INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

# *Project-Team Gallium*

# *Programming languages, types, compilation and proofs*

*Paris - Rocquencourt*

THEME SYM

# Table of contents

# 1. Team

**Research Scientist**

Xavier Leroy [ Team leader, DR INRIA ]
Damien Doligez [ CR INRIA ]
François Pottier [ DR INRIA, HdR ]
Didier Rémy [ Deputy team leader, DR INRIA, HdR ]

**External Collaborator**

Sandrine Blazy [ Assistant professor, ENSIIE, HdR ]
Michel Mauny [ Professor, ENSTA ]

**Technical Staff**

Nicolas Pouillard [ Ingénieur expert, until September 2008 ]

**PhD Student**

Arthur Charguéraud [ AMN grant, U. Paris 7 ]
Zaynah Dargaye [ Région Île de France grant, U. Paris 7 ]
Benoît Montagu [ AMX grant, Polytechnique ]
Alexandre Pilkiewicz [ AMX grant, Polytechnique, since December 2008 ]
Nicolas Pouillard [ Digiteo grant, INRIA, since October 2008 ]
Tahina Ramananandro [ AMN grant, U. Paris 7 ]
Jean-Baptiste Tristan [ ANR grant, INRIA ]
Boris Yakobowski [ AMN grant, U. Paris 7 ]

**Post-Doctoral Fellow**

Keiko Nakata [ CNAM, half-time with CNAM/ENSIIE, until September 2008 ]

**Administrative Assistant**

Stéphanie Aubin [ TR INRIA, since March 2008 ]
Nelly Maloisel [ TR INRIA, until February 2008 ]

**Other**

Alexandre Pilkiewicz [ Master's intern, July–November 2008 ]

# 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 embodies many of our earlier 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 [30]. 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 [43], [41], whose design is driven by the idea of regular expression types that enforce DTDs at compile-time. For this reason, research on type systems – their design, their proof of semantic correctness (type safety), the development and proof of associated type checking and inference algorithms – plays a large and central role in the field of programming language research, as evidenced by the huge number of type systems papers in conferences such as Principles of Programming Languages.

### *3.2.2. Polymorphism in type systems.*

There exists a fundamental tension in the field of type systems that drives much of the research in this area. On the one hand, the desire to catch as many programming errors as possible leads to type systems that reject more programs, by enforcing fine distinctions between related data structures (say, sorted arrays and general arrays). The downside is that code reuse becomes harder: conceptually identical operations must be implemented several times (say, copying a general array and a sorted array). On the other hand, the desire to support code reuse and to increase expressiveness leads to type systems that accept more programs, by assigning a common type to broadly similar objects (for instance, the `Object` type of all class instances in Java). The downside is a loss of precision in static typing, requiring more dynamic type checks (downcasts in Java) and catching fewer bugs at compile-time.

*Polymorphic* type systems offer a way out of this dilemma by combining precise, descriptive types (to catch more errors statically) with the ability to abstract over their differences in pieces of reusable, generic code that is concerned only with their commonalities. The paradigmatic example is parametric polymorphism, which is at the heart of all typed functional programming languages. Many forms of polymorphic typing have been studied since then. Taking examples from our group, the work of Rémy, Vouillon and Garrigue on row polymorphism [50], integrated in 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 little 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 that cries for application to more exotic programming languages. 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 two less investigated topics: compiler certification and efficient compilation of "exotic" languages.

### *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.

### *3.3.2. Efficient compilation of high-level languages.*

High-level and domain-specific programming languages raise fascinating compilation challenges: on the one hand, compared with Fortran and C, the wider semantic gap between these languages and machine code makes compilation more challenging; on the other hand, the stronger semantic guarantees and better controlled execution model offered by these languages facilitate static analysis and enable very aggressive optimizations. A paradigmatic example is the compilation of the Esterel reactive language: the very rich control structures of Esterel can be resolved entirely at compile-time, resulting in software automata or hardware circuits of the highest efficiency.

We have been working for many years on the efficient compilation of functional languages. The native-code compiler of the Objective Caml system embodies our results in this area. By adapting relatively basic compilation techniques to the specifics of functional languages, we achieved up to 10-fold performance improvements compared with functional compilers of the 80s. We are currently considering more advanced optimization techniques that should help bridge the last factor of 2 that separates Caml performance from that of C and C++. We are also interested in applying our knowledge in compilation to domain-specific languages that have high efficiency requirements, such as modeling languages used for simulations.

## 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 [49].

### *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. Software safety and security

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. Static typing detects programming errors early and prevents a number of popular security attacks: buffer overflows, executing network data as if it were code, etc. On the safety side, judicious uses of type abstraction and other encapsulation mechanisms allow static type checking to enforce program invariants. On the security side, the methods used in designing type systems and establishing their soundness are also applicable to the specification and automatic verification of some security policies such as non-interference for data confidentiality.

## 4.2. 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. Therefore, Caml is a suitable 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 analysis tools, etc.

## 4.3. Fast 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.

## 4.4. 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 French *classes préparatoires scientifiques*. Objective Caml is also widely used for teaching advanced programming in engineering schools, colleges and universities in France, the US, and Japan.

# 5. Software

## 5.1. Objective Caml

**Participants:** Xavier Leroy [correspondant], Damien Doligez, Jacques Garrigue [Kyoto University], Maxence Guesdon [team SED], Luc Maranget [project Moscova], Michel Mauny, Nicolas Pouillard, Pierre Weis [team AT-Roc].

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 9 processor architectures (IA32, PowerPC, AMD64, Alpha, Sparc, Mips, IA64, HPPA, StrongArm), 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, and the Camlp4 source pre-processor.

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

## 5.2. CompCert C

**Participant:** Xavier Leroy.

The CompCert C verified compiler is a compiler for a large subset of the C programming language that generates code for the PowerPC processor. 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, all structured control (`if`, loops, `break`, `continue`, structured `switch`) but not `goto`, 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.

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. Type systems

### *6.1.1. Partial type inference with first-class polymorphism*

**Participants:** Didier Rémy, Boris Yakobowski, Didier Le Botlan [INSA Toulouse].

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 initial study of MLF was the topic of Didier Le Botlan's Ph.D. dissertation [46]. There is an interesting restriction of MLF that retains most of its expressiveness while being simpler and more intuitive for which types can be interpreted as sets of System-F types and type-instantiation becomes set inclusion on the semantics. This justifies a posteriori the type-instance relation of MLF that was previously defined only by syntactic means. This works has been accepted for publication in the journal Information and Computation [45].

Boris Yakobowski completed his Ph.D., defended in December, under the supervision of Didier Rémy. His dissertation is a new simplification of MLF using graphs rather than terms to represent types. Graph types are the superposition of a DAG representation of first-order terms and a binding tree that describes *where* and *how* variables are bound. This representation is much more canonical than syntactic types and eliminates most of the notational redundancies of the syntactic presentation.

This exposed a linear-time unification algorithm on graph types, which can be decomposed into standard first-order unification on the underlying dags and a simple computation on the underlying binding trees. These results were presented at the workshop on Types in Language Design and Implementation (TLDI 07) last year [51].

Graphic types can also be used to represent type inference constraints internally. This is a stepping stone for efficient type inference for MLF. This also allows for a direct specification of type inference via generation of typing constraints. This result was presented at the International Conference on Functional Programming (ICFP 08) [25].

Although the implicitly typed version of MLF preserves well-typedness during reduction, explicit type annotations cannot be maintained during reduction in the type-inference version of MLF because types are still too implicit. Didier Rémy and Boris Yakobowski have designed a fully explicit version of MLF with a type-preserving reduction semantics. This work is described in Boris Yakobowski's Ph.D. dissertation [11] and is to be submitted for publication [38].

### 6.1.2. *Type-based analysis of hidden state in imperative programs*

**Participant:** François Pottier.

François Pottier studied the notion of *hidden state*, and proposed a type-theoretic explanation of it.

In earlier work, Charguéraud and Pottier designed a type system, featuring regions, capabilities, and notions of linearity, which allows fine-grained reasoning about aliasing and ownership in imperative programs with dynamic memory allocation. This work was presented this year at the International Conference on Functional Programming (ICFP 2008) [17].

In the type system of Charguéraud and Pottier, all side effects are explicit. A function that wishes to write to a memory region must be passed a *capability* over that region as an argument. A function has access to no capabilities other than those provided by its caller, so it is impossible for a function to have *persistent, private state*.

The lack of private state can be considered a problem. It leads to a loss of modularity. Furthermore, it means that there is no satisfactory encoding of a traditional type system, such as ML's, into Charguéraud and Pottier's system.

Pottier has investigated this problem and extended Charguéraud and Pottier's system with a single new typing rule, known as the *anti-frame rule*, which permits hidden state. This not only makes the system more expressive, but also sheds light on the essence of hidden state and of ML references. This work has strong connections with the higher-order frame rules found in separation logic. It was published and presented at the Logic in Computer Science symposium (LICS 2008) [22].

### 6.1.3. *Type-based complexity analysis*

**Participants:** Alexandre Pilkiewicz, François Pottier.

Alexandre Pilkiewicz and François Pottier have investigated how type systems can be used to state and check complexity properties.

The goal is fairly simple: a function type should not only indicate what kind of argument a function expects, and what kind of result it produces, but also how much time it consumes. For instance, the types of the "push" and "pop" operations in a FIFO queue could indicate that these are constant-time operations; the type of a sorting algorithm could indicate that it requires time proportional to $n \log n$, where $n$ is the length of the sequence that is being sorted. The type checker should be able to verify these claims.

The technical means that are being used are also quite simple. Pilkiewicz and Pottier begin with an existing linear type system. To this system, they add a linear type of *credits*, representing units of computation time. Furthermore, they change the way in which function applications are typechecked, so that every function call requires (and consumes) a credit. They show that this approach is sound, and allows stating both worst-case and amortized complexity estimates.

## 6.2. Module systems

### *6.2.1. 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 a new form of paths in types that do not depend on values 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 IFIP 2.8 working group and was accepted for presentation at the conference on Principles of Programming Languages (POPL 2009) to be held in Savannah, USA in January 2009 [21].

Several directions for future work are being considered. One is the study of paths at the type level. Another is to exploit the first-class nature of our approach to increase expressiveness and conciseness of the module sub-language. Finally, we plan to build on the simplicity of the formalism to tackle mixin modules.

### *6.2.2. Lazy semantics for recursive modules and mixin modules*

**Participant:** Keiko Nakata.

Recursively-defined modules are an extension of the ML module system, found in Objective Caml in particular, that significantly increases the expressive power of this module system. Mixin modules further extend the ML module system with open recursion and the ability to combine and override definitions a posteriori, in the style of class-based object-oriented programming.

Recusive modules and mixin modules raise several challenges, one of which is the definition of a suitable evaluation strategy for computations (initialization code) contained in modules. Keiko Nakata developed evaluation strategies for recursive and mixin modules based on lazy, on-demand evaluation of modules and sub-modules. While expressive and flexible, lazy evaluation makes the order of computations hard to predict by programmers when combined with side effects. Keiko Nakata investigated the use of constraint systems to enable programmers to control evaluation order more precisely. Preliminary results were submitted for publication [35], [36].

## 6.3. Formal verification of compilers

### *6.3.1. The Compcert verified compiler for the C language*

**Participants:** Xavier Leroy, Sandrine Blazy [CEDRIC-ENSIIE].

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 architecture. This compiler comprises a back-end part, translating the Cminor intermediate language to PowerPC assembly and reusable for source languages other than C [4], and a front-end translating the Clight subset of C to Cminor [42]. 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 40000-line, machine-checked Coq proof of semantic preservation, proving that the semantics of the generated PowerPC code matches that of the source Clight program.

This year, we investigated transition (small-step) semantics for the high-level languages of the Compcert compilation chain (Clight and Cminor). Following an earlier proposal by Andrew Appel and Sandrine Blazy [40], these transition semantics represent control points and call stacks as continuation terms. Compared with the natural (big-step) semantics used previously for Clight and Cminor, the transition semantics are more compact (there is no duplication of evaluation rules between the terminating and diverging cases), and can easily support the `goto` statement. Xavier Leroy revised the translation from Cminor to RTL (the next intermediate language of the back-end) and its proof of correctness to use the new reduction semantics for Cminor, including `goto`. Along the same lines, he attempted to adapt the correctness proof of the translation from Clight to Cminor to transition semantics, but ran into serious difficulties: some of the transformations performed over loops are very difficult to prove correct in terms of continuations, while their proofs are easy using natural semantics.

In parallel, Xavier Leroy experimented with re-targeting the back-end of the Compcert compiler to emit code for the ARM processor instead of the PowerPC. The "stacking" pass (materialization of activation records) was reworked to abstract away processor-dependent details. The enforcement of function calling conventions during register allocation and spilling was revised. In the end, less than 15% of the Coq development was modified to target the ARM processor. However, the machine-dependent parts of the Compcert back-end are not yet cleanly separated from the machine-independent parts, making it difficult to support simultaneously several target processors.

Two articles on parts of the Compcert development potentially reusable in other projects were published in *Journal of Automated Reasoning*. The first of these papers [13] describes the Coq formalization of the C-like memory model used throughout the Compcert development, and its uses to reason about program transformations that modify the memory behavior of the program. The second paper [14] focuses on the compilation algorithm for parallel assignments used in Compcert and on its surprisingly delicate Coq proof of correctness.

Xavier Leroy wrote and submitted a comprehensive (80 pages) article [34] describing in details the Compcert back-end compiler, its intermediate languages, and its proof of semantic preservation. Sandrine Blazy and Xavier Leroy submitted another article [31] describing the Clight source language of the Compcert compiler and the mechanization of its semantics.

Finally, two versions of the Compcert development were publically released: version 1.2 in April and 1.3 in August. The releases include everything needed to build the executable Compcert C compiler and re-check its correctness. They are licensed for non-commercial uses, including evaluation, research and teaching purposes.

### *6.3.2. Formal verification of register allocation*

**Participants:** Sandrine Blazy [CEDRIC-ENSIIE], Benoît Robillard [CEDRIC-ENSIIE].

As part of his Ph.D. thesis supervised by Sandrine Blazy, Benoît Robillard investigates the development and formal verification of graph-theoretic algorithms useful for register allocation, as well as the development of new algorithms based on integer linear programming to provide exact solutions for register allocation. The latter algorithms proceed by reductions and simplifications of interference graphs.

Last year, Benoît Robillard developed a greedy algorithm that is adapted to the CompCert register allocation, and proved in Coq the optimality of this algorithm [28]. This year, Benoît Robillard improved this Coq development by using the FMaps library in order to model graphs. He started to develop a prototype library in Coq for non-oriented graphs.

Sandrine Blazy and Benoît Robillard studied optimal coalescing in the context of extreme live-range splitting (i.e. live-range splitting after each instruction). They designed two optimizations: a graph reduction followed by a graph decomposition based on clique separators.

The two optimizations have been tested on a reference benchmark: Appel's optimal coalescing challenge. For this benchmark, the cutting-plane algorithm for optimal coalescing (the only optimal algorithm for coalescing) runs 300 times faster when combined with the optimizations. Moreover, thanks to these optimiztions, all the solutions of the optimal coalescing challenge are found, including the 3 instances that were until now unsolved. These results were presented in a paper submitted to the Compiler Construction conference [32].

### *6.3.3. Verified translation validation*

**Participants:** Jean-Baptiste Tristan, Xavier Leroy.

Verified translation validation provides an alternative to proving semantics preservation for the transformations involved in a certified compiler. Instead of proving that a given transformation is correct, we validate it a posteriori, i.e. we verify that the transformed program behaves like the original. The validation algorithm is described using the Coq proof assistant and proved correct, i.e. that it only accepts transformed programs semantically equivalent to the original. In contrast, the program transformation itself can be implemented in any language and does not need to be proved correct.

Jean-Baptiste Tristan, under the supervision of Xavier Leroy, investigated this approach in the case of Lazy Code Motion. Lazy Code Motion is a program optimization that reduces execution time by performing global common subexpression elimination, partial redundancy elimination and loop invariant code motion. This year, Jean-Baptiste Tristan successfully built and proved correct a validator for Lazy Code Motion. This result is presented in a paper that has been submitted [39]. This experiments is remarkable by the simplicity of its solution. It provides strong evidence that complex global optimizations such as Lazy Code Motion are within reach of formally verified compilers such as CompCert.

To complete our study on the use of verified translation validation in the context of verified compilers, we must address loop optimizations. We have chosen to study a pinnacle of compiler optimization: software pipelining. Software pipelining is a loop transformation that overlaps different iterations of a loop to reduce execution time. This optimization is very challenging to validate and certify because it heavily modifies the structure of the program while scheduling the code and renaming registers. To the best of our knowledge, this constitutes the first attempt to certify a loop optimization of this complexity.

### *6.3.4. Verification of a compiler front-end for Mini-ML*

**Participants:** Zaynah Dargaye, Xavier Leroy.

As part of her Ph.D. thesis and under Xavier Leroy's supervision, Zaynah Dargaye investigates the development and formal verification of a translator from a small call-by-value functional language (Mini-ML) to Cminor. Mini-ML features functions as first-class values, constructed data types and shallow pattern-matching, making it an adequate target for Coq's program extraction facility.

This year, Zaynah Dargaye finished the development and Coq proof of correctness of this translator. It performs function uncurrying (the transformation of curried functions into $n$-ary functions), closure conversion, explicit registration of memory roots with the garbage collector (using an original monadic intermediate form), and generation of Cminor code. An executable translator was generated by automatic extraction from the Coq specifications. In conjuction with the existing CompCert back-end compiler and a simple memory allocator and garbage collector written in Cminor, this executable translator is able to generate PowerPC executables from Mini-ML programs.

Zaynah Dargaye is writing up these results in her Ph.D. dissertation, tentatively titled *Vérification formelle d'un compilateur optimisant pour un langage fonctionnel*.

### *6.3.5. Formal verification of runtime systems*

**Participants:** Tahina Ramananandro, Xavier Leroy.

High-level languages such as ML or Java are equipped with runtime systems, providing services ranging from memory management to full virtual machines. Such runtime systems are linked with compiled programs. Thus, to develop a verified compiler for such languages, it is critical to prove that their runtime systems do not modify or break the semantics of source programs. Garbage collectors, automatically managing memory for the actual representation of objects in such languages, are critical parts of these runtime systems. As they heavily use pointer arithmetics, their proofs must be treated with specific methods.

Last year, Tahina Ramananandro attempted to formally prove the correctness of a mark-and-sweep garbage collector written in the Cminor intermediate language, using the Coq proof assistant. These proofs required manual reasoning between real Cminor code and a proof-level representation of the algorithm directly expressed in Gallina (the specification language of the Coq proof assistant), as well as fine reasoning over pointer and machine integer arithmetics.

This year, under Xavier Leroy's supervision, Tahina Ramananandro attacked the issue of developing automatization tools for such proofs, as well as a domain-specific language for writing garbage collectors, along with tools to reason about both its operational and axiomatic semantics and a verified compiler from this language to Cminor. Tahina Ramananandro then revisited the formal proof of the mark-and-sweep garbage collector implementation using this specific language.

A potential application contexts for this verified garbage collector, in addition to the MiniML front-end of Zaynah Dargaye, is a verified static compiler from Java bytecode to RTL (a CFG-style intermediate language of the CompCert back-end). Tahina Ramananandro developed (in OCaml) a prototype of this static compiler.

## 6.4. Program specification and proof

### *6.4.1. Interactive verification of functional programs*

**Participant:** Arthur Charguéraud.

In recent work [17], Arthur Charguéraud and François Pottier designed a type system which can be used to direct a fine-grained translation of well-typed imperative ML programs into a purely functional language. Thanks to this translation, one is able to verify an imperative program simply by verifying its functional counterpart. In order to complete the setting, a technique for formally reasoning on functional programs is required.

Arthur Charguéraud proposed a framework for modular verification of purely functional OCaml code using the Coq proof assistant. It relies on a *deep embedding*, that is, a description of the syntax and the semantics of a programming language in the logic of a proof assistant. Technically, programs are specified through lemmas describing their intended big-step behaviour, and they are verified through proofs of such lemmas.

In theory, this approach can be used to verify any true property of any given program. It imposes no restriction on the code, apart from its purity, and benefits from the strong expressiveness of Coq. Its pratical abilities have been established through a complete specification and verification of OCaml's list library, as well as a proof of total correctness for a bytecode compiler and interpreter for Mini-ML. This work is presented in a paper [33] that has been submitted for publication.

### *6.4.2. The Zenon automatic theorem prover*

**Participant:** Damien Doligez.

Damien Doligez continued the development of Zenon, a tableau-based automatic theorem prover for first-order logic with equality with extensions. The major developments this year include specialized rules for set theory (used for TLA+), as well as induction rules for recursive functions defined on inductive data types (used for Focal). Zenon was also extended with a back-end proof generator that produces proof scripts for the Isabelle system. Zenon version 0.6.3 was released in December.

Zenon is used within the Focal project [26], a joint effort with LIP6 and CNAM-CEDRIC. Focal is a programming language and a set of tools for software-proof codesign. Focal proofs are done in a hierarchical language invented by Leslie Lamport [44]. Each leaf of the proof tree is a lemma that must be proved before the proof is detailed enough for verification by Coq. The Focal compiler translates this proof tree into an incomplete proof script, which is then completed by Zenon.

Other uses of Zenon are being considered in the context of the *Action de Recherche Coopérative* "Quotient" (http://quotient.loria.fr/).

### *6.4.3. Tools for TLA+*

**Participants:** Damien Doligez, Leslie Lamport [Microsoft Research], Stephan Merz [project Mosel], Kaustuv Chaudhuri [Microsoft Research-INRIA Joint Centre], Simon Zambrowski [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 [44], and to build tools for writing TLA+ specifications and mechanically checking the corresponding formal proofs.

We have finished the design of the proof language and we have a first prototype of the front-end processor (parser and type-checker). Kaustuv Chaudhuri was hired in November 2007 for a 2-year post-doctoral position; he is the main architect and implementer of the "proof manager", a development environment for developing TLA+ specification with their proofs. The "proof manager" uses Isabelle and Zenon as back-ends. These developments are described in an article published in the KEAPPA workshop [18].

Simon Zambrowski has started working on a TLA+ plug-in for the Eclipse integrated development environment.

## 6.5. The Objective Caml system and extensions

### *6.5.1. The Objective Caml system*

**Participants:** Damien Doligez, Xavier Leroy, Michel Mauny, Nicolas Pouillard, Pascal Cuoq [CEA], Alain Frisch [Lexifi], Jacques Garrigue [University of Nagoya], Maxence Guesdon [team SED], Luc Maranget [project Moscova], Pierre Weis [project Estime].

This year, we released versions 3.10.1, 3.10.2 and 3.11.0 of the Objective Caml system. Damien Doligez acted as release manager for these three versions.

Versions 3.10.1, released in January 2008, and 3.10.2, released in March 2008, are minor releases that correct about 85 problem reports. In addition, Damien Doligez and Pascal Cuoq enhanced the performance of weak pointers in the run-time system. The problems with weak pointers, Doligez and Cuoq's solution, and applications to weak hash tables were presented in a paper published at the ACM Workshop on ML [19].

Version 3.11.0, released in December 2008, is a major release that brings the following improvements:

- Support for dynamic loading of natively-compiled Caml code (Alain Frisch).
- Addition of lazy patterns of the form `lazy` $p$, matching suspended computations whose values, after forcing, match the pattern $p$ (Michel Mauny).
- Extensions of private type declarations and of type coercions (Jacques Garrigue, Pierre Weis).
- Improvements in memory management to reduce fragmentation and better support 64-bit platforms (Damien Doligez, Xavier Leroy).
- Generation of 64-bit code for the Mac OS X / Intel platform (Xavier Leroy).

### *6.5.2. Software migration from reFLect to OCaml*

**Participants:** Michel Mauny, Nicolas Pouillard.

Since september 2005 and with support from Intel, Michel Mauny, Nicolas Pouillard and Virgile Prevosto developed an implementation of the reFLect functional language designed and used at Intel. This implementation reuses the back-end of the OCaml compiler. At the end of 2007, Intel decided to progressively translate their reFLect code base to OCaml.

In 2008, Michel Mauny and Nicolas Pouillard developed a translator from reFLect to OCaml, based on the CaFL front-end for parsing and type-checking of source programs, a CamlP4-based translator of CaFL abstract syntax trees to OCaml syntax, and a post-processor that makes explicit lazy evaluation by inserting `lazy` and `force` instructions at the appropriate places. The main technical challenges are the automatic generation of OCaml bindings for the thousands of reFLect primitives, and the efficiency and practicality of the "lazy" OCaml library.

Software migration may need to be progressive, especially when the code base to be translated is big. In such situations, we may want to translate only parts of the original applications, linking two runtime systems into one, and using functionalities of the old system as primitives in the new one, in a typed way. Michel Mauny and Nicolas Pouillard designed a mechanism that allows the automatic import/export of functions and data structures between two languages. Based on a few low-level primitives for each data type, the transfer of values of a data type proceeds by automatically translating the values from one runtime to another: marshalling when the target language provides an implementation for the data, coercing to an abstract data type otherwise. This mechanism has been implemented by Nicolas Pouillard for the reFLect to OCaml translator, and its formal properties are still under investigation.

To implement the lazy semantics of the reFLect language, the reFLect to OCaml translator mechanically generates delays and force operations in OCaml. On the one hand, as it is generally the case in generated programs, many of the delays can be optimized away, when the evaluation to be delayed is trivial. Michel Mauny modified the Objective Caml compiler for those situations to be detected. On the other hand, the encoding of lazy evaluation on a realistic language with algebraic data structures and pattern-matching involves a tight interaction between matching (control and bindings) and forcing the evaluation of components of the values being matched. In order to make simpler this interaction, Michel Mauny extended the OCaml notion of pattern with so-called "lazy patterns" which integrate forced evaluation of lazy values inside pattern-matching, and extended the compilation of pattern-matching accordingly.

# 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 current members of the Consortium are: CEA, Citrix, Dassault Aviation, Dassault Systèmes, LexiFi, Intel, Jane Street Capital, Microsoft, and OCamlCore. For a complete description of this structure, refer to http://caml.inria.fr/consortium/. Xavier Leroy chairs the scientific committee of the Consortium.

## 7.2. ReFLect over Caml

We have a contract with Intel Corporation that supports Nicolas Pouillard and Michel Mauny's work on formally defining and implementing the reFLect functional programming language designed at Intel. Michel Mauny is the principal investigator for this contract.

# 8. Other Grants and Activities

## 8.1. National initiatives

The Gallium project coordinates an *Action de Recherche Amont* in the programme *Sécurité des systèmes embarqué et Intelligence Ambiante* funded by the *Agence Nationale de la Recherche*. This 3-year action (2005-2008) is named "Compcert" and involves the Gallium and Marelle INRIA projects as well as CNAM/ENSIIE (Cedric laboratory) and University Paris 7/CNRS (PPS laboratory). The theme of this action is the mechanized verification of compilers and the development of supporting tools for specification and proof of programs. Xavier Leroy is the principal investigator for this action.

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 CEA LIST and involves Xavier Leroy at Gallium. It studies issues related to floating-point arithmetic in static analyzers and verified compilers.

The Gallium project is involved in two *Actions de Recherche Collaborative*. The "Quotient" ARC (2007–2008) is coordinated by Frédéric Blanqui at INRIA Lorraine and involves Damien Doligez on Gallium's side. Its purpose is to study and develop an intermediate between concrete data types and abstract data types in an extension of OCaml. The "CeProMi" ARC (2008–2009) is coordinated by Claude Marché at INRIA Saclay and involves François Pottier and Arthur Charguéraud. It focuses on deductive verification of imperative, functional and object-oriented programs.

# 9. Dissemination

## 9.1. Interaction with the scientific community

### *9.1.1. Collective responsibilities within INRIA*

François Pottier is a member of INRIA's COST (*Conseil d'Orientation Scientifique et Technologique*).

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

Didier Rémy was vice-chairman of the hiring committee for the INRIA Paris-Rocquencourt CR2 competition.

### *9.1.2. Collective responsibilities outside INRIA*

Sandrine Blazy is a member of the Board (*conseil d'administration*) of ENSIIE.

### *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.

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

### 9.1.4. Program committees and steering committees

Sandrine Blazy chaired the program committee for the Journées Francophones des Langages Applicatifs (JFLA 2008). She served on the program committee for the AFADL 2009 conference (Approches Formelles dans l'Assistance au Développement de Logiciels).

Xavier Leroy served on the program committee of the European Symposium on Programming (ESOP) 2009. He is a member of the steering committee of the ML workshop and the Commercial Users of Functional Programming workshop.

François Pottier is a member of the steering committees for the ICFP conference and for the ML workshop.

Didier Rémy participated in the program committees of the ML workshop (ML 2008) held in September in Victoria, Canada in conjunction with the ICFP conference, and of the Types in Language Design and Implementation workshop (TLDI 2009) to be held in Savannah, USA September in Victoria, USA in conjunction with the POPL conference.

### 9.1.5. Ph.D. and habilitation juries

Damien Doligez was a member of the Ph.D. defense committee for Jean-Frédéric Étienne (CNAM, july 2008).

Xavier Leroy was a reviewer (*rapporteur*) for the Ph.D. theses of Guillaume Salagnac (University of Grenoble, april 2008) and Thierry Hubert (University Paris Sud, june 2008). He was a member of the jury for the Habilitation of Sandrine Blazy (University of Évry, october 2008) and for the Ph.D. of Tiphaine Turpin (University of Rennes, december 2008).

François Pottier was a reviewer for the Ph.D. theses of Gwenaël Delaval (INP Grenoble, july 2008) and Matthieu Sozeau (University Paris Sud, december 2008). He was a member of the jury for the Ph.D. defense of Sylvain Lebresne (University Paris 7, december 2008).

Didier Rémy was a member of the Ph.D. defense committee of Boris Yakobowski (University Paris 7, december 2008).

### 9.1.6. Learned societies

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

## 9.2. Teaching

### 9.2.1. Supervision of Ph.D. and internships

Sandrine Blazy supervises Benoît Robillard's Ph.D. in cooperation with Eric Soutif (CNAM).

Xavier Leroy is Ph.D. advisor for Zaynah Dargaye, Jean-Baptiste Tristan, and Tahina Ramananandro.

François Pottier is Ph.D. advisor for Arthur Charguéraud (since 2007), Alexandre Pilkiewicz (since december 2008), and Nicolas Pouillard (since september 2008). He supervised the master's internship of Alexandre Pilkiewicz from july to november 2008.

Didier Rémy is Ph.D. advisor for Benoît Montagu and Boris Yakobowski.

### 9.2.2. 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 7, 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*.

François Pottier, Didier Rémy and Giuseppe Castagna (PPS) taught a 24-hour lecture on functional programming languages and type systems at the MPRI, attended by 15 students.

### *9.2.3. Undergraduate courses*

Since september 2008, Zaynah Dargaye is an instructor (*ATER*) at ENSIIE in Évry.

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

Boris Yakobowski was teaching assistant at university Paris 7.

### *9.2.4. Other courses*

Xavier Leroy gave a one-hour lecture titled *Du langage à l'action: compilation et typage* at Collège de France, in Gérard Berry's course, *Pourquoi et comment le monde devient numérique*.

## 9.3. Participation in conferences and seminars

### *9.3.1. Participation in conferences*

POPL: Principles of Programming Languages (San Francisco, USA, january).
Arthur Charguéraud presented [15]. Jean-Baptiste Tristan presented [24]. Xavier Leroy, François Pottier, Didier Rémy attended.

JFLA: Journées Francophones des Langages Applicatifs (Étretat, France, january).
Boris Yakobowski presented [29]. Sandrine Blazy attended.

Seminar on Types, Logics and Semantics for State (Dagstuhl, Germany, february).
Arthur Charguéraud gave a presentation. François Pottier participated.

DTP: workshop Dependently Typed Programming (Nottingham, GB, march).
Xavier Leroy presented an invited talk on compiler verification.

TTVSI: workshop Tools and Techniques for Verification of System Infrastructure (London, GB, march).
Sandrine Blazy presented a poster on the Clight language and its formal semantics. Xavier Leroy presented an invited talk on compiler verification.

PLDI: Programming Languages Design and Implementation (Tucson, USA, june).
Xavier Leroy attended.

LCTES: Languages, Compilers, and Tools for Embedded Systems (Tucson, USA, june).
Xavier Leroy presented an invited talk on compiler verification.

Meeting of IFIP Working Group 2.8 "Functional Programming" (Park City, USA, june).
Didier Rémy and François Pottier participated.

LICS: Logic in Computer Science (Pittsburgh, USA, June).
François Pottier presented a paper [22].

CAV: Computer Aided Verification; satellite workshops SMT, AFM (Princeton, USA, july).
Sandrine Blazy attended.

IJCAR: International Joint Conference on Automated Reasoning (Sydney, Australia, august).
Damien Doligez attended the conference and entered Zenon in the CASC-J4 prover competition hosted by the conference.

ICFP: International Conference on Functional Programming (Victoria, Canada, october).
Arthur Charguéraud presented [17]. Boris Yakobowski presented [25]. Damien Doligez, Alexandre Pilkiewicz, Nicolas Pouillard and Didier Rémy attended.

MLW: Workshop on ML and its Applications (Victoria, Canada, october).
Damien Doligez presented [19]. Nicolas Pouillard and Didier Rémy attended.

Meeting of the GDR “Génie de la Programmation et du Logiciel”, working group “Langages, Types et Preuves” (Évry, France, november).
Sandrine Blazy served on the organizing committee. Arthur Charguéraud and Zaynah Dargaye gave one presentation each. Xavier Leroy and Francois Pottier attended.

### 9.3.2. Invitations and participation in seminars

Sandrine Blazy gave a talk about the Compcert memory model at the seminar of the LSL team (CEA, march), and a talk about the formalization of separation logic in Coq at a working meeting on separation logic (Princeton University, july).

Sandrine Blazy and Xavier Leroy were invited to present the CompCert verified compiler at an internal workshop of Airbus (Toulouse, december).

Benoît Montagu gave a talk on open existential types for module systems at the LAMA seminar (University of Chambéry, june).

Didier Rémy visited Andrew Tolmach at Portland University, where he gave a seminar on MLF, and Daan Leijen at Microsoft Research in Redmond, where he gave a seminar on open existential types.

### 9.3.3. Participation in summer schools

Arthur Charguéraud and Benoît Montagu attended the Oregon Summer School on Logic and Programming Languages (Eugene, Oregon, USA, July).

## 9.4. 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.

Maxence Guesdon maintains the Caml web site (http://caml.inria.fr/) and especially the Caml Humps (http://caml.inria.fr/humps/), a comprehensive Web index of about 500 Caml libraries, tools and tutorials contributed by Caml users. This Web site contributes significantly to the visibility of the Caml language.

# 10. Bibliography

## Major publications by the team in recent years

[1] T. HIRSCHOWITZ, X. LEROY. *Mixin modules in a call-by-value setting*, in "ACM Transactions on Programming Languages and Systems", vol. 27, n$^{o}$ 5, 2005, p. 857–881, http://gallium.inria.fr/~xleroy/publi/mixins-cbv-toplas.pdf.

[2] 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, p. 27–38, http://gallium.inria.fr/~remy/work/mlf/icfp.pdf.

[3] X. LEROY, H. GRALL. *Coinductive big-step operational semantics*, in "Information and Computation", accepted for publication, to appear in the special issue on Structural Operational Semantics, 2007, http://gallium.inria.fr/~xleroy/publi/coindsem-journal.pdf.

[4] X. LEROY. *Formal certification of a compiler back-end, or: programming a compiler with a proof assistant*, in "33rd ACM symposium on Principles of Programming Languages", ACM Press, 2006, p. 42–54, http://gallium.inria.fr/~xleroy/publi/compiler-certif.pdf.

[5] 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://gallium.inria.fr/~fpottier/publis/fpottier-pure-freshml.ps.gz.

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

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

[8] D. RÉMY. *Using, Understanding, and Unraveling the OCaml Language*, in "Applied Semantics. Advanced Lectures", G. BARTHE (editor), Lecture Notes in Computer Science, vol. 2395, Springer, 2002, p. 413–537.

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

## Year Publications

### Doctoral Dissertations and Habilitation Theses

[10] S. BLAZY. *Sémantiques formelles*, Habilitation à diriger les recherches, Université Évry Val d'Essone, October 2008, http://www.ensiie.fr/~blazy/hdr.html.

[11] B. YAKOBOWSKI. *Graphical types and constraints: second-order polymorphism and inference*, Ph. D. Thesis, University of Paris 7, December 2008, http://www.yakobowski.org/phd-dissertation.html.

### Articles in International Peer-Reviewed Journal

[12] B. DOLIGEZ, A. BERTHOULY, D. DOLIGEZ, M. TANNER, V. SALADIN, D. BONFILS, H. RICHNER. *Spatial scale of local breeding habitat quality and adjustment of breeding decisions in a wild tit population*, in "Ecology", vol. 89, n$^{o}$ 5, 2008, p. 1436–1444, http://www.esajournals.org/doi/abs/10.1890/07-0113.1.

[13] X. LEROY, S. BLAZY. *Formal verification of a C-like memory model and its uses for verifying program transformations*, in "Journal of Automated Reasoning", vol. 41, n$^{o}$ 1, 2008, p. 1–31, http://dx.doi.org/10.1007/s10817-008-9099-0.

[14] L. RIDEAU, B. P. SERPETTE, X. LEROY. *Tilting at windmills with Coq: Formal verification of a compilation algorithm for parallel moves*, in "Journal of Automated Reasoning", vol. 40, n$^{o}$ 4, 2008, p. 307–326, http://dx.doi.org/10.1007/s10817-007-9096-8.

### International Peer-Reviewed Conference/Proceedings

[15] B. AYDEMIR, A. CHARGUÉRAUD, B. C. PIERCE, R. POLLACK, S. WEIRICH. *Engineering Formal Metatheory*, in "Proceedings of the 35th ACM Symposium on Principles of Programming Languages (POPL'08)", ACM Press, January 2008, p. 3–15, http://doi.acm.org/10.1145/1328897.1328443.

[16] J. BRUNEL, D. DOLIGEZ, R. R. HANSEN, J. L. LAWALL, G. MULLER. *A Foundation for Flow-Based Program Matching Using Temporal Logic and Model Checking*, in "Proceedings of the 36th ACM Symposium on Principles of Programming Languages (POPL'09)", To appear, ACM Press, January 2009.

[17] 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.

[18] K. CHAUDHURI, D. DOLIGEZ, L. LAMPORT, S. MERZ. *A TLA+ Proof System*, in "Proceedings of the LPAR Workshops: Knowledge Exchange: Automated Provers and Proof Assistants, and The 7th International Workshop on the Implementation of Logics", G. SUTCLIFFE, P. RUDNICKI, R. SCHMIDT, B. KONEV, S. SCHULZ (editors), CEUR Workshop Proceedings, vol. 418, CEUR-WS.org, November 2008, http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-418/paper2.pdf.

[19] P. CUOQ, D. DOLIGEZ. *Hashconsing in an Incrementally Garbage-Collected System: A Story of Weak Pointers and Hashconsing in OCaml 3.10.2*, in "2008 ACM SIGPLAN Workshop on ML", ACM Press, September 2008, p. 13–22, http://doi.acm.org/10.1145/1411304.1411308.

[20] J. N. FOSTER, A. PILKIEWICZ, B. C. PIERCE. *Quotient Lenses*, in "Proceedings of the 13th International Conference on Functional Programming (ICFP'08)", ACM Press, September 2008, p. 383–396, http://doi.acm.org/10.1145/1411204.1411257.

[21] 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)", To appear, ACM Press, January 2009, http://gallium.inria.fr/~remy/modules/Montagu-Remy@popl09:fzip.pdf.

[22] 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://doi.ieeecomputersociety.org/10.1109/LICS.2008.16.

[23] Y. RÉGIS-GIANAS, F. POTTIER. *A Hoare logic for call-by-value functional programs*, in "Mathematics of Program Construction, 9th International Conference, MPC 2008", Lecture Notes in Computer Science, vol. 5133, Springer, July 2008, p. 305–335, http://dx.doi.org/10.1007/978-3-540-70594-9_17.

[24] 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.

[25] B. YAKOBOWSKI, D. RÉMY. *Graphic Type Constraints and Efficient Type Inference: from ML to MLF*, in "Proceedings of the 13th International Conference on Functional Programming (ICFP'08)", ACM Press, September 2008, p. 63–74, http://doi.acm.org/10.1145/1411204.1411216.

### National Peer-Reviewed Conference/Proceedings

[26] P. AYRAULT, M. CARLIER, D. DELAHAYE, C. DUBOIS, D. DOLIGEZ, L. HABIB, T. HARDIN, M. JAUME, C. MORISSET, F. PESSAUX, R. RIOBOO, P. WEIS. *Trusted Software within Focal*, in "Computer & Electronics Security Applications Rendez-vous (C&SAR 2008)", P. CHOUR, Y. CORREC, O. HEEN, L. MÉ (editors), December 2008, p. 142–158, http://www-spi.lip6.fr/~jaume/cesar.pdf.

[27] S. BLAZY, B. ROBILLARD, É. SOUTIF. *Coloration avec préférences: complexité, inégalités valides et vérification formelle*, in "ROADEF'08, 9e congrès de la Société Française de Recherche Opérationnelle et d'Aide à la Décision", February 2008, http://hal.inria.fr/inria-00260712.

[28] S. BLAZY, B. ROBILLARD, É. SOUTIF. *Vérification formelle d'un algorithme d'allocation de registres par coloration de graphes*, in "Journées Francophones des Langages Applicatifs (JFLA'08), Étretat, France", INRIA, January 2008, p. 31–46, http://hal.inria.fr/inria-00202713/.

[29] B. YAKOBOWSKI. *Le caractère "backquote" à la rescousse – Factorisation et réutilisation de code grâce aux variants polymorphes*, in "Journées Francophones des Langages Applicatifs (JFLA'08), Étretat, France", INRIA, January 2008, p. 63–78, http://hal.inria.fr/inria-00202817/.

### Research Reports

[30] X. LEROY, D. DOLIGEZ, J. GARRIGUE, D. RÉMY, J. VOUILLON. *The Objective Caml system, documentation and user's manual – release 3.11*, INRIA, December 2008, http://caml.inria.fr/pub/docs/manual-ocaml/.

### Other Publications

[31] S. BLAZY, X. LEROY. *Mechanized semantics for the Clight subset of the C language*, Submitted, October 2008, http://gallium.inria.fr/~xleroy/publi/Clight.pdf.

[32] S. BLAZY, B. ROBILLARD. *Live-range unsplitting for faster optimal coalescing*, Submitted, October 2008.

[33] A. CHARGUÉRAUD. *Interactive Verification of Call-by-Value Functional Programs*, Submitted, November 2008, http://arthur.chargueraud.org/research/2009/deep/deep.pdf.

[34] X. LEROY. *A formally verified compiler back-end*, Submitted, July 2008, http://gallium.inria.fr/~xleroy/publi/compcert-backend.pdf.

[35] K. NAKATA. *Lazy mixins and disciplined effects*, Submitted, July 2008, http://gallium.inria.fr/~nakata/papers/Lyre08.pdf.

[36] K. NAKATA. *Lazy modules: A lazy evaluation strategy for more recursive initialization patterns*, Submitted, September 2008, http://gallium.inria.fr/~nakata/papers/Osan08.pdf.

[37] A. PILKIEWICZ. *Towards a foundational type system for complexity analysis*, Master's dissertation (mémoire de stage de Master 2), École Polytechnique, November 2008.

[38] D. RÉMY, B. YAKOBOWSKI. *A Church-Style Intermediate Language for MLF*, Submitted, September 2008, http://gallium.inria.fr/~remy/mlf/xmlf.pdf.

[39] J.-B. TRISTAN, X. LEROY. *Verified Validation of Lazy Code Motion*, Submitted, November 2008, http://gallium.inria.fr/~xleroy/publi/validation-LCM.pdf.

## References in notes

[40] A. W. APPEL, S. BLAZY. *Separation logic for small-step Cminor*, in "Theorem Proving in Higher Order Logics, 20th Int. Conf. TPHOLs 2007", Lecture Notes in Computer Science, vol. 4732, Springer, 2007, p. 5–21, http://www.ensiie.fr/~blazy/AppelBlazy07.pdf.

[41] 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.

[42] S. BLAZY, Z. DARGAYE, X. LEROY. *Formal Verification of a C Compiler Front-End*, in "FM 2006: Int. Symp. on Formal Methods", Lecture Notes in Computer Science, vol. 4085, Springer, 2006, p. 460–475, http://gallium.inria.fr/~xleroy/publi/cfront.pdf.

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

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

[45] D. LE BOTLAN, D. RÉMY. *Recasting MLF*, Research report, n$^{o}$ 6228, INRIA, June 2007, http://hal.inria.fr/inria-00156628.

[46] D. LE BOTLAN. *MLF: Une extension de ML avec polymorphisme de second ordre et instanciation implicite.*, Ph. D. Thesis, École Polytechnique, May 2004, http://www.inria.fr/rrrt/tu-1071.html.

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

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

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

[50] D. RÉMY, J. VOUILLON. *Objective ML: A simple object-oriented extension to ML*, in "24th ACM Conference on Principles of Programming Languages", ACM Press, 1997, p. 40–53.

[51] D. RÉMY, B. YAKOBOWSKI. *A graphical presentation of MLF types with a linear-time incremental unification algorithm.*, in "ACM SIGPLAN Workshop on Types in Language Design and Implementation", ACM Press, January 2007, p. 27–38, http://gallium.inria.fr/~remy/project/mlf/mlf-graphic-types.pdf.