be inhabited. Since abstracting over incoherent types permits to write unsound terms, incoherent abstraction has to block the reduction of terms.

This work is part of Julien Cretin's Ph.D. dissertation [11], which will be defended in January 2014.

6.2.3. Type inference for GADTs

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

Type inference for generalized algebraic data types (GADTs) is inherently non monotone: assuming more specific types for GADTs may ensure more invariants, which may result in more general types. This is problematic for type inference and some amount of type annotations is required.

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 create an ambiguity when leaving the scope of the equation: which element should be used for representing the equivalent forms? Idealy, one should use a type disjunction, but this is not allowed—for good reasons. Hence, to avoid arbitrary choices, these situations must be rejected as ambiguous, forcing the user to write more annotations to resolve the 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*, which are a way to trace unavoidable uses of equations within types themselves. We then redefined ambiguities so that only ambivalent types become ambiguous and should be rejected or resolved by a programmer annotation. Interestingly, this solution is fully compatible with unification-based type inference algorithms used in ML dialects.

This work was presented at the APLAS 2013 conference [20]. It is also implemented in the OCaml language since version 4.00.

6.2.4. 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. A paper describing our formalization was presented at the ESOP 2013 conference [23].

6.2.5. Singleton types for code inference

Participants: Gabriel Scherer, Didier Rémy.

We continued working on 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. As this is still work in progress, there was no publication on this topic this year, but we presented our directions on three occasions: at the PLUME team in ENS Lyon, at the LIX team in École Polytechnique (whose proof-search research is highly relevant to our work), and at the Dependently Typed Programming workshop (satellite of the International Conference on Functional Programming) in Boston.

6.2.6. Open closure types

Participants: Gabriel Scherer, Jan Hoffmann [Yale University, FLINT group].

During a visit to Yale, Gabriel Scherer worked with Jan Hoffmann on a type system for program analysis of higher-order functional languages. Open closure types are a novel typing construct that lets the type system statically reason about closure variables present in the lexical context. This allows fine-grained analysis (e.g., for resource consumption or information-flow control) of functional programming patterns such as function currying. This work was presented at the LPAR 2013 conference [22] (Logic for Programming, Artificial Intelligence, and Reasoning) in October.

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 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 in the previous three years. 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 have recently been pursuing two main lines of work.

We have been developing an algorithm that is able to perform dynamic load balancing in the style of work stealing but without requiring atomic read-modify-write operations. These operations may scale poorly with the number of cores due to synchronization bottlenecks. We have designed the algorithm, proved it correct using a new technique for the x86-TSO weak memory model. We have evaluated our algorithm on a modern multicore machine. Although we use no synchronization operations, we achieve performance that is no more than a few percent slower than the industrial-strengh algorithm, even though the industrial-strength algorithm takes full advantage of synchronization operations. We have a soon-to-be-submitted research article describing our contributions [25].

The design of efficient parallel graph algorithms requires a sequence data structure that supports logarithmic-time split and concatenation operations in addition to push and pop operations with excellent constant factors. We have designed such a data structure by building on a recently introduced data structure called Finger Tree and by integrating a “chunking” technique. Our chunking technique is based on instantiating the leaves of the Finger Tree with chunks of contiguous memory. Unlike previous chunked data structures, we are able to prove efficient constant factors even in worst-case scenarios. Moreover, we implemented our data structure in C++ and OCaml and showed it to be competitive with state-of-the-art sequence data structures that do not support split and concatenation operations. We are currently writing a report on our results.

6.3.2. Weak memory models

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

Modern multicore and multiprocessor 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 instruction 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 University College London) and Peter Sewell (University of Cambridge) achieved several significant results, including two semantics for the IBM Power and ARM memory models: one of the operational kind [52] and the other of the axiomatic kind [46]. 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 2013, Luc Maranget pursued this collaboration. He mainly worked with Jade Alglave to produce 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 model is at the core of a journal submission which is now at the second stage of reviewing. The submitted work contains in-depth testing of ARM devices which led to the discovery of anomalous behaviours acknowledged as such by our ARM contact, and of legitimate features now included in the model. The new model also impacted our **diy** tool suite that now includes a generic memory model simulator built by following the principles exposed in the submitted article. At the moment the new simulator is available as an experimental release (http://diy.inria.fr/herd). It will be include in future releases of the tool suite.

In the same research theme, Luc Maranget supervises the internship of Jacques-Pascal Deplaix (EPITECH), from Oct. 2013 to May 2014. The internship aims at extending **litmus**, our tool to to run tests on hardware: at the moment **litmus** accepts test written in assembler; Jacques-Pascal is extending **litmus** so that it accepts tests written in C. The general objective is to achieve conformance testing of C compilers and machines with respect to the new C11/C++11 standard.

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, Gabriel Scherer.

This year, we released version 4.01.0 of the OCaml system. This is a major release that fixes about 140 bugs and introduces 44 new features suggested by users. Damien Doligez acted as release manager for this version.

The major innovations in OCaml 4.01 are:

- The overloading of variant constructors and record field labels, resolved using typing information. Before this, programmers had to use globally unique field labels across all record types. The new typechecking algorithm enables programmers to use more natural names for fields in their data structures. The algorithm is carefully engineered to preserve principality of inferred types.
- New warnings give the programmer the option of applying very strict checking of problematic constructs in the source code.

Other features of this release include:

- Suggestion of possible typos in case of “unbound identifier” error.
- New infix application operators in the standard library.
- Options to reduce the verbosity (and enhance the readability) of error messages.
- Many internal improvements, especially in compiler performance.

In parallel, we designed and experimented with several new features that are candidate for inclusion in the next major release of OCaml in 2014:

- Module aliases: a more efficient way of typechecking and compiling module declarations of the form `module M = ModuleName`, providing a lighter, more practical alternative to packed modules and reducing the need for name spaces.
- Extension points and preprocessing by rewriting abstract syntax trees: this approach provides an alternative to Camlp4 for macro processing and automatic code generation.
- A native code generator for the new ARM 64 bit instruction set (also known as AArch64).
- Several ongoing experiments to improve the performance of OCaml-compiled code: more aggressive function inlining and constant propagation; more unboxing of numbers; and a pass of common subexpression elimination.

6.4.2. Run-time types for the OCaml language

Participants: Grégoire Henry, Jacques Garrigue [University of Nagoya], Fabrice Le Fessant.

With the addition of GADTs to OCaml in version 4.00, it is now possible to provide a clean implementation of run-time types in the language, thus allowing the definition of polytypic function, a.k.a. generic function defined by case analysis on the structure of its argument’s type. However, when integrating this mechanism into the language, its interaction with other parts of the type-system proved delicate, the main difficulty being the semantic of abstract types.

In collaboration with Jacques Garrigue during a 3 month stay in Japan, Grégoire Henry worked on different semantics for the runtime representation of abstract types. They tried to design a mechanism that preserves abstraction by default, and still allows to propagate type information when requested by the programmer.

6.4.3. Multi-runtime OCaml

Participants: Luca Saiu, Fabrice Le Fessant.

Multicore architectures are now broadly available, and developers expect their programs to be able to benefit from them. In OCaml, there is no portable way to use such architectures, as only one OCaml thread can run at any time.

As part of the ANR project “BWare”, Luca Saiu and Fabrice Le Fessant developed a multi-runtime version of OCaml that takes advantage of multicore architectures. In this version, a program can start several runtimes that can run on different cores. As a consequence, OCaml threads running on different runtimes can run concurrently. This implementation required a lot of rewriting of the OCaml runtime system (written in C), to make all global variables context-dependent and all functions reentrant. The compiler was also modified to generate reentrant code and context-dependent variables. The sources of the prototype were released in September 2013, to be tested by users.

Luca Saiu then developed a library based on skeletons to facilitate the development of parallel applications that take advantage of the multi-runtime architecture.

6.4.4. Evaluation strategies and standardization

Participants: Thibaut Balabonski, Flávio de Moura [Universidade de Brasília].

During the past years, Thibaut Balabonski studied evaluation strategies, laziness and optimality for functional programming languages, in particular in relation to pattern matching. These investigations continued this year, with two highlights:

- Publication in the ICFP conference [14] of a theoretical result relating fully lazy evaluation (as can be found in some Haskell compilers) to optimal reduction in the weak λ-calculus.
- Collaboration with Flávio de Moura (Universidade de Brasília) on so-called “standard” evaluation strategies for a calculus with rich pattern matching mechanisms (the *Pure Pattern Calculus* of Jay and Kesner [42]). The challenge here lies in that the calculus does not satisfies the usual stability properties. As a consequences, standard strategies are not unique anymore, and new approaches are needed. A paper is in preparation.

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

This year, the TLA+ tools were released as open-source (MIT license), and in September we released a new version of the TLA+ Proof System (TLAPS), an environment for writing and checking TLA+ proofs. This environment is described in [38].

We have implemented a (not yet released) extension of TLAPS to deal with proofs of temporal formulas, using the propositional temporal logic prover LS4 as a back-end. Until now, TLAPS could only be used to prove safety properties (invariants). With this new version, our users will be able to prove liveness properties (absence of deadlock), refinement relations between specifications, etc.

Jael Kriener started a 2-year post-doc contract in December. She is working on theoretical and implementation aspects of TLA+ and TLAPS.

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 [CNAM], Olivier Hermant [Mines ParisTech], Mélanie Jacquel [CNAM].

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

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 [37]. Mélanie Jacquel defended her thesis on this subject in April.

Pierre Halmagrand did an internship and started a thesis on integrating Deduction Modulo in Zenon; some results of this work are described in two papers published at LPAR [19] and IWIL [18].

6.5.3. 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 make 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. In a joint article at ITP 2013 [17], we described three different implementation techniques for hash-consed data-structures in Coq through the running example of Binary Decision Diagrams (BDDs). BDDs are representations of Boolean functions, and are often used in software and hardware verification tools (e.g., model checkers).

We substantially improved the work described in this ITP 2013 article afterwards. First, we came up with a fourth implementation technique for 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. This work is currently under revision for publication in a journal.

6.5.4. Working with names and binders

Participant: François Pottier.

François Pottier released **dblib**, a Coq library that helps work with de Bruijn indices in a generic and lightweight manner. This library is used in the formalization of Mezzo (see section 6.2.1). It is available at http://gallium.inria.fr/~fpottier/.

6.6. Technology transfer

6.6.1. Analysis of the Scilab Language

Participants: Fabrice Le Fessant, Michael Laporte.

The Scilab language is a scripting language providing easy access to efficient implementations of mathematical operations (on matrices, for example). It suffers from the lack of verifications of an untyped language, together with the performance problems of an interpreted language. As part of the FUI Richelieu project, Fabrice Le Fessant and Michael Laporte have been investigating solutions to these issues.

The first part of the work was to clarify the semantics of the Scilab language. For that, an interpreter was implemented in OCaml, based on the C++ AST provided by the forthcoming version 6 of Scilab. This work exhibited a number of bugs in the new implementation, and proved to be more performant than the C++ implementation, thanks to a better algorithm to manage the dynamic scopes of Scilab.

The second part of the work was to understand how users write Scilab code. For that, a style-checking application, called *Scilint*, has been developed. It implements static checking of some properties of Scilab programs, to be able to detect runtime errors before running the program. Warnings are displayed for suspicious cases. Using Scilint on large sets of Scilab code (from the Scilab forge or the Atom repository) showed that the most erroneous features of Scilab are commonly used and that, to achieve the ultimate goal of partial typing of the language, a subset of the language must be specified that the user should conform to, in order for the code to benefit from the next part of the work, i.e. just-in-time compilation.

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 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 11 member companies:

- CEA
- Citrix
- Dassault Aviation
- Dassault Systèmes
- Esterel Technologies
- Jane Street
- LexiFi
- Microsoft
- Mylife.com
- 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.

7.1.2. OCamlPro

Participant: Fabrice Le Fessant.

Fabrice Le Fessant is consulting for OCamlPro, a SME that provides services and tools to companies wanting to use OCaml as their development language.

8. Partnerships and Cooperations

8.1. National Initiatives

8.1.1. ANR projects

8.1.1.1. BWare

Participants: Damien Doligez, Fabrice Le Fessant, Luca Saiu.

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 BGLE 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, 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 (FUI)

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 Projects

8.2.1.1. DEEPSEA

Type: IDEAS

Instrument: ERC Starting Grant

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 Labs

Fabrice Le Fessant visited CIRIC (Center of Excellence on TIC, created by Inria in Chile) during two weeks. He gave several lectures on OCaml: a presentation at StarTechConf'2013, a presentation at University Adolfo Ibañez, and a presentation and a lecture at University of Chile.

8.4. International Research Visitors

8.4.1. Visits of International Scientists

Olin Shivers, professor at Northeastern University (Boston), visited the Gallium team from July 2013 to December 2013. He worked on static analysis and intermediate representations for functional programming languages.

8.4.1.1. Internships

Robbert Krebbers

Subject: formal semantics for the C language

Date: from Jan 2013 until Mar 2013

Institution: Radboud University (Netherlands)

9. Dissemination

9.1. Scientific Animation

9.1.1. Conference organization

Didier Rémy organized the October 2013 meeting of IFIP working group 2.8 “Functional programming”, which took place in Aussois, France.

Thomas Braibant participated in the organization of the LOLA 2013 workshop, associated with LICS 2013.

9.1.2. Editorial boards

Xavier Leroy is on the editorial board for the Research Highlights column of Communications of the ACM. He 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

Xavier Leroy was a member of the program committee for VSTTE 2013, the conference on Verified Software: Theory, Tools and Experiments.

François Pottier was a member of the program committee for ESOP 2014, the European Symposium On Programming.

9.1.4. Steering committees

Xavier Leroy is a member of the steering committees for the Certified Programming and Proofs (CPP) conference and the Programming Languages meet Program Verification (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 Workshop.

9.1.5. Collective responsibilities

Damien Doligez chairs the *Commission des actions de développement technologiques* of Inria Paris-Rocquencourt.

Xavier Leroy is *vice-président du comité des projets* of Inria Paris-Rocquencourt and appointed member of *Commission d'Évaluation*. He participated in the following Inria hiring and promotion committees: *jury d'admissibilité CR2 Paris-Rocquencourt* (vice-chair, with Philippe Robert as chair); *jury d'admissibilité DR2*; *promotions CR1, DR1, DR0*. He was a member of the hiring committee for a *Maître de conférences* position at Université Rennes 1.

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

François Pottier is a member of the post-doctoral hiring committee of Inria Paris-Rocquencourt. He was a member of the hiring committee for a *Maître de conférences* position at Université Paris Diderot.

Jonathan Protzenko curated the Junior Seminar of Inria Paris-Rocquencourt until June 2013, which marked the end of his two-year involvement in the seminar.

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.

9.2. Teaching - Supervision - Juries

9.2.1. Teaching

Licence: Thibaut Balabonski, "Travaux dirigés de Caml Light", 14 hours, L1, Collège Stanislas (classes préparatoires MPSI), France.

Licence: Julien Cretin, "Bases de données", 26h, L3, U. Paris Diderot, France.

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

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, "IF1: Introduction to computer science and programming", 42h, L1, U. Paris Diderot, France.

Master: Xavier Leroy and Didier Rémy, "Functional programming and type systems", 12h + 18h, M2, MPRI master, France.

Master: Luc Maranget, "Semantics, languages and algorithms for multicore programming", 9h, M2, MPRI master, France.

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

Master: Jonathan Protzenko, “Conception et mise en œuvre d’algorithmes” (MOOC), 32h, M1, Coursera / École Polytechnique, France.

Master: Gabriel Scherer, “Advanced Functional Programming”, 30h, M1, U. Paris Diderot, France.

Doctorat: Xavier Leroy, “Mechanized semantics”, 6h, Verification Technology, Systems & Applications summer school 2013, Nancy, France.

9.2.2. Supervision

PhD in progress: Julien Cretin, “Erasable coercions: a unified approach to type systems”, École Polytechnique, since December 2010, supervised by Didier Rémy, to be defended January 30th, 2014.

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

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, “Term inference”, 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.

9.2.3. Juries

Damien Doligez was a member of the Ph.D. jury of Mélanie Jacquel, CNAM, Paris, april 2013.

Xavier Leroy was a member of the Ph.D. jury of Xiaomu Shi, Université Joseph Fourier, Grenoble, july 2013. Xavier Leroy was president of the Ph.D. jury of Pierre-Nicolas Tollitte, CNAM, Paris, december 2013.

9.3. Popularization

Jacques-Henri Jourdan and Arthur Charguéraud 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 170,000 participants.

Fabrice Le Fessant is one of the organizers of the OCaml meetup in Paris. Four events were organized in 2013, each featuring four short presentations on topics related to OCaml. Each event was attended by about 60 participants.

Xavier Leroy gave a tutorial on using theorem provers in programming language research at the 2013 ACM SIGPLAN Programming Languages Mentoring Workshop, which was attended by about 80 undergraduate, graduate and post-doctoral students.

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 2013, with 26 posts by 12 different authors. It covered various changes in the OCaml language, announced small software libraries from members of the team, and discussed Gallium’s research, notably the Mezzo language.

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, pp. 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, pp. 142–148, http://dx.doi.org/10.1007/978-3-642-14203-1_12

[3] 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

[4] D. Le Botlan, D. Rémy. *Recasting MLF*, in "Information and Computation", 2009, vol. 207, n^o 6, pp. 726–785, http://dx.doi.org/10.1016/j.ic.2008.12.006

[5] X. Leroy. *A formally verified compiler back-end*, in "Journal of Automated Reasoning", 2009, vol. 43, n^o 4, pp. 363–446, http://dx.doi.org/10.1007/s10817-009-9155-4

[6] X. Leroy. *Formal verification of a realistic compiler*, in "Communications of the ACM", 2009, vol. 52, n^o 7, pp. 107–115, http://doi.acm.org/10.1145/1538788.1538814

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

[8] F. Pottier, 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. , *Coercions effaçables : une approche unifiée des systèmes de types*, Université Paris-Diderot - Paris VII, January 2014, http://hal.inria.fr/tel-00940511

Articles in International Peer-Reviewed Journals

[12] F. Pottier. *Syntactic soundness proof of a type-and-capability system with hidden state*, in "Journal of Functional Programming", January 2013, vol. 23, n^o 1, pp. 38-144 [*DOI :* 10.1017/S0956796812000366], http://hal.inria.fr/hal-00877589

[13] 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^o 1, pp. 1–54, http://hal.inria.fr/hal-00772757

International Conferences with Proceedings

[14] T. BALABONSKI. *Weak Optimality, and the Meaning of Sharing*, in "International Conference on Functional Programming (ICFP)", Boston, United States, September 2013, pp. 263-274 [*DOI :* 10.1145/2500365.2500606], http://hal.inria.fr/hal-00907056

[15] S. BOLDO, J.-H. JOURDAN, X. LEROY, G. MELQUIOND. *A Formally-Verified C Compiler Supporting Floating-Point Arithmetic*, in "Arith - 21st IEEE Symposium on Computer Arithmetic", Austin, United States, A. NANNARELLI, P.-M. SEIDEL, P. T. P. TANG (editors), IEEE, 2013, pp. 107-115, http://hal.inria.fr/hal-00743090

[16] T. BRAIBANT, A. CHLIPALA. *Formal Verification of Hardware Synthesis*, in "Computer Aided Verification - 25th International Conference", Saint Petersburg, Russian Federation, N. SHARYGINA, H. VEITH (editors), Lecture notes in computer science, Springer, 2013, vol. 8044, pp. 213-228 [*DOI :* 10.1007/978-3-642-39799-8_14], http://hal.inria.fr/hal-00776876

[17] T. BRAIBANT, J.-H. JOURDAN, D. MONNIAUX. *Implementing hash-consed structures in Coq*, in "Interactive Theorem Proving, 4th international conference", Rennes, France, S. BLAZY, C. PAULIN-MOHRING, D. PICHARDIE (editors), Lecture notes in computer science, Springer, July 2013, vol. 7998, pp. 477-483 [*DOI :* 10.1007/978-3-642-39634-2_36], http://hal.inria.fr/hal-00816672

[18] D. DELAHAYE, D. DOLIGEZ, F. GILBERT, P. HALMAGRAND, O. HERMANT. *Proof Certification in Zenon Modulo: When Achilles Uses Deduction Modulo to Outrun the Tortoise with Shorter Steps*, in "IWIL - 10th International Workshop on the Implementation of Logics - 2013", Stellenbosch, South Africa, S. SCHULZ, G. SUTCLIFFE, B. KONEV (editors), EasyChair, December 2013, http://hal.inria.fr/hal-00909688

[19] D. DELAHAYE, D. DOLIGEZ, F. GILBERT, P. HALMAGRAND, O. HERMANT. *Zenon Modulo: When Achilles Outruns the Tortoise using Deduction Modulo*, in "LPAR - Logic for Programming Artificial Intelligence and Reasoning - 2013", Stellenbosch, South Africa, K. MCMILLAN, A. MIDDELDORP, A. VORONKOV (editors), LNCS, Springer, December 2013, vol. 8312, pp. 274-290 [*DOI :* 10.1007/978-3-642-45221-5_20], http://hal.inria.fr/hal-00909784

[20] J. GARRIGUE, D. RÉMY. *Ambivalent Types for Principal Type Inference with GADTs*, in "APLAS 2013 - 11th Asian Symposium on Programming Languages and Systems", Melbourne, Australia, CHUNG-CHIEH. SHAN (editor), Lecture Notes in Computer Science, December 2013, vol. 8301, pp. 257-272 [*DOI :* 10.1007/978-3-319-03542-0_19], http://hal.inria.fr/hal-00914560

[21] F. POTTIER, J. PROTZENKO. *Programming with permissions in Mezzo*, in "ICFP - The 18th ACM SIGPLAN International Conference on Functional Programming - 2013", Boston, United States, September 2013, pp. 173-184 [*DOI :* 10.1145/2500365.2500598], http://hal.inria.fr/hal-00877590

[22] G. SCHERER, J. HOFFMANN. *Tracking Data-Flow with Open Closure Types*, in "LPAR- 19th International Conference Logic for Programming, Artificial Intelligence, and Reasoning", Stellenbosch, South Africa, K. MCMILLAN, A. MIDDELDORP, A. VORONKOV (editors), Lecture Notes in Computer Science, Springer Verlag, October 2013, vol. 8312, pp. 710-726, http://hal.inria.fr/hal-00911656

[23] G. SCHERER, D. RÉMY. *GADTs meet subtyping*, in "ESOP 2013 - 22nd European Symposium on Programming", Rome, Italy, M. FELLEISEN, P. GARDNER (editors), Lecture Notes in Computer Science, Springer, January 2013, vol. 7792, pp. 554-573 [*DOI :* 10.1007/978-3-642-37036-6], http://hal.inria.fr/hal-00772993

Scientific Books (or Scientific Book chapters)

[24] 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, http://hal.inria.fr/hal-00905435

Research Reports

[25] U. ACAR, A. CHARGUÉRAUD, S. MULLER, M. RAINEY. , *Atomic Read-Modify-Write Operations are Unnecessary for Shared-Memory Work Stealing*, September 2013, http://hal.inria.fr/hal-00910130

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

[27] X. LEROY, D. DOLIGEZ, A. FRISCH, J. GARRIGUE, D. RÉMY, J. VOUILLON. , *The OCaml system release 4.01: Documentation and user's manual*, September 2013, http://hal.inria.fr/hal-00930213

[28] G. SCHERER, J. HOFFMANN. , *Tracking Data-Flow with Open Closure Types*, Inria, August 2013, n^o RR-8345, 24 p. , http://hal.inria.fr/hal-00851658

Other Publications

[29] S. BOLDO, J.-H. JOURDAN, X. LEROY, G. MELQUIOND. , *A Formally-Verified C Compiler Supporting Floating-Point Arithmetic*, 2013, http://hal.inria.fr/hal-00862689

[30] T. BRAIBANT, J.-H. JOURDAN, D. MONNIAUX. , *Implementing and reasoning about hash-consed data structures in Coq*, November 2013, http://hal.inria.fr/hal-00881085

[31] P.-E. DAGAND, C. MCBRIDE. , *Transporting Functions across Ornaments*, December 2013, Under submission to "Journal of Functional Programming", http://hal.inria.fr/hal-00922581

[32] J. GARRIGUE, D. RÉMY. , *Ambivalent Types for Principal Type Inference with GADTs (extended version)*, 2013, http://hal.inria.fr/hal-00914493

[33] A. GUÉNEAU, F. POTTIER, J. PROTZENKO. , *The ins and outs of iteration in Mezzo*, 2013, http://hal.inria.fr/hal-00912381

[34] J. PROTZENKO. , *Illustrating the Mezzo programming language*, 2013, http://hal.inria.fr/hal-00910402

References in notes

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

[36] S. BLAZY, V. LAPORTE, A. MARONEZE, D. PICHARDIE. *Formal Verification of a C Value Analysis Based on Abstract Interpretation*, in "Static Analysis - 20th International Symposium, SAS 2013", Lecture Notes in Computer Science, Springer, 2013, vol. 7935, pp. 324-344

[37] 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, pp. 41-50, http://hal.inria.fr/inria-00133557

[38] 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

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

[40] T. GRANLUND, P. L. MONTGOMERY. *Division by Invariant Integers using Multiplication*, in "Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI)", ACM, 1994, pp. 61-72

[41] 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

[42] C. B. JAY, D. KESNER. *First-class patterns*, in "J. Functional Programming", 2009, vol. 19, n^{o} 2, pp. 191-225

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

[44] D. LE BOTLAN, D. RÉMY. *MLF: Raising ML to the power of System F*, in "Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming", ACM Press, August 2003, pp. 27–38, http://gallium.inria.fr/~remy/work/mlf/icfp.pdf

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

[46] 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 "Computer Aided Verification - 24th International Conference, CAV 2012", Lecture Notes in Computer Science, Springer, 2012, vol. 7358, pp. 495-512

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

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

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

[50] S. RIDEAU, X. LEROY. *Validating register allocation and spilling*, in "Compiler Construction (CC 2010)", Lecture Notes in Computer Science, Springer, 2010, vol. 6011, pp. 224–243, http://dx.doi.org/10.1007/978-3-642-11970-5_13

[51] 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

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