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

Activity Report

2009

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]
Na Xu [CR INRIA, since November 2009]

Faculty Member
Roberto Di Cosmo [Professor, on leave from U. Paris 7 Diderot, since September 2009, HdR]

External Collaborator
Sandrine Blazy [Assistant professor, ENSIIE, until August 2009, HdR]
Michel Mauny [Professor, ENSTA]

Technical Staff
Xavier Clerc [IR INRIA, SED, 40% part time]

PhD Student
Arthur Charguéraud [AMN grant, U. Paris 7 Diderot]
Zaynah Dargaye [ATER, ENSIIE, until August 2009]
Benoît Montagu [AMX grant, Polytechnique]
Alexandre Pilkiewicz [AMX grant, Polytechnique]
Nicolas Pouillard [Digiteo grant, INRIA]
Tahina Ramananandro [AMN grant, U. Paris 7 Diderot]
Jean-Baptiste Tristan [ANR grant, INRIA, until September 2009]

Visiting Scientist
David MacQueen [Professor, on sabbatical from U. Chicago, March–April 2009]

Administrative Assistant
Stéphanie Aubin [TR INRIA, until April 2009]
Stéphanie Chaix [Temporary personnel, since June 2009]

Other
Paolo Herms [M2 graduate intern, March–July 2009]
Silvain Rideau [ENS undergraduate intern, June–July 2009]

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 [41]. 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 [43] 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 [39], [34], 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 [48], 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 [44].

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 [46].

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

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 [38]. 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 [Lex-iFi], Jacques Garrigue [Kyoto University], Maxence Guesdon [team SED], Luc Maranget [EPI Moscova], Michel Mauny, 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

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 and ARM 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 [university 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. Semantics and type systems for imperative programs

6.1.1. Semantic analysis of hidden state in imperative programs

Participants: François Pottier, Lars Birkedal [IT Univ. of Copenhagen], Jan Schwinghammer [Saarland Univ.], Hongseok Yang [Queen Mary Univ. of London], Bernhard Reus [Univ. of Sussex].

In collaboration with Lars Birkedal, Jan Schwinghammer, Hongseok Yang, and Bernhard Reus, François Pottier studied a semantic model of a separation logic equipped with higher-order frame and anti-frame rules. The purpose of these rules is to enable local reasoning on functions and higher-order functions that maintain and operate upon *hidden state*: mutable data structures that persist between function invocations and are used only internally by the function, but not directly exposed to the caller.

The anti-frame rule was introduced by Pottier in earlier work [6]. At the time, a syntactic proof of its soundness was only sketched. This new work uses a very different proof technique, namely the construction of a semantic model, in order to establish the soundness of the rule. A paper describing this result was accepted for presentation at the FoSSaCS 2010 conference [23].

6.1.2. Semantic analysis of general references

Participant: François Pottier.

The construction of a semantic model for a typed programming language implicitly contains a translation of this language into a mathematical meta-language. In some cases, it is possible to make this translation explicit and to view it as a type-preserving translation of the programming language into a well-defined, typed core calculus.

Inspired by this idea, which arose from his collaborative work with Schwinghammer *et al.*, François Pottier developed a type-preserving translation of System F, equipped with general references, into a typed core calculus, baptised FORK, which can be defined as an extension of System F_ω with certain recursive kinds. The translation is a type-preserving, store-passing translation. It is analogous to the classic monadic translation, but is significantly more complex, because it deals with dynamic memory allocation and higher-order store. It is in fact the first type-preserving store-passing translation that deals with these features. A prototype FORK type-checker was developed and used to check the validity of the translation.

6.1.3. A type sytem for monotonicity

Participants: Alexandre Pilkiewicz, François Pottier.

Alexandre Pilkiewicz and François Pottier studied the notion of *monotonicity* in type systems.

Last year, Charguéraud and Pottier designed a type system, based on capabilities, allowing a powerful management of mutable locations in imperative programs [1]. The type of such locations can be freely modified at the cost of a fine-grained control of aliasing, thus prohibiting a function from having a private hidden state. Then, Pottier presented a extension of this type system—the *anti-frame* rule—allowing hidden state [6]. Mutable state can be seen as freely aliased at the cost of an immutable type.

This year, Alexandre Pilkiewicz and François Pottier explored a middle ground that allows to modify the type of an aliased location, provided this is done in a monotonic way. The new value can only have a type that is more precise that the one of the old value, so that any assumption made by other clients of the mutable location remains correct.

To achieve this, they developed a notion of *fates* that can be viewed as ghost logical references whose contents can only evolve monotonically according to a user-defined law. A fate can be associated with a run-time variable, forcing its content to follow the same law. It is then possible to state logical properties about the future evolutions of this run time variable through *predictions* over its fate. A paper describing this approach was submitted [32].

6.2. Partial type inference with first-class polymorphism

Participants: Didier Rémy, Boris Yakobowski [University Paris 7 Diderot], Paolo Herms.

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 this year in *Information and Computation* [14]. The initial MLF design was then simplified and made more expressive by Boris Yakobowski in his Ph.D. dissertation [49], 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 ML, that can be used both to show the preservation of types during reduction and as an internal language for MLF [33].

Paolo Herms (master's intern from University of Pisa) and Didier Rémy proposed an extension of MLF with higher-order types. To avoid the problem of undecidable second and higher-order unification that would appear with implicit higher-order types, the implicit type abstraction and type application mechanism of MLF are first combined with the original explicit type abstraction and type application mechanism of System F, and explicit coercions between the two forms. Then, higher-order types can be added but only in the explicit forms. This gives MLF$^\omega$ the power of System F$^\omega$ while still retaining the possibility of leaving introduction and elimination of polymorphism at second-order types implicit. Paolo Herms showed how to adapt graphical types to cope with MLF$^\omega$ [31].

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 difficult to formalize and also often become harder to use in larger, more complex but practical examples. In fact, useful extensions such as recursive modules or mixin modules are technically challenging and still an active topic of research.

This persisting gap between the apparent simplicity and formal complexity of modules is surprising. Benoît Montagu and Didier Rémy have identified at least two orthogonal sources of complexity, width-wise and depth-wise. 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, in order 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.

Benoît Montagu and Didier Rémy's work aims at providing a new presentation of modules that is conceptually more economical while retaining (or increasing) the expressiveness and conciseness of the actual approaches. This presentation relies 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 symposimum on Principles of Programming Languages (POPL 2009) that was held in Savannah, USA in January 2009 [21].

Benoît Montagu and Didier Rémy currently focus their efforts on the study of the singleton kinds system, more specifically on giving an alternative definition in which type equivalence would be based on $\beta\eta$-conversion. This work would give a more operational presentation of the singleton kinds system and hopefully facilitate extensions.

6.4. Formal verification of compilers

6.4.1. The Compcert verified compiler for the C language

Participants: Xavier Leroy, Sandrine Blazy [ENSIIE then U. Rennes 1].

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 and ARM architectures [17]. This compiler comprises a back-end part, translating the Cminor intermediate language to PowerPC assembly and reusable for source languages other than C [16], and a front-end translating the Clight subset of C to Cminor [35]. 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 establishing that the generated assembly code executes exactly as prescribed by the semantics of the source Clight program.

This year, we improved the Compcert C compiler in several ways. First and foremost, we added support for "goto" and labeled statements to the Clight language. While the compilation of "goto" is trivial, it was difficult to find a suitable operational semantics for this unstructured control operation. The solution was a transition semantics using continuations to represent the current control point within the program. The proofs of semantic preservation from Clight to C#minor to Cminor were entirely re-done with this new, extended semantics. Other improvements to the C compiler include:

- A new optimization pass: recognition of tail calls to functions.
- Small improvements in the speed and size of the generated code.
- Better modularization of the compiler between the architecture-independent and the architecture-dependent parts.
- Accounting for alignment constraints in the memory model.
- Reduced compilation times and compile-time memory requirements.

Two versions of the Compcert development were publically released, integrating these improvements: version 1.4 in April and 1.5 in August.

Three journal articles on the Compcert project were published this year: a short general overview in the *Research Highlights* column of *Communications of the ACM* [17]; a very detailed (80 pages) description of the back-end and its proof of correctness in *Journal of Automated Reasoning* [16]; a description of the Clight source language and the mechanization of its semantics, also in *Journal of Automated Reasoning* [13].

6.4.2. Verified translation validation

Participants: Jean-Baptiste Tristan, Silvain Rideau, 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 software pipelining. Software pipelining is a very ambitious loop optimization that exploits instruction level parallelism between consecutive iterations of a loop. This year, Jean-Baptiste Tristan succesfully designed, implemented, and verified a validator for a software pipeliner, the key component of the software pipelining optimization. The validator is based on symbolic evaluation, here applied for the first time to the verification of a loop optimization. This result was accepted for publication at the forthcoming POPL 2010 symposium [25].

Jean-Baptiste Tristan completed his Ph.D. dissertation, titled *Formal Verification of Translation Validators* [12] and successfully defended in November. The dissertation consolidates the results of four case studies of verified validation applied to advanced optimizations: lazy code motion, list scheduling, trace scheduling, and software pipelining.

Silvain Rideau, first-year student at ENS Paris supervised by Xavier Leroy, developed and proved correct another translation validator dedicated to the register allocation and spilling/reloading phases of a compiler. While state-of-the-art register allocation algorithms have been formally verified before in the context of the Compcert project [16] [36], the spilling and reloading algorithm that has been verified so far is quite naive and inappropriate for register-poor target processors such as the popular x86 architecture. Silvain Rideau designed a translation validation algorithm, based on backward dataflow analysis, that can validate *a posteriori* the results of register allocation with aggressive spilling and reloading strategies, including live range splitting, reuse of reloaded quantities, and coalescing. He proved the soundness of this validator using the Coq proof assistant, and validated it experimentally on a nontrivial spilling strategy involving live range splitting at each use and definition of a spilled temporary. These results were accepted for publication at the Compiler Construction 2010 conference [22].

6.4.3. Verified compilation of object-oriented languages

Participants: Tahina Ramananandro, 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, Tahina Ramananandro tackled the issue of formally verifying object layout and management in class-based, single-inheritance languages. This is a step towards building a formally verified static compiler from a subset of Java bytecode to RTL (a CFG-style intermediate language of the CompCert back-end). Then, Tahina Ramananandro has been extending this formalization to multiple inheritance to deal with a subset of C++ as a source language.

6.5. Program specification and proof

6.5.1. Characteristic formulae for modular verification of functional programs

Participant: Arthur Charguéraud.

In previous work [37], Arthur Charguéraud has proposed a framework for modular verification of purely functional OCaml code using the Coq proof assistant. The approach relied 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.

This year, Arthur Charguéraud has introduced *characteristic formulae*, which are higher-order logic formulae that can be used to reason on programs. Intuitively, characteristic formulae can be viewed as an abstract layer built on the top of a deep embedding. By hiding all the technical details associated with deep embeddings, characteristic formulae make program verification significantly easier.

The result of this work is a practical, sound and complete approach to the compositional verification of total correctness properties. It applies to call-by-value, well-typed, purely functional programs, and relies on the use of a standard higher-order logic proof assistant, namely Coq, for carrying out formal reasoning.

Characteristic formulae are described in an article submitted for publication [29]. A characteristic formulae generator has been implemented in OCaml and used to formally prove correct implementations of advanced purely-functional data structures such as persistent real-time queues.

6.5.2. 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. The major extensions this year include: support for TLA+'s character strings; support for CASE expressions; support for Hilbert's choice operator. Zenon's handling of equality was also improved.

6.5.3. Tools for TLA+

Participants: Damien Doligez, Leslie Lamport [Microsoft Research], Stephan Merz [EPI 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 [40], and to build tools for writing TLA+ specifications and mechanically checking the corresponding formal proofs.

This year, the TLA+ project released the first version of the TLA+ tools: the GUI-based TLA Toolbox and the TLA+ Proof System, and an environment for writing and checking TLA+ proofs. Kaustuv Chaudhuri, the post-doc who developped the Proof System, left in November. He is replaced by Denis Cousineau, on a 2-year post-doc position. Dan Ricketts was hired as an engineer to continue the development of the Toolbox, replacing Simon Zambrowski who left in October.

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 very often: 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.

At first sight, it seems very easy to represent binders in general purpose programming languages. Several approaches are known: named variables; de-Bruijn indices; the "locally nameless" technique (a combination of both); higher order abstract syntax; etc. However, with more experience, it appears that the obvious representations are hard to manage, and that delicate programming errors occur often.

Our initial goal was to improve one of the known languages that provides support for binders: Pure FreshML [5]. We obtained a clean and quite general view of data structures with binders. In this presentation, data structures are indexed by a so-called "world". In particular, the type for names (called atom) is also indexed. Then, name abstraction is achieved via an existential type for the inner world and a safe way to move across this abstraction frontier.

We gave a presentation of this ongoing work at the Computational Applications of Nominal Sets seminar at Cambridge University. We are currently working on finishing the last details of a very core system and mechanically proving its soundness using the AGDA proof system. This work will soon be submitted to a conference.

6.7. The Objective Caml system

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

This year, we released version 3.11.1 of the Objective Caml system. This is a minor release that corrects about 50 problem reports. Damien Doligez acted as release manager for this version.

Three important extensions of the Caml language were designed and implemented, and will be made available in the next major release of Objective Caml:

- 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. (Jacques 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 [47]. (Alain 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. (Alain Frisch.)

Xavier Clerc, research engineer at SED, joined the Caml development team at 40% of his time in early 2009. He improved and updated the replay debugger, studied cross-compilation issues, restructured the Caml test suite, and participated in bug-fixing.

Xavier Leroy and Xavier Clerc converted the Caml source repository from CVS to Subversion, making it easier for non-INRIA developers to access the repository.

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 10 member companies: CEA, Citrix, Dassault Aviation, Dassault Systèmes, LexiFi, Intel, Jane Street Capital, Microsoft, 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 CIRILL laboratory

Roberto Di Cosmo is working on the creation of the CIRILL (Centre d'Innovation et Recherche sur le Logiciel Libre), also known as FSRII (Free Software Research and Innovation Institute), which 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), and to industry players willing to collaborate with researchers on these issues. The CIRILL is supported by INRIA, and its activities are expected to start in the first semester of 2010.

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. Sandrine Blazy and Xavier Leroy are 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.

The Gallium project is involved in the "CeProMi" *Actions de Recherche Collaborative*. This ARC (2008–2009) is coordinated by Claude Marché at INRIA Saclay and involves François Pottier, Arthur Charguéraud and Alexandre Pilkiewicz. It focuses on deductive verification of imperative, functional and object-oriented programs.

Finally, 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

Xavier Leroy was a member of the hiring committee for the INRIA Paris-Rocquencourt CR2 and CR1 competitions.

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

François Pottier is a member of the selection committee for post-doctoral grants at INRIA Paris-Rocquencourt.

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 a member of the hiring committee for the joint Chair position between INRIA and University of Bordeaux.

9.1.2. Collective responsibilities outside INRIA

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

Roberto Di Cosmo acted as an expert for the selection of the Integrated Projects in Objective 1.2 of the Call 5 of the FP7 european research program.

Roberto Di Cosmo was head of graduate studies for Computer Science at University Paris Diderot.

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

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 served on the program committee for the AFADL 2009 (Approches Formelles dans l'Assistance au Développement de Logiciels) conference.

Damien Doligez was a member of the program comittees of the ACM 2009 workshop on ML and of the JFLA 2010 conference.

Xavier Leroy served on the program committees of the ACM International Conference on Functional Committee (ICFP 2009), of the Theorem Proving in Higher-Order Logics (TPHOLs 2009) conference, and of the ACM Programming Languages meet Program Verification (PLPV 2010) workshop.

François Pottier served on the program committees of the Logic in Computer Science (LICS) 2009 conference, the Functional and Logic Programming (FLOPS) 2010 symposium, the Programming Languages and Analysis for Security (PLAS) 2009 worksop, and the 2009 workshop on Mechanized Metatheory (WMM). He is a member of the steering committee for the International Conference on Functional Programming (ICFP).

Didier Rémy served on the program committee of the 2010 European Symposium on Programming (ESOP).

9.1.5. Ph.D. and habilitation juries

Roberto Di Cosmo was president of the Ph.D. jury of Jean-Baptiste Tristan (University Paris Diderot, november 2009) and a member of the Ph.D. juries of Benoît Razet (University Paris Diderot, november 2009) and Denis Cousineau (Ecole Polytechnique, december 2009).

Xavier Leroy was external examiner (*rapporteur*) for the Ph.D. theses of Yannick Moy (University Paris Sud, january 2009), Cesar Kunz (École des Mines ParisTech, february 2009), and Magnus Myreen (Cambridge University, april 2009). He was a member of the Ph.D. juries of his students Zaynah Dargaye (University Paris Diderot, july 2009) and Jean-Baptiste Tristan (University Paris Diderot, november 2009).

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 17 collaborative R&D projects, with a budget over 30 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

Sandrine Blazy supervises Benoît Robillard's Ph.D. in cooperation with Eric Soutif (CNAM) and William Bartlett's Ph.D. in cooperation with Loïc Correnson (CEA).

Roberto Di Cosmo supervises the Ph.D. of Jaap Boender at University Paris Diderot.

Xavier Leroy was Ph.D. advisor for Zaynah Dargaye and Jean-Baptiste Tristan, who both defended this year. Xavier Leroy currently supervises Tahina Ramananandro's Ph.D. He also supervised the 2-month summer internship of Silvain Rideau (first-year student at ENS Paris).

François Pottier is Ph.D. advisor for Arthur Charguéraud (since 2007), Alexandre Pilkiewicz (since 2008), and Nicolas Pouillard (since 2008).

Didier Rémy is Ph.D. advisor for Benoît Montagu (since 2007). He supervised Paolo Herms master's internship (from University of Pisa) between april and july.

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*.

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

Roberto Di Cosmo gave a 12-hour lecture on Linear Logic at the MPRI (september-october 2009).

Sandrine Blazy taught a 22-hour lecture on abstract interpretation at the MOPS master curriculum of university of Évry.

Xavier Leroy taught a 4-hour course on mechanized semantics with applications to program proof and compiler verification at the Marktoberdorf summer school 2009, attended by about 80 Ph.D. students.

9.3.3. Undergraduate courses

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

Until september 2009, Zaynah Dargaye was an instructor (*ATER*) at ENSIIE in Évry.

Alexandre Pilkiewicz was teaching assistant at University Paris Diderot for the courses "Concepts Informatiques" and "Internet et outils".

Tahina Ramananandro was teaching assistant at University Paris Diderot for the courses "Datatypes and objects in Java" (first-year undergraduate students, 52 hours, february–june) and "Object-oriented programming with Java" (third-year undergraduate students, 26 hours, september–december).

9.4. Participation in conferences and seminars

9.4.1. Participation in international conferences

POPL: Principles of Programming Languages (Savannah, USA, january).
Benoît Montagu presented [21]. Xavier Leroy participated in a panel discussion on future trends in programming language research. Damien Doligez, Didier Rémy and Jean-Baptiste Tristan attended.

PLDI: Programming Languages Design and Implementation (Dublin, Ireland, june)
Jean-Baptiste Tristan presented [24]. Sandrine Blazy and Xavier Leroy attended.

LCTES: Languages, Compilers, and Tools for Embedded Systems (Dublin, Ireland, june)
Sandrine Blazy attended.

IFIP working group 2.8 "Functional programming" (Frauenchiemsee, Germany, june).
Xavier Leroy gave a talk on dependently-typed programming. Didier Rémy gave a talk on xMLF [33].

ICFP: International Conference on Functional Programming (Edimburgh, Great Britain, september).
Xavier Leroy and Didier Rémy attended.

MLW: ACM workshop on ML and its applications (Edimburgh, Great Britain, september).
Didier Rémy participated in a panel discussion on the future of ML. Xavier Leroy attended.

9.4.2. Participation in national conferences

AFADL: Approches Formelles dans l'Assistance au Développement de Logiciels (Toulouse, France, january)
Sandrine Blazy attended.

Plenary meeting of the GDR "Génie de la Programmation et du Logiciel" (Toulouse, France, january)
Xavier Leroy presented an invited talk on compiler verification. Sandrine Blazy attended.

JFLA: Journées Francophones des Langages Applicatifs (Saint-Quentin sur Isère, France, February).
Damien Doligez and Nicolas Pouillard attended.

OCaml users's meeting (Grenoble, France, february)
Xavier Leroy gave the opening talk. Xavier Clerc, Damien Doligez and Nicolas Pouillard attended.

Working group "Langages, types et preuves" of the GDR GPL (Créteil, France, october)
Benoît Montagu gave a presentation on first-class modules. François Pottier gave a presentation on System F_ω with recursive kinds. Didier Rémy attended.

Open World Forum (Paris, France, october)
Roberto Di Cosmo gave a talk on the Mancoosi project.

FOSSA: Free Open Source Software Academia Conference (Grenoble, France, november)
Roberto Di Cosmo gave a talk on "Mancoosi: helping communities build better packages".

9.4.3. Invitations and participation in seminars

Sandrine Blazy gave a talk about the CompCert memory model at the LIFO seminar (Orléans, january), a talk about the Clight formal semantics at the 68NQRT seminar (Rennes, march) and a talk about formal verification of register allocation at the LABRI seminar (Bordeaux, march).

Sandrine Blazy was invited to present the results of the CompCert project to the *Comité scientifique sectoriel STIC* of ANR (Paris, may).

Xavier Leroy was invited to speak at the Morgenstern colloquium of INRIA Sophia-Antipolis (october 2009).

Nicolas Pouillard gave a presentation at the Computational Applications of Nominal Sets seminar, Cambridge University.

9.4.4. Participation in summer schools

Tahina Ramananandro participated in the École des Jeunes Chercheurs en Programmation (EJCP) summer school (Dinard and Rennes, France, june 2009).

9.5. Other dissemination activities

Roberto Di Cosmo and Alexandre Pilkiewicz participated in the *Fête de la science* in november 2009. Roberto Di Cosmo gave a tutorial titled "the complexity of free software". Alexandre Pilkiewicz participated in an introduction to programming for the 6 to 20 age group.

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.

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", vol. 207, n^o 6, 2009, 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", vol. 43, n^o 4, 2009, 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", vol. 52, n^o 7, 2009, p. 107–115, http://doi.acm.org/10.1145/1538788.1538814.

[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://dx.doi.org/10.1109/LICS.2007.44.

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

[7] 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.

[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, http://gallium.inria.fr/~remy/cours/appsem/.

[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://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.

Year Publications

Doctoral Dissertations and Habilitation Theses

[11] Z. Dargaye. *Vérification formelle d'un compilateur optimisant pour langages fonctionnels*, University Paris Diderot (Paris 7), July 2009, Ph. D. Thesis.

[12] J.-B. Tristan. *Formal verification of translation validators*, University Paris Diderot (Paris 7), November 2009, http://tel.archives-ouvertes.fr/tel-00437582/, Ph. D. Thesis.

Articles in International Peer-Reviewed Journal

[13] S. Blazy, X. Leroy. *Mechanized semantics for the Clight subset of the C language*, in "Journal of Automated Reasoning", vol. 43, n^o 3, 2009, p. 263-288, http://dx.doi.org/10.1007/s10817-009-9148-3.

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

[15] X. LEROY, H. GRALL. *Coinductive big-step operational semantics*, in "Information and Computation", vol. 207, n^{o} 2, 2009, p. 284–304, http://dx.doi.org/10.1016/j.ic.2007.12.004.

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

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

International Peer-Reviewed Conference/Proceedings

[18] P. ABATE, J. BOENDER, R. DI COSMO, S. ZACCHIROLI. *Strong Dependencies between Software Components*, in "3rd International Symposium on Empirical Sofware Engineering and Measurement, ESEM 2009", IEEE Computer Society Press, October 2009, p. 89-99, http://dx.doi.org/10.1109/ESEM.2009.5316017.

[19] S. BLAZY, B. ROBILLARD. *Live-range Unsplitting for Faster Optimal Coalescing*, in "Proceedings of the ACM SIGPLAN/SIGBED 2009 conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2009)", ACM Press, 2009, p. 70–79, http://doi.acm.org/10.1145/1542452.1542462.

[20] 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)", ACM Press, January 2009, p. 114-126, http://doi.acm.org/10.1145/1480881.1480897DK.

[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)", ACM Press, January 2009, p. 354-365, http://doi.acm.org/10.1145/1480881.1480926.

[22] S. RIDEAU, X. LEROY. *Validating register allocation and spilling*, in "International Conference on Compiler Construction (CC 2010)", Lecture Notes in Computer Science, Springer, 2010, http://gallium.inria.fr/~xleroy/publi/validation-regalloc.pdf, Accepted for publication, to appear.

[23] J. SCHWINGHAMMER, H. YANG, L. BIRKEDAL, F. POTTIER, B. REUS. *A Semantic Foundation for Hidden State*, in "13th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS 2010)", Lecture Notes in Computer Science, Springer, 2010, http://gallium.inria.fr/~fpottier/publis/sfhs.pdf, Accepted for publication, to appear DK DE GB .

[24] J.-B. TRISTAN, X. LEROY. *Verified Validation of Lazy Code Motion*, in "Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09)", ACM Press, 2009, p. 316–326, http://doi.acm.org/10.1145/1542476.1542512.

[25] 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, http://gallium.inria.fr/~xleroy/publi/validation-softpipe.pdf, Accepted for publication, to appear.

Scientific Popularization

[26] R. DI COSMO. *La science du logiciel libre*, in "La Recherche", vol. 436, December 2009, Les cahiers de l'INRIA.

[27] R. DI COSMO. *Offrons aux jeunes les clés du pouvoir et de la liberté*, in "Science et Avenir", vol. 751, September 2009, p. 42-45, http://sciencesetavenirmensuel.nouvelobs.com/hebdo/sea/p751/articles/a407123-.html.

Other Publications

[28] A. W. APPEL, R. DOCKINS, X. LEROY. *A list-machine benchmark for mechanized metatheory*, July 2009, http://gallium.inria.fr/~xleroy/publi/listmachine-journal.pdf, Submitted US .

[29] A. CHARGUÉRAUD. *Formal Software Verification Through Characteristic Formulae*, October 2009, http://arthur.chargueraud.org/research/2009/cfg/cfg.pdf, Submitted.

[30] Z. DARGAYE, X. LEROY. *A verified framework for higher-order uncurrying optimizations*, March 2009, http://gallium.inria.fr/~xleroy/publi/higher-order-uncurrying.pdf, Submitted.

[31] P. HERMS. *Partial Type Inference with Higher-Order Types*, University of Pisa, July 2009, http://gallium.inria.fr/~remy/mlf/herms@master2009.pdf, Master's thesis.

[32] A. PILKIEWICZ, F. POTTIER. *The essence of monotonic state*, October 2009, http://gallium.inria.fr/~fpottier/publis/pilkiewicz-pottier-monotonicity-2009.pdf, Submitted.

[33] D. RÉMY, B. YAKOBOWSKI. *A Church-Style Intermediate Language for MLF*, October 2009, http://gallium.inria.fr/~remy/mlf/xmlf.pdf, Submitted.

References in notes

[34] 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.

[35] 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.

[36] 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://www.ensiie.fr/~blazy/JFLABRS08.pdf.

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

[38] 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.

[39] 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.

[40] 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.

[41] 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/.

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

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

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

[45] 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.

[46] 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.

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

[48] 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.

[49] 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/, Ph. D. Thesis.