INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Project-Team Gallium

Programming languages, types, compilation and proofs

Paris - Rocquencourt

Theme : Programs, Verification and Proofs

Table of contents

1. Team

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]

Faculty Member

Roberto Di Cosmo [Professor, on leave from U. Paris Diderot, HdR]

Technical Staff

Xavier Clerc [IR INRIA, SED, 20% part time]

PhD Students

Arthur Charguéraud [AMN grant, U. Paris Diderot]
Julien Cretin [AMX grant, U. Paris Diderot, since December 2010]
Benoît Montagu [AMX grant, Polytechnique]
Alexandre Pilkiewicz [AMX grant, Polytechnique]
Nicolas Pouillard [Digiteo grant, INRIA]
Jonathan Protzenko [ENS Lyon student, U. Paris Diderot, since September 2010]
Tahina Ramananandro [AMN grant, U. Paris Diderot]

Visiting Scientist

Brigitte Pientka [Professor, on sabbatical from Mc Gill U., January-May 2010]

Administrative Assistant

Stéphanie Chaix [Temporary personnel]

Others

Julien Cretin [M2 graduate intern, July–November 2010]
Vivien Maisonneuve [M2 graduate intern, March–July 2010]
Gabriel Scherer [M1 graduate intern, April–July 2010]

2. Overall Objectives

2.1. Overall Objectives

The research conducted in the Gallium group aims at improving the safety, reliability and security of software through advances in programming languages and formal verification of programs. Our work is centered on the design, formalization and implementation of functional programming languages, with particular emphasis on type systems and type inference, formal verification of compilers, and interactions between programming and program proof. The Caml language and the CompCert verified C compiler embody many of our research results. Our work spans the whole spectrum from theoretical foundations and formal semantics to applications to real-world problems.

3. 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 Objective Caml language and system embodies many of our earlier results in this area [33]. 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 [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 (mis-spelled 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], [38], 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 Objective Caml, 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. Objective Caml 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 participated in the Focal project, which designed and implemented an environment for combined programming and proving [50].

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 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 [44] 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 de-structure 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*. Objective Caml is also widely used for teaching advanced programming in engineering schools, colleges and universities in France, USA, and Japan.

5. Software

5.1. Objective Caml

Participants: Xavier Leroy [correspondant], Xavier Clerc [team SED], Damien Doligez, Alain Frisch [LexiFi], Jacques Garrigue [Nagoya University], Maxence Guesdon [team SED], Luc Maranget [EPI Moscova], Michel Mauny [ENSTA], Nicolas Pouillard, Pierre Weis [EPI Estime].

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 Objective Caml 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 Objective Caml 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], Alexandre Pilkiewicz.

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 retargetted to output scripts for different frameworks (for example, Isabelle).

Web site: http://focal.inria.fr/zenon/.

5.4. Menhir

Participants: François Pottier [correspondant], Yann Régis-Gianas [U. Paris Diderot].

Menhir is a new LR(1) parser generator for Objective Caml. Menhir improves on its predecessor, `ocamlyacc`, in many ways: more expressive language of grammars, including EBNF syntax and the ability to parameterize a non-terminal by other symbols; support for full LR(1) parsing, not just LALR(1); ability to explain conflicts in terms of the grammar.

Web site: http://gallium.inria.fr/~fpottier/menhir/.

6. New Results

6.1. Program specification and program proof

6.1.1. Characteristic formulae for interactive program verification

Participant: Arthur Charguéraud.

Arthur Charguéraud has developed a new approach to program verification, based on *characteristic formulae*. The characteristic formula of a program is a higher-order logic formula that describes the behavior of that program, in the sense that it is sound and complete with respect to the semantics. This formula can be exploited in an interactive theorem prover to establish that a program satisfies a specification expressed in the style of Separation Logic, with respect to total correctness.

The characteristic formula of a program is automatically generated from its source code alone. In particular, there is no need to annotate the source code with specifications or loop invariants, as such information can be given in interactive proof scripts. Characteristic formulae, which are expressed in terms of basic logical connectives, can be pretty-printed so as to closely resemble the source code that they describe, even though they do not refer to the syntax of the programming language. Thanks to this pretty-printing mechanism, proof obligations are easily readable.

Characteristic formulae serve as a basis for a tool, called CFML, that supports the verification of Caml programs using the Coq proof assistant. More precisely, CFML supports higher-order functions, recursion, mutual recursion, polymorphic recursion, tuples, data constructors, pattern matching, references, records and arrays. This tool has been applied to the verification of a number of purely-functional data structures and of imperative higher-order functions.

A paper describing characteristic formulae for purely-functional programs was published at ICFP 2010 [16]. Their generalization to imperative programs is described in Arthur Charguéraud's PhD thesis [11].

6.1.2. A generic fixed point combinator for Coq

Participant: Arthur Charguéraud.

In a theorem prover such as Coq or Isabelle/HOL, recursive functions must terminate on all arguments, otherwise the soundness of the logic would be compromised. Similarly, a productivity requirement applies to co-recursive functions (i.e., recursive functions over co-inductive data structures) and to co-recursive values. In Coq, the termination and productivity requirements are enforced through syntactic analyses. Those analyses are inherently limited, making it difficult to formalize nontrivial circular definitions.

Building on the theory of optimal fixed points of Manna and Shamir [45], as well as on contraction conditions for co-recursive definitions [46], Arthur Charguéraud developed a generic fixed point combinator. This "optimal fixed point combinator", which can be defined in higher-order logic equipped with Hilbert's epsilon operator, allows for a direct and effective formalization of advanced definitions involving recursion, co-recursion, or both at the same time. In particular, the combinator supports higher-order recursion and nested recursion, and it offers a proper treatment of partial functions in the sense that domains need not be hard-wired in the definition of functionals.

The optimal fixed point combinator is described in a paper presented at ITP 2010 [17]. The development is entirely formalized in a Coq library, which provides a practical way to formalize advanced circular definitions in Coq.

6.1.3. The Zenon automatic theorem prover

Participant: Damien Doligez.

Damien Doligez continued the development of Zenon, a tableau-based prover for first-order logic with equality and theory-specific extensions. This year, Zenon was extended to handle TLA+ records, tuples, and sequences. The Isabelle back-end was tuned for proofs involving strings, integers, and CASE expressions. This resulted in speed-up of the time taken by Isabelle to check these proofs. Zenon version 0.6.3 was released in February.

6.1.4. Tools for TLA+

Participants: Damien Doligez, Leslie Lamport [Microsoft Research], Stephan Merz [EPI Mosel], Denis Cousineau [Microsoft Research-INRIA Joint Centre], Dan Ricketts [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 [42], and to build tools for writing TLA+ specifications and mechanically checking the corresponding formal proofs.

This year, the TLA+ project released the second version of the TLA+ tools: the GUI-based TLA Toolbox and the TLA+ Proof System, an environment for writing and checking TLA+ proofs. In this new version, the Toolbox and the Proof System interact to provide an IDE for developing proofs, with folding of proof subtrees, coloring of subgoals depending on their status (proved, failed, unknown), clickable error reports, etc. The system is described in a paper presented at IJCAR 2010 [18].

6.2. Formal verification of compilers

6.2.1. The Compcert verified compiler for the C language

Participants: Xavier Leroy, Sandrine Blazy [EPI Celtique], Alexandre Pilkiewicz.

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 [4]. This compiler comprises a back-end part, translating the Cminor intermediate language to assembly code, reusable for source languages other than C [3], and a front-end translating the CompCert C subset of C to Cminor. The compiler is mostly written within the specification language of the Coq proof assistant, from which Coq's extraction facility generates executable Caml code. The compiler comes with a 50000-line, machine-checked Coq proof of semantic preservation establishing that the generated assembly code executes exactly as prescribed by the semantics of the source C program.

This year, we improved the Compcert C compiler in several ways:

- The input language to the proved part of the compiler was extended to support side-effects within expressions. The formal semantics for this language is non-deterministic, as it accounts for the partially unspecified evaluation order of C expressions. The transformations that pull side effects out of expressions and materialize implicit casts, formerly performed by untrusted Caml code, are now fully proved in Coq.
- A port targeting Intel/AMD x86 processors was added to the two existing ports for PowerPC and ARM. The new port generates 32-bit x86 code using SSE2 extensions for floating-point arithmetic. CompCert's compilation strategy is not a very good match for the x86 architecture, therefore the performance of the generated code is not as good as for the PowerPC port, but still usable. (About 75% of the performance of `gcc -O1` for x86, compared with more than 90% for PowerPC.)
- The operational semantics for accesses to `volatile`-qualified variables was revised to capture more precisely their intended behavior. In particular, volatile reads and writes from/to a volatile global variable are treated like input and output system calls, respectively, bypassing the memory model entirely.
- The performance of the generated code was improved, in particular via a better treatment of spilled temporaries during register allocation.
- Compilation times were reduced thanks to several algorithmic improvements in the optimization passes based on dataflow analysis.

Three versions of the CompCert development were publically released: versions 1.7 in March, 1.7.1 in April, and 1.8 in September.

Several of these improvements were prompted by the result of an experimental study of CompCert's usability conducted at Airbus by Ricardo Bedin França under the supervision of Denis Favre-Felix, Marc Pantel and Jean Souyris. Preliminary results are reported in an article to be presented at the 2011 workshop on Predictability and Performance in Embedded Systems [15].

6.2.2. Verified compilation of C++

Participants: Tahina Ramananandro, Gabriel Dos Reis [Texas A&M University], Xavier Leroy.

Object layout and management, including dynamic allocation, field resolution, method dispatch and type casts, is a critical part of the compilation and runtime systems of object-oriented languages such as Java or C++. Formal verification of this part needs relating an abstract formalization of object operations at the level of the source language semantics with a concrete representation of objects in the memory model provided by the target low-level language. As this work heavily uses pointer arithmetic, the proofs must be treated with specific methods.

This year, under Xavier Leroy's supervision and with precious C++ advice from Gabriel Dos Reis, Tahina Ramananandro tackled the issue of formally verifying object layout and management in multiple-inheritance languages, especially the C++ flavour featuring non-virtual and virtual inheritance (allowing repeated and shared base class subobjects), and also structure array fields. This is a step towards building a formally verified compiler from an object-oriented subset of C++ to RTL (a CFG-style intermediate language of the

CompCert back-end). This formalization consists in proving, in Coq, the correctness of a family of object layout algorithms (including one popular algorithm inspired from the Common Vendor ABI for Itanium, which has since been reused and adapted by GNU GCC) with respect to the formal operational semantics of an object-oriented subset of C++ featuring static and dynamic casts, and field accesses. These results were accepted for publication at the forthcoming POPL 2011 symposium [24].

Then, Tahina Ramananandro has been reusing and extending this work to formalize object construction and destruction, especially their impact on the behaviour of dynamic operations such as virtual function dispatch, and the consequences of such changes on a realistic compiler implementation based on virtual tables (again inspired from the aforementioned Common Vendor ABI for Itanium).

6.2.3. Optimizations in the CompCert C compiler

Participant: Alexandre Pilkiewicz.

Alexandre Pilkiewicz made some experiments on optimizations implementation in the CompCert C compiler. The first was a fully proved implementation of a mixed data-flow analyzer/code transformer in the style of Lerner, Grove and Chamber [43] and Hoopl [51]. Even if this allowed a combined implementation of constant propagation and common sub-expressions elimination, it was too limited to be of real interest since it only allows to replace one instruction by exactly one other instruction, and not to work at the level of basic blocks.

Another experiment was an implementation of the global value numbering algorithm by Gulwani and Necula [40], but the proof is not finished yet.

Finally, Alexandre Pilkiewicz implemented a function inlining pass over the Cminor intermediate language of CompCert. Its correctness proof is a work in progress.

6.2.4. Validation of polyhedral optimizations

Participants: Alexandre Pilkiewicz, François Pottier.

Numerical codes make heavy use of nested loops. Those nests can be optimized for data locality (reducing the number of cache miss) or automatic parallelization. One of the ways to do that is to represent the loop nest as a multidimensional polyhedron, where the program is just a particular scheduling of a walk over the polyhedron. The optimization process can then be summarized as finding a better schedule that respect the initial constraints (for example, one cannot interchange the assignments `t[0] = 1;` and `t[0] = 42;`).

Finding such a valid schedule relies on subtle heuristics and optimized C libraries for polyhedron manipulation. They are, therefore, prone to error. Alexandre Pilkiewicz, under François Pottier's supervision, works on developing and proving in Coq a *validator* of such optimizations. The idea is to check a posteriori and at each run of the optimizer that the produced program is equivalent to the original one. This work in progress is done in collaboration with Nicolas Magaud, Julien Narboux and Eric Violard of the INRIA Camus team.

6.3. First-class module systems

Participants: Benoît Montagu, Didier Rémy.

Advanced module systems have now been in use for two decades in modern, statically typed languages. Modules are easy to understand intuitively and also easy to use in simple cases. However, they remain surprisingly hard to formalize and also often become harder to use in larger, more complex but practical examples. In fact, useful features such as recursive modules or mixins are technically challenging and still an active topic of research.

This persisting gap between the apparent simplicity and formal complexity of modules is surprising. We have identified at least two orthogonal sources of width-wise and depth-wise complexity. On the one hand, the stratified presentation of modules as a small calculus of functions and records on top of the underlying base language duplicates the base constructs and therefore complicates the language as a whole. On the other hand, the use of paths to designate abstract types relatively to value variables so as to keep track of sharing pulls the whole not-so-simple formalism of dependent types, even though only a very limited form of dependent types is effectively used.

Our goal is to provide a new presentation of modules that is conceptually more economical while retaining (or increasing) the expressiveness and conciseness of the actual approaches. We rely on first-class modules to avoid duplications of constructs, a new form of open existential types to represent type abstraction, and the theory of singleton kinds to handle type definitions and to preserve the conciseness of writing.

Open existential types improve over existential types with a novel, open-scoped, unpacking construct that is the essence of type abstraction in modules, and can also easily handle recursive modules. This work was presented at the POPL 2009 symposium [5]. The soundness proof of the core system was mechanically verified with the Coq proof assistant. The formal proof and the issues related to the technology that was used to deal with binders were presented at the Workshop on Mechanizing Metatheory in Baltimore in September 2010 [31]. The Coq development is publicly available at http://gallium.inria.fr/~montagu/proofs/FzipCore/FzipCore.tar.bz2.

Benoît Montagu studied systems equipped with singleton kinds, and gave a characterization of type equivalence, based on $\beta\eta$-equivalence and proved correct and complete. As in other works on this topic, the unfolding of type definitions is understood as an η-expansion step at a singleton kind. The novelty of the work lies in the definition of a small-step reduction relation that features β-reduction and η-expansion, while remaining confluent and strongly normalizing on wellformed types. The key ingredient is making the η-expansion points explicit in the syntax. This characterization has the advantage of being more flexible and easier to understand than the known algorithms for computing normal forms and deciding equivalence. It can also lead to simpler algorithms.

Benoît Montagu defined a language, called Fzip-full, that features open existential types, singleton kinds, and subtyping, and that can serve as an explicitly-typed core language for modules. He described a translation from a module language to Fzip-full, that shows the similar degree of verbosity of the two languages. A basic typechecker for Fzip-full was developed. It is available at http://gallium.inria.fr/~montagu/thesis/fzip.tar.bz2.

This work is fully described in Benoît Montagu's Ph.D. thesis [12], which was defended in December 2010.

6.4. Type systems and type inference

6.4.1. Leveraging constraint-based type inference

Participants: Jonathan Protzenko, François Pottier.

Jonathan Protzenko, under the supervision of François Pottier, improved constraint-based type inference. Instead of just checking that a program is typeable, he now generates a typed AST. This allows to translate the source program into a core language, namely System F^η augmented with explicit coercions. The benefits are threefold: some features of the input language are now precisely described in terms of core constructions; the resulting core program can be type-checked to ensure the consistency of the whole process; this provides a starting point for the rest of the compilation process. This work is a first building block towards a more trustworthy compiler for Caml.

6.4.2. Partial type inference with first-class polymorphism

Participants: Didier Rémy, Boris Yakobowski [CEA LIST], Gabriel Scherer.

The ML language uses simple types (first-order types without quantifiers) enriched with type schemes (simple types with outer-most universal quantifiers). This allows for simple type inference based on first-order unification, relieving the user from the burden of writing type annotations. However, it only enables a limited form of parametric polymorphism. In contrast, System F uses second-order types (types with inner universal quantifiers at arbitrary depth) that are much more expressive. As a result, type inference is undecidable in System F which forces the user to provide all type annotations.

Didier Le Botlan and Didier Rémy have proposed a type system, called MLF, which enables type synthesis as in ML while retaining the expressiveness of System F. Only type annotations on parameters of functions that are used polymorphically in their body are required. All other type annotations, including all type abstractions and type applications are inferred. Remarkably, type inference in MLF reduces to a new form of unification that amounts to performing first-order unification in the presence of second-order types. The journal version of this work was published in Information and Computation [2]. The initial design was then simplified and made more expressive by Boris Yakobowski in his Ph.D. dissertation [54], using graphs rather than terms to represent types and also to perform type inference.

This year, Didier Rémy and Boris Yakobowski have further improved the graphical presentation of MLF by following a more algebraic approach. This is described in two journal articles in preparation. Didier Rémy and Boris Yakobowski also revisited the fully explicit version of MLF, which can be used both to show the preservation of types during reduction and as an internal language for MLF. This work was presented at the International Conference on Functional and Logic Programming, held in Sendai, Japan in April 2010 [26]. An extended version of this work has been submitted for journal publication. The elaboration of eMLF into xMLF has been added to the prototype implementation of MLF by Gabriel Scherer during his master internship [37].

Gabriel Scherer (master's intern from ENS Paris) and Didier Rémy have continued the exploration of MLF with higher-order types. To keep the system decidable, type abstraction over higher-order kind variables is made explicit while type abstraction over base kinds left implicit as in the original MLF. This gives MLF the power of System F^ω while keeping implicit the introduction and elimination of polymorphism at second-order types and many type type annotations on function parameters. This work is described in Gabriel Scherer's internship dissertation [37] and has been implemented in the new MLF prototype[1].

6.4.3. A generalization of F-eta with abstraction over retyping functions

Participants: Julien Cretin, Didier Rémy.

Expressive type system often allow non trivial conversions between types, leading to complex, challenging, and sometimes ad hoc type systems. Such examples are the extension of System F with type equalities to model Haskell GADT and type families, or the extension of System F with explicit contracts. A useful technique to simplify the meta-theoretical studies of such systems is to make conversions fully explicit in terms via "coercions".

The essence of coercion functions is perhaps to be found in System F^η, which is the closure of System F by η-reduction, but which can also be seen at the extension of System F with *retyping functions*: retyping functions allow deep type specialization of terms, strengthening the domain of functions or weakening their codomains *a posteriori*.

However, System F^η lacks abstraction over retyping functions, which coercion systems often need. We have studied an extension $F^{\lambda\eta}$ of F^η that precisely allows such abstraction. The main difficulty in the design of F^η is to preserve the erasure semantics of System F, *i.e.* to allow coercions to be dropped before evaluation without changing the meaning of programs. The language $F^{\lambda\eta}$ extends both F^η and xMLF, the internal language of MLF. As a side result, this proves the termination of reduction in xMLF. This work is described in Julien Cretin's Master dissertation [36].

6.4.4. A machine-checked proof of a type-and-capability calculus

Participant: François Pottier.

This year, François Pottier developed a machine-checked proof of an expressive type-and-capability system. Such a system 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. It confirms that earlier publications by Charguéraud and Pottier [1], [7] were indeed correct, offers insights into the design of the type-and-capability system, and provides a firm foundation for further research. François Pottier is presently writing a paper on this topic.

[1] Available electronically at http://gallium.inria.fr/~remy/mlf/proto/.

6.4.5. Fine-grained static control of side effects

Participants: François Pottier, Jonathan Protzenko.

Building on previous work by Arthur Charguéraud and François Pottier, Jonathan Protzenko started designing a new type system for a functional and imperative language. The goal is to strike a reasonable balance between a fine-grained, highly sophisticated type system, that allows one to gain precise control over side-effects, and a language that is practically usable by mere programmers. This is part of a very active research field that aims at designing new languages that offer stronger guarantees and help better understand side effects.

6.5. The Caml language and system

6.5.1. The Objective Caml system

Participants: Xavier Clerc [team SED], Damien Doligez, Alain Frisch [Lexifi], Jacques Garrigue [University of Nagoya], Xavier Leroy, Nicolas Pouillard.

This year, we released two versions of the Objective Caml system. Damien Doligez acted as release manager for both versions. Version 3.11.2, released in January, is a minor release that corrects about 30 problem reports and grants 9 feature wishes. Version 3.12.0, released in August, offers a number of long-awaited new features:

- Polymorphic recursion: programmers can now declare polymorphic type schemes for recursive and mutually-recursive function definitions, enabling these definitions to reference themselves at different types. (J. Garrigue.)
- First-class modules: the ability to encapsulate modules as value of the core language, manipulate them like any other value, then extract them back to module expressions. This design extends an earlier proposal by Claudio Russo implemented in Moscow ML [52]. (A. Frisch.)
- A new type parameterization mechanism whereas a module-level type name, local to a core-language expression, is mapped to a core-language type variable outside of that expression. (A. Frisch.)
- Two new constructs to facilitate the reuse and incremental modification of module signatures: "destructive" substitution of a type component by a type expression (J. Garrigue), and the `module type of` operator to recover an inferred module type (X. Leroy).
- Three convenience features: local opening of modules in a subexpressions (A. Frisch), shorthand notations for records (X. Leroy), and additional warnings over abbreviated record patterns (X. Leroy).

6.5.2. Dynamic contract checking for Caml

Participant: Na Xu.

A contract in a programming language is a formal and checkable interface specification that allows programmers to declare what a function assumes and what a function guarantees. Na Xu used the OCaml 3.11.2 system as an experimental basis for integrating a contract paradigm into the compiler. This prototype supports dynamic contract checking (DCC), that is, contract violation is reported at run-time. It allows OCaml programmers to reason about their program as well as debug their program based on the contract violation messages, which blame precisely the faulty functions.

6.6. Meta-programming

Participants: Nicolas Pouillard, François Pottier.

In an effort to improve meta-programming support (the ability to write programs that manipulate other programs) in programming languages, we have focused first on the issue of binders. Programming with data structures containing binders occurs frequently: from compilers and static analysis tools to theorem provers and code generators, it is necessary to manipulate abstract syntax trees, type expressions, logical formulae, proof terms, etc. All these data structures contain variables and binding constructs.

Nicolas Pouillard, under the supervision of François Pottier, investigated the design of a programming interface for names and binders where the representations of these two types are kept abstract. This interface is sufficiently general to enable a large body of program transformations. Moreover, it is sufficiently abstract to avoid committing on the choice of the representation of names and binders. This interface comes with two implementations, one using De Bruijn indices and the other using a nominal approach. Each of these implementations is equipped with a logical relation, which defines both α-equivalence of values and contextual equivalence of programs. Since well-typed programs inhabit their logical relation, this establishes that names and binders are handled in a well-behaved manner. These results were published at the ICFP 2010 conference [23].

6.7. Probabilistic contracts for component-based design

Participants: Na Xu, Gregor Gössler [EPI POPART], Alain Girault [EPI POPART].

Na Xu, Gregor Gössler and Alain Girault defined a probabilistic contract frameworkfor the construction of component-based embedded systems, based on the theory of Interactive Markov Chains. A contract specifies the assumptions a component makes on its context and the guarantees it provides. Probabilistic transitions allow for uncertainty in the component behavior, for example, to model observed black-box behavior (internal choice) or reliability. An interaction model specifies how components interact.

Ingredients for a component-based design flow include (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 this result was presented at the ATVA 2010 conference [29]. A technical report version is also available [34].

6.8. Formal management of package dependencies

Participants: Roberto Di Cosmo, Ralf Treinen [U. Paris Diderot], Jaap Boender [U. Paris Diderot], Pietro Abate [U. Paris Diderot], Jérôme Vouillon [U. Paris Diderot], Stefano Zacchiroli [U. Paris Diderot].

Roberto Di Cosmo's current main line of research is the study and analysis of large, component based software repositories, in particular GNU/Linux based distributions. These distributions consists of collections of dozens of thousands of *software packages*, together with metadata, installation and configuration tools, and a variety of different production processes, involving quality assurance at several levels. Ensuring quality of software assemblies built using these components is a challenging issue: the simple question of knowing whether a single component can or not be deployed turns out to be NP-complete, and yet industry needs to deploy components all the time.

The current research done in the framework of the Mancoosi FP7 european project, which is coordinated by Roberto Di Cosmo, adresses some of the relevant issues, by elaborating sophisticated deployment algorithms, and conceiving specialised installation and configuration languages targeted at enabling transactional capabilities in the tools used to mainain software assemblies built out of GNU/Linux based distributions. The results of this project are available at http://www.mancoosi.org/ and include two conference publications this year [19], [20].

7. Contracts and Grants with Industry

7.1. The Caml Consortium

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, Jane Street Capital, LexiFi, Microsoft, MLstate, Mylife.com, OCamlCore, 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. Other Grants and Activities

8.1. The IRILL laboratory

Roberto Di Cosmo has been working on the creation of the IRILL (Initiative d'Innovation et Recherche sur le Logiciel Libre), also known as FSRII (Free Software Research and Innovation Institute). The IRILL has the ambition of providing an attractive environment to researchers working on the new, emerging scientific issues coming from Free Software (the work on package dependencies is an archetypical example), to industry players willing to collaborate with researchers on these issues, and to educators working on improving the CS curricula using Free and Open Source Software.

IRILL is an INRIA joint initiative with University Paris Diderot and University Pierre et Marie Curie: it has beed established by an agreement formally signed on November 2010, and its activities have started with the IRILL Days event in October 2010 (see http://www.irill.org).

8.2. National initiatives

The Gallium project participates in the "U3CAT" project of the *Arpège* programme of *Agence Nationale de la Recherche*. This 3-year action (2009-2011) is coordinated by CEA LIST and focuses on program verification tools for critical embedded C codes. Xavier Leroy is involved in this project on issues related to memory models and formal semantics for the C language, at the interface between compilers and verification tools.

Xavier Leroy participates in the "Ascert" project (2009-2011), coordinated by David Pichardie at INRIA Rennes and funded by *Fondation de Recherche pour l'Aéronautique et l'Espace*. The objective of Ascert is the formal verification of static analyzers.

8.3. Regional initiatives

We participate in two projects of the Digiteo RTRA. The first, named "Metal" (2008-2010), is coordinated by François Pottier and involves also Nicolas Pouillard. This project focuses on formal foundations and static type systems for meta-programming. The second, named "Hisseo" (2008-2010), is coordinated by Pascal Cuoq at CEA LIST and involves Xavier Leroy at Gallium. It studies issues related to floating-point arithmetic in static analyzers and verified compilers.

9. Dissemination

9.1. Interactions with the scientific community

9.1.1. Collective responsibilities within INRIA

Damien Doligez chaired the *Commission du Développement Technologique* of INRIA-Paris Rocquencourt.

François Pottier is a member of INRIA's COST (*Conseil d'Orientation Scientifique et Technologique*) and of INRIA Paris-Rocquencourt's postdoctoral selection committee.

François Pottier is a member of the organizing committee for INRIA Paris-Rocquencourt's *Le modèle et l'algorithme*, a seminar series intended for a general scientific audience. François Pottier also organizes the Gallium-Moscova local seminar.

9.1.2. Collective responsibilities outside INRIA

Roberto Di Cosmo is a member of the Scientific Advisory Board and of the Board of Trustees of the IMDEA Software research institute in Madrid.

Xavier Leroy was a member of the AERES evaluation committee for the VERIMAG laboratory (Grenoble, February 2010).

Xavier Leroy was a member of the hiring committee for an assistant professor position at University Joseph Fourier, Grenoble.

Xavier Leroy was a member of the scientific committee for the ANR colloquium on embedded systems, security and safety (Toulouse, December 2010).

9.1.3. Editorial boards

Xavier Leroy is co-editor in chief of the Journal of Functional Programming. He is a member of the editorial boards of the Journal of Automated Reasoning and the Journal of Formalized Reasoning.

François Pottier is an associate editor for the ACM Transactions on Programming Languages and Systems. His term ends in December 2010.

Didier Rémy is a member of the editorial board of the Journal of Functional Programming.

9.1.4. Program committees and steering committees

Roberto Di Cosmo was a member of the program committee for the LoCoCo 2010 (Logics for Component Configuration) workshop.

Damien Doligez was a member of the program committee for the Journées Francophones des Langages Applicatifs (JFLA 2010).

Xavier Leroy was a member of the program committee of the Verified Software: Theories, Tools and Experiments workshop (VSTTE 2010), the Interactive Theorem Proving conference (ITP 2010), and the Compiler Construction conference (CC 2011). He was a member of the external review committee for the Programming Language Design and Implementation conference (PLDI 2010).

François Pottier was a member of the program committees of the European Symposium on Programming (ESOP 2011) and of the workshop on Syntax and Semantics of Low Level Languages (LOLA 2010).

François Pottier is a member of the steering commitee for the Types in Language Design and Implementation (ACM TLDI) workshop.

Didier Rémy was a member of the program committee of the ACM International Conference on Functional Programming (ICFP, Baltimore, USA, September 2010).

9.1.5. Ph.D. and habilitation juries

Roberto Di Cosmo was reviewer (*rapporteur*) of the *habilitation* of Daniel Le Berre (CRIL, Lens, December). He chaired the Ph.D. jury for Arthur Chargéraud (U. Paris Diderot, December) and was a member of the Ph.D. juries of Vincent Silès (École Polytechnique, November) and Ivan Noyer (ENSIIE, September).

Xavier Leroy was reviewer (*rapporteur*) for the Ph.D. of Élie Soubiran (École Polytechnique, September). He chaired the Ph.D. jury for Benoît Robillard (CNAM, November) and was a member of the Ph.D. juries for Johannes Kanig (U. Paris Sud, November) and Gustavo Petri (U. Nice, November).

9.1.6. Learned societies

Xavier Leroy and Didier Rémy are members of IFIP Working Group 2.8 (Functional Programming).

9.2. Interactions with industry

As founder, president and now vice-president of the Free Software thematic group of the Systematic competitiveness cluster (also known as GTLL), Roberto Di Cosmo played a major role in fostering the emergence of 21 collaborative R&D projects, with a budget over 50 MEUR. These projects bring together researchers from most of the universities and research centers in the Paris area, and industries ranging from SMEs to large corporations. See http://www.gt-logiciel-libre.org/projets/ for more informations on the ongoing projects.

9.3. Teaching

9.3.1. Supervision of Ph.D. and internships

Roberto Di Cosmo is Ph.D. advisor for Jaap Boender at the PPS laboratory.

Xavier Leroy is Ph.D. advisor for Tahina Ramananandro. He also supervised the second-year Master's internship of Vivien Maisonneuve (March–July 2010).

François Pottier is Ph.D. advisor for Arthur Charguéraud, Alexandre Pilkiewicz, Nicolas Pouillard, and (since October this year) Jonathan Protzenko. Arthur Charguéraud defended in December.

Didier Rémy is Ph.D advisor for Benoît Montagu and (since December this year) Julien Cretin. He also supervised the first-year Master's internship (April—July 2010) of Gabriel Scherer and the second-year Master's internship of Julien Cretin.

9.3.2. Graduate courses

The Gallium project-team is involved in the *Master Parisien de Recherche en Informatique* (MPRI), a research-oriented graduate curriculum co-organized by University Paris Diderot, École Normale Supérieure Paris, École Normale Supérieure de Cachan, and École Polytechnique. Xavier Leroy participates in the organization of the MPRI, as INRIA representative on its board of directors and as a member of the *commission des études*.

Roberto Di Cosmo taught a course on Linear Logic at the MPRI (12 hours, September-October 2010).

Didier Rémy, Yann Régis-Gianas (PPS), Giuseppe Castagna (PPS) and Xavier Leroy taught a 48-hour course on functional programming languages and type systems at the MPRI. François Pottier actively participated in the course by designing and grading the mandatory programming assignment.

Xavier Leroy taught a 7-hour course at the Oregon Programming Languages Summer School "Logic, Languages, Compilation, and Verification" (University of Oregon, June 2010), attended by about 80 graduate and post-graduate students.

9.3.3. Undergraduate courses

François Pottier is a part-time assistant professor (*professeur chargé de cours*) at École Polytechnique.

Alexandre Pilkiewicz was teaching assistant for the courses "Algorithmics" for first-year undergraduate students and "Virtual Machines" for third-year undergraduate students at University Paris Diderot (60 hours, January–May 2010).

Tahina Ramananandro was teaching assistant for the course "The C programming language" for second-year undergraduate students (48 hours, January-June 2010), and for the course "Functional programming with Objective Caml" for third-year undergraduate students (24 hours, September-December 2010), both at University Paris Diderot.

9.4. Participation in conferences and seminars

9.4.1. Participation in international conferences

- POPL: Principles of Programming Languages (Madrid, Spain, January).
  Xavier Leroy presented [28]. Arthur Charguéraud, François Pottier, Alexandre Pilkiewicz, Nicolas Pouillard, Tahina Ramananandro and Na Xu attended.

- ETAPS: European Joint Conferences on Theory and Practice of Software (Paphos, Cyprus, March).
  Xavier Leroy presented [25] at the Compiler Construction conference.
- IFIP Working Group 2.8 "Functional programming" (Shirahama, Japan, April).
  Didier Rémy participated.
- FLOPS: Functional and Logic Programming Symposium (Sendai, Japan, April).
  Didier Rémy presented [26].
- FLOC: Federated Logic Conferences (Edinburgh, UK, July 2010).
  Roberto Di Cosmo attended. Damien Doligez attended the IJCAR conference where his co-authored paper [18] was presented, and entered Zenon in the CASC-J5 prover competition hosted by the conference.
- ITP: International Conference on Interactive Theorem Proving (Edinburgh, UK, July).
  Arthur Charguéraud presented [17].
- CONCUR: International Conference on Concurrency Theory (Paris, France, August).
  Na Xu attended.
- Dagstuhl seminar: Modelling, Controlling and Reasoning About State (Dagstuhl, Germany, August).
  Arthur Charguéraud presented a summary of [11]. François Pottier attended.
- ATVA: International Symposium on Automated Technology for Verification and Analysis (Singapore, September).
  Na Xu presented [29].
- ICFP: International Conference on Functional Programming (Baltimore, USA, September).
  Arthur Charguéraud presented [16]. Julien Cretin presented [14]. Nicolas Pouillard presented [23]. Benoît Montagu and François Pottier attended.
- WMM: Workshop on Mechanizing Metatheory (Baltimore, USA, September).
  Benoît Montagu presented [31].

9.4.2. Participation in national conferences

- ANR STIC colloquium (Paris, France, January).
  Xavier Leroy was invited to present the CompCert project in a plenary session.
- Journées du GDR Génie de la Programmation et du Logiciel (Pau, France, March).
  Benoît Montagu presented [30].
- INRIA-industry meeting on Aerospace, part 1 (Bordeaux, France, January).
  Roberto Di Cosmo gave a talk on "Free/Open Source Software: scientific and technological challenges for complex systems engineering".
- INRIA-industry meeting on Aerospace, part 2 (Toulouse, France, May).
  Roberto Di Cosmo gave a talk on "Free/Open Source Software: scientific and technological challenges for complex systems engineering". Xavier Leroy gave a tutorial on the formal verification of software.
- Open World Forum (Paris, France, October).
  Roberto Di Cosmo gave a talk on "IRILL and Open Innovation".
- Free Open Source Academia Conference (Grenoble, France, November).
  Roberto Di Cosmo gave a talk on "Education with and to FOSS".

9.4.3. Invitations and participation in seminars

Julien Cretin gave two talks on [14] at the PL Club seminar of the University of Pennsylvania (Philadelphia, USA, September) and at the Programming Languages seminar of Cornell University (Ithaca, USA, September).

Xavier Leroy visited Andrew Appel's team at Princeton University in June and gave a seminar talk on validating register allocation. He was also invited to talk on compiler verification at the seminar of the AdaCore company (Paris, August).

François Pottier gave a talk on types for complexity-checking at the Plume seminar of ENS Lyon, in May.

Tahina Ramananandro visited the Parasol laboratory of Texas A&M University (College Station, Texas, USA, February 11th–20th) and gave a seminar talk on a machine-checked formalization of C++ object layout.

Na Xu was invited to present *Static Contract Checking for Haskell* to Department of Informatics, University of Sussex, UK, and at Microsoft Research Cambridge summer school, Cambridge, UK (July 2010). Na Xu was invited to present *Probabilistic Contracts for Component-based Design* to School of Computing, National University of Singapore, Singapore (Sep 2010).

9.4.4. Participation in summer schools

Alexandre Pilkiewicz attended the Oregon Programming Languages Summer School "Logic, Languages, Compilation, and Verification" (Eugene, OR, USA, June).

Nicolas Pouillard attended the Spring School on Generic and Indexed Programming (Wadham College, Oxford, UK, March).

Na Xu attended the CEA-EDF-INRIA summer school on "Modelling and verifying algorithms in Coq" (Paris, June).

9.5. Other dissemination activities

Arthur Charguéraud is co-trainer of the French team for the International Olympiads in Informatics (IOI). He participates in the France-IOI association, which aims at promoting programming and algorithmics among high-school students.

Roberto Di Cosmo gave a tutorial on "The complexity of free software" at the *Fête de la Science* in October.

Damien Doligez was the recipient of a Cygwin Gold Star Award for taking over maintenance of the Cygwin OCaml package.

Xavier Leroy published a popular science article on compiler verification in the monthly *La Recherche* [35]. This text was republished on the *Interstices* Web site.

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] 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.

[3] 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.

[4] 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.

[5] 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.

[6] F. POTTIER. *Static Name Control for FreshML*, in "Twenty-Second Annual IEEE Symposium on Logic In Computer Science (LICS'07)", IEEE Computer Society Press, July 2007, p. 356–365, http://dx.doi.org/10.1109/LICS.2007.44.

[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] 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://doi.acm.org/10.1145/1180475.1180476.

[10] J.-B. TRISTAN, X. LEROY. *Formal verification of translation validators: A case study on instruction scheduling optimizations*, in "Proceedings of the 35th ACM Symposium on Principles of Programming Languages (POPL'08)", ACM Press, January 2008, p. 17–27, http://doi.acm.org/10.1145/1328897.1328444.

Publications of the year

Doctoral Dissertations and Habilitation Theses

[11] A. CHARGUÉRAUD. *Characteristic Formulae for Mechanized Program Verification*, Université Paris Diderot (Paris 7), December 2010, http://arthur.chargueraud.org/research/2010/thesis/.

[12] B. MONTAGU. *Programmer avec des modules de première classe dans un langage noyau pourvu de sous-typage, sortes singletons et types existentiels ouverts*, École Polytechnique, December 2010, English title: Programming with first-class modules in a core language with subtyping, singleton kinds and open existential types.

Articles in International Peer-Reviewed Journal

[13] M. DEZANI-CIANCAGLINI, R. DI COSMO, E. GIOVANNETTI, M. TATSUTA. *On isomorphisms of intersection types*, in "ACM Transactions on Computational Logic", 2010, vol. 11, n^{o} 4, Article No. 25, http://doi.acm.org/10.1145/1805950.1805955.

International Peer-Reviewed Conference/Proceedings

[14] D. M. J. BARBOSA, J. CRETIN, N. FOSTER, M. GREENBERG, B. C. PIERCE. *Matching Lenses: Alignment and View Update*, in "Proceedings of the 15th International Conference on Functional Programming (ICFP 2010)", ACM Press, 2010, p. 193–204, http://doi.acm.org/10.1145/1863543.1863572.

[15] R. BEDIN FRANÇA, D. FAVRE-FELIX, X. LEROY, M. PANTEL, J. SOUYRIS. *Towards Optimizing Certified Compilation in Flight Control Software*, in "Workshop on Predictability and Performance in Embedded Systems (PPES 2011)", OpenAccess Series in Informatics, Dagstuhl Publishing, 2011, To appear.

[16] A. CHARGUÉRAUD. *Program Verification Through Characteristic Formulae*, in "Proceedings of the 15th International Conference on Functional Programming (ICFP 2010)", ACM Press, 2010, p. 321–332, http://doi.acm.org/10.1145/1863543.1863590.

[17] A. CHARGUÉRAUD. *The Optimal Fixed Point Combinator*, in "Interactive Theorem Proving (ITP 2010)", Lecture Notes in Computer Science, Springer, 2010, vol. 6172, p. 195-210, http://dx.doi.org/10.1007/978-3-642-14052-5_15.

[18] 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.

[19] R. DI COSMO, J. BOENDER. *Using strong conflicts to detect quality issues in component-based complex systems*, in "ISEC '10: Proceedings of the 3rd India software engineering conference", ACM Press, 2010, p. 163–172, http://doi.acm.org/10.1145/1730874.1730905.

[20] R. DI COSMO, S. ZACCHIROLI. *Feature Diagrams as Package Dependencies*, in "Software Product Lines: Going Beyond - 14th International Conference, SPLC 2010", Lecture Notes in Computer Science, Springer, 2010, vol. 6287, p. 476-480, http://dx.doi.org/10.1007/978-3-642-15579-6_40.

[21] A. PILKIEWICZ, F. POTTIER. *The essence of monotonic state*, in "6th Workshop on Types in Language Design and Implementation (TLDI 2011)", ACM Press, 2011, To appear, http://gallium.inria.fr/~fpottier/publis/pilkiewicz-pottier-monotonicity.pdf.

[22] F. POTTIER. *A typed store-passing translation for general references*, in "Proceedings of the 38th ACM Symposium on Principles of Programming Languages (POPL'11)", ACM Press, 2011, To appear, http://gallium.inria.fr/~fpottier/publis/fpottier-fork.pdf.

[23] 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.

[24] T. RAMANANANDRO, G. DOS REIS, X. LEROY. *Formal verification of object layout for C++ multiple inheritance*, in "Proceedings of the 38th ACM Symposium on Principles of Programming Languages (POPL'11)", ACM Press, 2011, To appear, http://gallium.inria.fr/~xleroy/publi/cpp-object-layout.pdf.

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

[26] D. RÉMY, B. YAKOBOWSKI. *A Church-Style Intermediate Language for MLF*, in "Functional and Logic Programming, 10th International Symposium, FLOPS 2010", Lecture Notes in Computer Science, Springer, 2010, vol. 6009, p. 24–39, http://dx.doi.org/10.1007/978-3-642-12251-4_4.

[27] J. SCHWINGHAMMER, H. YANG, L. BIRKEDAL, F. POTTIER, B. REUS. *A Semantic Foundation for Hidden State*, in "Foundations of Software Science and Computation Structures (FoSSaCS 2010)", Lecture Notes in Computer Science, Springer, 2010, vol. 6014, p. 2–17, http://dx.doi.org/10.1007/978-3-642-12032-9_2.

[28] 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.

[29] D. N. XU, G. GÖSSLER, A. GIRAULT. *Probabilistic Contracts for Component-based Design*, in "Automated Technology for Verification and Analysis (ATVA 2010)", Lecture Notes in Computer Science, Springer, 2010, vol. 6252, p. 325–340, http://dx.doi.org/10.1007/978-3-642-15643-4_24.

National Peer-Reviewed Conference/Proceedings

[30] B. MONTAGU, D. RÉMY. *Types abstraits et types existentiels ouverts*, in "Actes des deuxièmes journées nationales du Groupement De Recherche CNRS du Génie de la Programmation et du Logiciel", Université de Pau, É. CARIOU, L. DUCHIEN, Y. LEDRU (editors), March 2010, p. 147–148, http://gdr-gpl.imag.fr/.

Workshops without Proceedings

[31] B. MONTAGU. *Experience report: Mechanizing Core F-zip using the locally nameless approach (extended abstract)*, in "5th ACM SIGPLAN Workshop on Mechanizing Metatheory", 2010, http://www.cis.upenn.edu/~bcpierce/wmm/wmm10/montagu.pdf.

Scientific Books (or Scientific Book chapters)

[32] X. LEROY. *Mechanized semantics*, in "Logics and languages for reliability and security", NATO Science for Peace and Security Series D: Information and Communication Security, IOS Press, 2010, vol. 25, p. 195–224, http://dx.doi.org/10.3233/978-1-60750-100-8-195.

Research Reports

[33] X. LEROY, D. DOLIGEZ, J. GARRIGUE, D. RÉMY, J. VOUILLON. *The Objective Caml system, documentation and user's manual – release 3.12*, INRIA, August 2010, http://caml.inria.fr/pub/docs/manual-ocaml/.

[34] D. N. XU, G. GÖSSLER, A. GIRAULT. *Probabilistic Contracts for Component-based Design*, INRIA, 07 2010, n^{o} RR-7328, http://hal.inria.fr/inria-00507785/en/.

Scientific Popularization

[35] X. LEROY. *Comment faire confiance à un compilateur?*, in "La Recherche", April 2010, vol. 440, Les cahiers de l'INRIA, http://www.lescahiersinria.com/sites/default/files/inria-n440-avril10.pdf.

Other Publications

[36] J. CRETIN. *A generalization of F-eta with abstraction over retyping functions*, École Polytechnique, December 2010, http://gallium.inria.fr/~jcretin/papers/master_thesis.pdf.

[37] G. SCHERER. *Extending MLF with Higher-Order Types*, École Normale Supérieure, Paris, August 2010, http://gallium.inria.fr/~remy/mlf/scherer@master2010:mlfomega.pdf.

References in notes

[38] 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.

[39] 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.

[40] S. GULWANI, G. NECULA. *A polynomial-time algorithm for global value numbering*, in "Science of Computer Programming", 2007, vol. 64, n° 1, p. 97–114.

[41] 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.

[42] L. LAMPORT. *How to write a proof*, in "American Mathematical Monthly", August 1993, vol. 102, n° 7, p. 600–608, http://research.microsoft.com/users/lamport/pubs/lamport-how-to-write.pdf.

[43] S. LERNER, D. GROVE, C. CHAMBERS. *Composing dataflow analyses and transformations*, in "Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '02", ACM Press, 2002, p. 270–282.

[44] 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.

[45] Z. MANNA, A. SHAMIR. *The optimal fixedpoint of recursive programs*, in "Proceedings of the 7th Annual ACM Symposium on Theory of Computing, STOC'75", ACM Press, 1975, p. 194–206.

[46] J. MATTHEWS. *Recursive Function Definition over Coinductive Types*, in "Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99", Lecture Notes in Computer Science, Springer, 1999, vol. 1690, p. 73–90.

[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° 2, p. 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° 1, p. 117–158, http://gallium.inria.fr/~fpottier/publis/fpottier-simonet-toplas.ps.gz.

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

[51] N. RAMSEY, J. DIAS, S. PEYTON JONES. *Hoopl: Dataflow optimization made simple*, in "ACM SIGPLAN Haskell Symposium", ACM Press, 2010.

[52] C. V. RUSSO. *First-Class Structures for Standard ML*, in "Nordic Journal of Computing", 2000, vol. 7, n° 4, p. 348-374.

[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] B. YAKOBOWSKI. *Graphical types and constraints: second-order polymorphism and inference*, University Paris Diderot (Paris 7), December 2008, http://tel.archives-ouvertes.fr/tel-00357708/.