Activity Report 2011

Project-Team GALLIUM

Programming languages, types, compilation and proofs

RESEARCH CENTER
Paris - Rocquencourt

THEME
Programs, Verification and Proofs

Table of contents

Project-Team GALLIUM

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

1. Members

Research Scientists

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

Faculty Member

Roberto Di Cosmo [Professor, on leave from U. Paris Diderot, until September 2011, HdR]

Technical Staff

Xavier Clerc [IR INRIA, SED, 30% part time]

PhD Students

Julien Cretin [AMX grant, U. Paris Diderot]
Alexandre Pilkiewicz [AMX grant, Polytechnique]
Nicolas Pouillard [Digiteo grant, INRIA]
Jonathan Protzenko [CORDI-S grant, INRIA]
Tahina Ramananandro [AMN grant, U. Paris Diderot]
Gabriel Scherer [ENS Paris, since October 2011]

Administrative Assistant

Stéphanie Chaix [Temporary personnel]

Others

Sylvain Dailler [M2 graduate intern, March–July 2011]
Jacques-Henri Jourdan [M2 graduate intern, March–September 2011]
Valentin Robert [M2 graduate intern, July–December 2011]

2. Overall Objectives

2.1. Introduction

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

2.2. Highlights

Xavier Leroy (EPI Gallium), Sandrine Blazy (EPI Celtique), Zaynah Dargaye (CEA) and Jean-Baptiste Tristan (Oracle Labs) were awarded the 2011 *La Recherche* prize in Information Sciences for their work on the CompCert verified compiler.

3. Scientific Foundations

3.1. Programming languages: design, formalization, implementation

Like all languages, programming languages are the media by which thoughts (software designs) are communicated (development), acted upon (program execution), and reasoned upon (validation). The choice of adequate programming languages has a tremendous impact on software quality. By "adequate", we mean in particular the following four aspects of programming languages:

- **Safety.** The programming language must not expose error-prone low-level operations (explicit memory deallocation, unchecked array accesses, etc) to the programmers. Further, it should provide constructs for describing data structures, inserting assertions, and expressing invariants within programs. The consistency of these declarations and assertions should be verified through compile-time verification (e.g. static type checking) and run-time checks.
- **Expressiveness.** A programming language should manipulate as directly as possible the concepts and entities of the application domain. In particular, complex, manual encodings of domain notions into programmatic notations should be avoided as much as possible. A typical example of a language feature that increases expressiveness is pattern matching for examination of structured data (as in symbolic programming) and of semi-structured data (as in XML processing). Carried to the extreme, the search for expressiveness leads to domain-specific languages, customized for a specific application area.
- **Modularity and compositionality.** The complexity of large software systems makes it impossible to design and develop them as one, monolithic program. Software decomposition (into semi-independent components) and software composition (of existing or independently-developed components) are therefore crucial. Again, this modular approach can be applied to any programming language, given sufficient fortitude by the programmers, but is much facilitated by adequate linguistic support. In particular, reflecting notions of modularity and software components in the programming language enables compile-time checking of correctness conditions such as type correctness at component boundaries.
- **Formal semantics.** A programming language should fully and formally specify the behaviours of programs using mathematical semantics, as opposed to informal, natural-language specifications. Such a formal semantics is required in order to apply formal methods (program proof, model checking) to programs.

Our research work in language design and implementation centers around the statically-typed functional programming paradigm, which scores high on safety, expressiveness and formal semantics, complemented with full imperative features and objects for additional expressiveness, and modules and classes for compositionality. The OCaml language and system embodies many of our earlier results in this area [46]. Through collaborations, we also gained experience with several domain-specific languages based on a functional core, including XML processing (XDuce, CDuce), reactive functional programming, distributed programming (JoCaml), and hardware modeling (ReFLect).

3.2. Type systems

Type systems [49] are a very effective way to improve programming language reliability. By grouping the data manipulated by the program into classes called types, and ensuring that operations are never applied to types over which they are not defined (e.g. accessing an integer as if it were an array, or calling a string as if it were a function), a tremendous number of programming errors can be detected and avoided, ranging from the trivial (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 [44], [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 [53], integrated in OCaml, extended the benefits of this approach (reusable code with no loss of typing precision) to object-oriented programming, extensible records and extensible variants. Another example is the work by Pottier on subtype polymorphism, using a constraint-based formulation of the type system [50].

3.2.3. Type inference.

Another crucial issue in type systems research is the issue of type inference: how many type annotations must be provided by the programmer, and how many can be inferred (reconstructed) automatically by the typechecker? Too many annotations make the language more verbose and bother the programmer with unnecessary details. Too few annotations make type checking undecidable, possibly requiring heuristics, which is unsatisfactory. OCaml requires explicit type information at data type declarations and at component interfaces, but infers all other types.

In order to be predictable, a type inference algorithm must be complete. That is, it must not find *one*, but *all* ways of filling in the missing type annotations to form an explicitly typed program. This task is made easier when all possible solutions to a type inference problem are *instances* of a single, *principal* solution.

Maybe surprisingly, the strong requirements – such as the existence of principal types – that are imposed on type systems by the desire to perform type inference sometimes lead to better designs. An illustration of this is row variables. The development of row variables was prompted by type inference for operations on records. Indeed, previous approaches were based on subtyping and did not easily support type inference. Row variables have proved simpler than structural subtyping and more adequate for typechecking record update, record extension, and objects.

Type inference encourages abstraction and code reuse. A programmer's understanding of his own program is often initially limited to a particular context, where types are more specific than strictly required. Type inference can reveal the additional generality, which allows making the code more abstract and thus more reuseable.

3.3. Compilation

Compilation is the automatic translation of high-level programming languages, understandable by humans, to lower-level languages, often executable directly by hardware. It is an essential step in the efficient execution, and therefore in the adoption, of high-level languages. Compilation is at the interface between programming languages and computer architecture, and because of this position has had considerable influence on the designs of both. Compilers have also attracted considerable research interest as the oldest instance of symbolic processing on computers.

Compilation has been the topic of much research work in the last 40 years, focusing mostly on high-performance execution ("optimization") of low-level languages such as Fortran and C. Two major results came out of these efforts: one is a superb body of performance optimization algorithms, techniques and methodologies; the other is the whole field of static program analysis, which now serves not only to increase performance but also to increase reliability, through automatic detection of bugs and establishment of safety properties. The work on compilation carried out in the Gallium group focuses on a less investigated topic: compiler certification.

3.3.1. Formal verification of compiler correctness.

While the algorithmic aspects of compilation (termination and complexity) have been well studied, its semantic correctness – the fact that the compiler preserves the meaning of programs – is generally taken for granted. In other terms, the correctness of compilers is generally established only through testing. This is adequate for compiling low-assurance software, themselves validated only by testing: what is tested is the executable code produced by the compiler, therefore compiler bugs are detected along with application bugs. This is not adequate for high-assurance, critical software which must be validated using formal methods: what is formally verified is the source code of the application; bugs in the compiler used to turn the source into the final executable can invalidate the guarantees so painfully obtained by formal verification of the source.

To establish strong guarantees that the compiler can be trusted not to change the behavior of the program, it is necessary to apply formal methods to the compiler itself. Several approaches in this direction have been investigated, including translation validation, proof-carrying code, and type-preserving compilation. The approach that we currently investigate, called *compiler verification*, applies program proof techniques to the compiler itself, seen as a program in particular, and use a theorem prover (the Coq system) to prove that the generated code is observationally equivalent to the source code. Besides its potential impact on the critical software industry, this line of work is also scientifically fertile: it improves our semantic understanding of compiler intermediate languages, static analyses and code transformations.

3.4. Interface with formal methods

Formal methods refer collectively to the mathematical specification of software or hardware systems and to the verification of these systems against these specifications using computer assistance: model checkers, theorem provers, program analyzers, etc. Despite their costs, formal methods are gaining acceptance in the critical software industry, as they are the only way to reach the required levels of software assurance.

In contrast with several other INRIA projects, our research objectives are not fully centered around formal methods. However, our research intersects formal methods in the following two areas, mostly related to program proofs using proof assistants and theorem provers.

3.4.1. Software-proof codesign

The current industrial practice is to write programs first, then formally verify them later, often at huge costs. In contrast, we advocate a codesign approach where the program and its proof of correctness are developed in interaction, and are interested in developing ways and means to facilitate this approach. One possibility that we currently investigate is to extend functional programming languages such as Caml with the ability to state logical invariants over data structures and pre- and post-conditions over functions, and interface with automatic or interactive provers to verify that these specifications are satisfied. Another approach that we practice is to start with a proof assistant such as Coq and improve its capabilities for programming directly within Coq. Finally, we also participated in the Focal project, which designed and implemented an environment for combined programming and proving [52].

3.4.2. Mechanized specifications and proofs for programming languages components

We emphasize mathematical specifications and proofs of correctness for key language components such as semantics, type systems, type inference algorithms, compilers and static analyzers. These components are getting so large that machine assistance becomes necessary to conduct these mathematical investigations. We have already mentioned using proof assistants to verify compiler correctness. We are also interested in using them to specify and reason about semantics and type systems. These efforts are part of a more general research topic that is gaining importance: the formal verification of the tools that participate in the construction and certification of high-assurance software.

4. Application Domains

4.1. High-assurance software

A large part of our work on programming languages and tools focuses on improving the reliability of software. Functional programming and static type-checking contribute significantly to this goal.

Because of its proximity with mathematical specifications, pure functional programming is well suited to program proof. Moreover, functional programming languages such as Caml are eminently suitable to develop the code generators and verification tools that participate in the construction and qualification of high-assurance software. Examples include Esterel Technologies's KCG 6 code generator, the Astrée static analyzer, the Caduceus/Jessie program prover, and the Frama-C platform. Our own work on compiler verification combines these two aspects of functional programming: writing a compiler in a pure functional language and mechanically proving its correctness.

Static typing detects programming errors early, prevents a number of common sources of program crashes (null references, out-of bound array accesses, etc), and helps tremendously to enforce the integrity of data structures. Judicious uses of 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 [47] and enforcement of data confidentiality through type-based inference of information flows and noninterference properties [51].

4.3. Processing of complex structured data

Like most functional languages, Caml is very well suited to expressing processing and transformations of complex, structured data. It provides concise, high-level declarations for data structures; a very expressive pattern-matching mechanism to 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 [42]. Therefore, Caml is an excellent match for applications involving significant amounts of symbolic processing: compilers, program analyzers and theorem provers, but also (and less obviously) distributed collaborative applications, advanced Web applications, financial modeling tools, etc.

4.4. Rapid development

Static typing is often criticized as being verbose (due to the additional type declarations required) and inflexible (due to, for instance, class hierarchies that must be fixed in advance). Its combination with type inference, as in the Caml language, substantially diminishes the importance of these problems: type inference allows programs to be initially written with few or no type declarations; moreover, the OCaml approach to object-oriented programming completely separates the class inheritance hierarchy from the type compatibility relation. Therefore, the Caml language is highly suitable for fast prototyping and the gradual evolution of software prototypes into final applications, as advocated by the popular "extreme programming" methodology.

4.5. Teaching programming

Our work on the Caml language has an impact on the teaching of programming. Caml Light is one of the programming languages selected by the French Ministry of Education for teaching Computer Science in *classes préparatoires scientifiques*. OCaml is also widely used for teaching advanced programming in engineering schools, colleges and universities in France, USA, and Japan.

5. Software

5.1. OCaml

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

OCaml, formerly known as Objective Caml, is our flagship implementation of the Caml language. From a language standpoint, it extends the core Caml language with a fully-fledged object and class layer, as well as a powerful module system, all joined together by a sound, polymorphic type system featuring type inference. The OCaml system is an industrial-strength implementation of this language, featuring a high-performance native-code compiler for several processor architectures (IA32, AMD64, PowerPC, ARM, etc) as well as a bytecode compiler and interactive loop for quick development and portability. The OCaml distribution includes a standard library and a number of programming tools: replay debugger, lexer and parser generators, documentation generator, compilation manager, and the Camlp4 source pre-processor.

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

5.2. CompCert C

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

The CompCert C verified compiler is a compiler for a large subset of the C programming language that generates code for the PowerPC, ARM and x86 processors. The distinguishing feature of Compcert is that it has been formally verified using the Coq proof assistant: the generated assembly code is formally guaranteed to behave as prescribed by the semantics of the source C code. The subset of C supported is quite large, including all C types except `long long` and `long double`, all C operators, almost all control structures (the only exception is unstructured `switch`), and the full power of functions (including function pointers and recursive functions but not variadic functions). The generated PowerPC code runs 2–3 times faster than that generated by GCC without optimizations, and only 7% (resp. 12%) slower than GCC at optimization level 1 (resp. 2).

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

5.3. Zenon

Participant: Damien Doligez.

Zenon is an automatic theorem prover based on the tableaux method. Given a first-order statement as input, it outputs a fully formal proof in the form of a Coq proof script. It has special rules for efficient handling of equality and arbitrary transitive relations. Although still in the prototype stage, it already gives satisfying results on standard automatic-proving benchmarks.

Zenon is designed to be easy to interface with front-end tools (for example integration in an interactive proof assistant), and also to be easily retargetted to output scripts for different frameworks (for example, Isabelle).

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

5.4. Menhir

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

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

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

6. New Results

6.1. Formal verification of compilers and static analyses

6.1.1. The Compcert verified compiler for the C language

Participants: Xavier Leroy, Sandrine Blazy [project-team Celtique], Alexandre Pilkiewicz.

In the context of our work on compiler verification (see section 3.3.1), since 2005 we have been developing and formally verifying a moderately-optimizing compiler for a large subset of the C programming language, generating assembly code for the PowerPC, ARM, and x86 architectures [5]. This compiler comprises a back-end part, translating the Cminor intermediate language to assembly and reusable for source languages other than C [4], and a front-end translating the CompCert C subset of C to Cminor. The compiler is mostly written within the specification language of the Coq proof assistant, from which Coq's extraction facility generates executable Caml code. The compiler comes with a 50000-line, machine-checked Coq proof of semantic preservation establishing that the generated assembly code executes exactly as prescribed by the semantics of the source C program.

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

- The formal semantics for the CompCert C source language was made executable and turned into a reference interpreter. This interpreter is proved sound and complete with respect to the formal semantics. It makes it possible to animate the semantics on test programs, identifying undefined behaviors and enumerating all possible execution orders. Another application is to provide an experimental validation of the semantics itself.
- The top-level statements of compiler correctness were strengthened. In particular, semantic preservation is shown to hold even in the presence of a non-deterministic execution context. Also, we showed that if the source program goes wrong after performing some input/output actions, the compiled code performs at least these actions before continuing with an arbitrary behavior.
- A new optimization pass, redundant reload optimization, was added, improving performance by up to 10% on the x86 architecture.
- A general annotation mechanism was added to observe the values of local program variables at user-specified program points, such observations being guaranteed to produce the same results in the source code and in the compiled code. These annotations can be used to improve the precision of worst-case execution time (WCET) analysis over the compiled code. They can also provide stronger evidence of traceability for code qualification purposes.

Three versions of the CompCert development were publically released, integrating these improvements: versions 1.8.1 in March, 1.8.2 in April, and 1.9 in August.

In parallel, we continued our collaboration with Jean Souyris, Ricardo Bedin França and Denis Favre-Felix at Airbus. They are conducting an experimental evaluation of CompCert's usability for avionics software, and studying the regulatory issues (DO-178 certification) surrounding the potential use of CompCert in this context. Preliminary results were reported at the Predictability and Performance in Embedded Systems workshop [20]. More detailed results will be presented at the 2012 Embedded Real-Time Software and Systems conference (ERTS'12) [19].

6.1.2. *Formal specification and verified compilation of C++*

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

This year, under Xavier Leroy's supervision and with precious C++ advice from Gabriel Dos Reis, Tahina Ramananandro tackled the issue of formally specifying object construction and destruction in multiple-inheritance languages, especially the C++ flavour featuring non-virtual and virtual inheritance (allowing repeated and shared base class subobjects), and also structure array fields. This formalization consists in specifying, in Coq, a small-step operational semantics for a subset of C++ featuring multiple inheritance, static and dynamic casts, field accesses, and object construction and destruction, and mechanically proving properties about resource management, thus obtaining a formal account of the RAII (*Resource Acquisition is Initialization*) principle. Moreover, this formalization also studies the impact of object construction and destruction on the behaviour of dynamic operations such as virtual function dispatch, introducing the notion of generalized dynamic type. These results were accepted for publication at the POPL 2012 symposium [29].

Finally, this formalization includes a verified realistic compiler for this subset of C++ to a CFG-style 3-address intermediate language featuring low-level memory accesses in the style of the CompCert RTL language. Following usual compilation schemes and techniques inspired from the Common Vendor ABI for Itanium (which has since been reused and adapted by GNU GCC), the target language additionally features virtual tables to model object-oriented features, and virtual table tables to model the generalized dynamic type changes during object construction and destruction. This verified compiler reuses and extends the results of a previous work on verified C++ object layout by Tahina Ramananandro, Gabriel Dos Reis and Xavier Leroy published this year at the POPL 2011 symposium [28].

6.1.3. *Validation of polyhedral optimizations*

Participants: Alexandre Pilkiewicz, François Pottier.

The polyhedral representation of loop nests with affine bounds is a unified way to compute and represent a large set of optimizations, including loop fusion, skewing, splitting, peeling, tilling etc. Polyhedral optimizers usually rely on heavily optimized C tools and libraries to manipulate polyhedrons. Those C libraries are, like any other programs, bug prone, which can easily lead to erroneous optimizations.

Those two facts—powerful yet error prone—make the formal proof of such optimizations appealing. Proving a full optimizer however would probably be unrealistic: the proof would be terribly challenging, but even writing in Coq an optimizer efficient enough to handle non trivial loop nest might be impossible.

Another option is to write and prove in Coq a validator: after each run of the unproved optimizer—considered as a black box—the validator is used to compare the program before and after optimization to make sure that its semantics—the meaning of the program—has not been change. If the validator does not report an error, we have formal certitude that no bug has been introduced by the optimization.

Alexandre Pilkiewicz, under François Pottier's supervision, has implemented and proved in Coq such a validator.

6.1.4. A formally-verified parser for CompCert

Participants: Jacques-Henri Jourdan, François Pottier, Xavier Leroy.

During a 6-month Master's internship (M2), Jacques-Henri Jourdan built a formally-verified parser for the C99 language. This parser was obtained through a general method for checking that an LR(1) parser produced by the parser generator Menhir is correct and complete, that is, it conforms exactly to the specification represented by the context-free grammar. This check is carried out by a validator that is implemented in Coq and proved correct, so that, in the end, there is no need to trust Menhir. A paper describing this work was accepted for presentation at the ESOP 2012 conference [24].

6.1.5. Formal verification of an alias analysis

Participants: Valentin Robert, Xavier Leroy.

As part of his 5-month Master's internship, Valentin Robert developed and proved correct a static analysis for pointers and non-aliasing. This alias analysis is intraprocedural and flow-sensitive, and follows the "points-to" approach of Andersen [40]. An originality of this alias analysis is that it is conducted over the RTL intermediate language of the CompCert compiler: since RTL is essentially untyped, the traditional approaches to field sensitivity do not apply, and are replaced by a simple but effective tracking of the numerical offsets of pointers with respect to their base memory blocks. Using the Coq proof assistant and techniques inspired from abstract interpretation, Valentin Robert proved the soundness of his alias analysis against the operational semantics of RTL.

6.2. Type systems

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

Participants: François Pottier, Jan Schwinghammer [Saarland University, Saarbrücken], Lars Birkedal [IT University of Copenhagen], Bernhard Reus [University of Sussex, Brighton], Kristian Støvring [University of Copenhagen], Hongseok Yang [University of Oxford].

During the year 2010, François Pottier developed a machine-checked proof of an expressive type-and-capability system. Such a system can be used to type-check and prove properties of imperative ML programs. The proof, which follows a "syntactic" method, is carried out in Coq and takes up roughly 20,000 lines of code. It confirms that earlier publications by Charguéraud and Pottier [1], [7] were indeed correct, offers insights into the design of the type-and-capability system, and provides a firm foundation for further research. In the first half of 2011, François Pottier wrote a paper that describes the system and its proof in detail. This paper has been submitted for publication [37].

Together with Jan Schwinghammer and other co-authors, François Pottier also worked on a (pencil-and-paper) proof of this type-and-capability system. This proof is based on a "semantic" method and is quite different from the proof mentioned in the previous paragraph. It offers somewhat different insights, and proves (for the first time) that the ideas presented in an unpublished note by Pottier ("Generalizing the higher-order frame and anti-frame rules", 2009) were correct. A paper that describes this proof has been submitted for publication [39].

6.2.2. Fine-grained static control of side effects in HaMLet

Participants: Jonathan Protzenko, François Pottier.

In the past ten years, the type systems community and the separation logic community, among others, have developed highly expressive formalisms for describing ownership policies and controlling side effects in imperative programming languages. In spite of this extensive knowledge, it remains very difficult to come up with a programming language design that is simple, effective (it actually controls side effects!) and expressive (it does not force programmers to alter the design of their data structures and algorithms). Jonathan Protzenko and François Pottier have recently made significant progress on this topic. They are designing a programming language, tentatively called HaMLet, in the tradition of ML and Caml-Light. The language offers immutable and mutable algebraic data structures and first-class functions. It allows very fine-grained control of ownership and side effects. The project is still at a preliminary stage and no publications have appeared yet.

6.2.3. Partial type inference with first-class polymorphism

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

The language MLF uses optional type annotations of function parameters and instance bounded polymorphism—quantification over all types that are instances of a given type—to smoothly combine the simple type inference mechanism of ML with the expressive types of System F. In MLF, programs need only type annotations on parameters of functions that are used polymorphically in their body.

While the surface language requires just these very few type annotations, MLF also comes with an internal language, called *x*MLF, where all type manipulations become explicit so that they it can traced during program transformations and symbolic evaluation. The internal language is described in a journal paper [13].

Gabriel Scherer has maintained and improved a prototype implementation of MLF including the elaboration of MLF into *x*MLF and an extension to higher-order types.

6.2.4. First-class module systems

Participants: Benoît Montagu [University of Pennsylvania], Didier Rémy, Gabriel Scherer.

Singleton kinds are used to handle type definitions in modules. They accurately model the propagation of type definitions through higher-order functor applications. However, type equivalence in the presence of singleton types is hard to formalize and to implement. In his PhD dissertation [48], Benoît Montagu has proposed a new way of checking equivalence in the presence of singleton types, based on expansors. Expansors are eta-expansion constants that are inserted in the source program in such a way that equivalence of two programs becomes equality of their normal forms after insertion of expansors. This approach was described in an article to be submitted to a conference.

Since October, Gabriel Scherer has been working on mixin modules. Mixin modules are an attractive generalization of modules with horizontal composition, a mechanism that allows more flexible construction of modules.

Gabriel Scherer has been studying whether the use of open existential types introduced earlier by Benoît Montagu for first-class modules can be used to simplify the presentation of mixin modules, hoping that they could be given a direct semantics, instead of one by means of elaboration into another language with recursive modules.

6.2.5. Coercion abstraction

Participants: Julien Cretin, Didier Rémy.

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

We studied F^p_ι, a language where all type transformations are represented as coercions. This language provides polymorphism as in System F, (upper) bounded polymorphism as in $F_{<:}$, lower bounded polymorphism as in MLF, and η-expansion as in F_η. Hence, F^p_ι unifies these four languages in a generic framework.

We showed that F^p_ι has a type erasing semantics by bisimulation with the lambda calculus. This means that coercions can be dropped before evaluation without changing the meaning of programs.

This work is described in a paper to be presented at the POPL 2012 conference [21] and in a technical report [33].

6.2.6. Kind-level typing in Haskell

Participants: Julien Cretin, Brent Yorgey [University of Pennsylvania], Stephanie Weirich [University of Pennsylvania], José Pedro Magalhães [Utrecht University], Simon Peyton Jones [Microsoft Research Cambridge], Dimitrios Vytiniotis [Microsoft Research Cambridge].

Haskell is a functional programming language with a rich static type system. Programmers use advanced type features to enforce invariants over data structures. This quickly leads to the need for computation in types. Until now, computation at the type level was untyped in Haskell and therefore prone to errors and hard to debug.

We extended the kind level of Haskell with two features already present at the type level: data types and polymorphism. These features are already well-known at the type level, and should remain easy to understand for programmers at the kind level.

Kind polymorphism is now implemented and used in the core language of the

Glasgow Haskell Compiler (GHC). Promotion of data-types is implemented in a branch of GHC. Both extensions are described in a paper to be presented at the TLDI 2012 workshop [31].

6.3. Software specification and verification

6.3.1. Proved time complexity bounds for program components

Participants: Sylvain Dailler, François Pottier.

During a six-month master internship (M2), Sylvain Dailler extended Arthur Charguéraud's CFML tool with a notion of "time credit". This allows CFML to be used to prove not only that an algorithm (or a data structure, or a library) is correct, but also that it meets a desired worst-case asymptotic complexity bound. Because CFML is hosted within Coq, these proofs are machine-checked. Sylvain Dailler was able to establish the functional correctness and the time complexity of a library that implements "bags" as circular doubly-linked lists [35].

6.3.2. Hybrid contract checking via symbolic simplification

Participant: Na Xu.

Program errors are hard to detect or prove absent. Allowing programmers to write formal and precise specifications, especially in the form of contracts, is one popular approach to program verification and error discovery. Na Xu formalized and implemented a hybrid contract checker for a subset of OCaml. The key technique is the use of symbolic simplification, which makes integrating static and dynamic contract checking easy and effective. This technique statically verifies that a function satisfies its contract or blames the function violating the contract. When a contract satisfaction is undecidable, it leaves residual code for dynamic contract checking. A paper describing this result will be presented at the PEPM'2012 conference [30]. A technical report version is also available [34].

6.3.3. Tools for TLA+

Participants: Damien Doligez, Leslie Lamport [Microsoft Research], Stephan Merz [EPI VeriDis], Denis Cousineau [Microsoft Research-INRIA Joint Centre], Markus Kuppe [Microsoft Research-INRIA Joint Centre], Hernán Vanzetto [Microsoft Research-INRIA Joint Centre].

Damien Doligez is head of the "Tools for Proofs" team in the Microsoft-INRIA Joint Centre. The aim of this team is to extend the TLA+ language with a formal language for hierarchical proofs, formalizing the ideas in [45], and to build tools for writing TLA+ specifications and mechanically checking the corresponding formal proofs.

This year, the TLA+ project prepared the release of the third version of the TLA+ tools: the GUI-based TLA Toolbox and the TLA+ Proof System, an environment for writing and checking TLA+ proofs. This new release will add many improvements in terms of efficiency, notably with a system of fingerprints to support incremental development of proofs. It will also bring support for new back-ends based on SMT provers (CVC3, Z3, Yices, VeriT). This extends the range of proof obligations that the system can discharge automatically.

Web site: http://tlaplus.net/.

6.3.4. The Zenon automatic theorem prover

Participant: Damien Doligez.

Damien Doligez continued the development of Zenon, a tableau-based prover for first-order logic with equality and theory-specific extensions. This year, a refactoring of the prover's architecture was started.

6.4. The Caml language and system

6.4.1. The OCaml system

Participants: Xavier Clerc [team SED], Damien Doligez, Alain Frisch [Lexifi SAS], Jacques Garrigue [University of Nagoya], Fabrice Le Fessant [EPI Asap and OCamlPro start-up company], Jacques Le Normand [Lexifi SAS], Xavier Leroy, Nicolas Pouillard, Pierre Weis [EPI Estime].

This year, we released version 3.12.1 of the OCaml system. This is a minor release that fixes 65 reported bugs and 9 unreported bugs, and introduces 11 small extensions. Damien Doligez acted as release manager for this version.

In parallel, we have been working on the next major release of OCaml. The major innovation is support for generalized algebraic datatypes (GADTs). These non-uniform datatype definitions enable programmers to express some invariants over data structures, and the OCaml type-checker to enforce these invariants. They also support interesting ways of reflecting types into run-time values. GADTs are found in proof assistants such as Coq and in functional languages such as Agda and Haskell. Their integration in OCaml raised delicate issues of partial type inference and principality of inferred types, to which Jacques Garrigue and Jacques Le Normand provided original solutions [43].

Other features in preparation for the next major release include:

- More lightweight first-class modules. Signature annotations over first-class modules can now be omitted when they are determined by the context.
- Better reporting of type errors: shorter but more relevant context is shown; improved tracking of source code locations in modules.
- Improvements in native-code generation, for instance in the case of partial function applications.
- Improvements in the generic hashing primitive and the standard library for hash tables.

6.4.2. Customizable unmarshaling for OCaml

Participants: Pascal Cuoq [CEA LIST], Damien Doligez, Julien Signoles [CEA LIST].

In collaboration with members of the CEA LIST laboratory, Damien Doligez developed a Caml library for treating marshaled data by applying user-specified on-the-fly transformations during the unmarshaling process. This library is used in CEA's Frama-C software to support marshaling of hash-consed data. The library was presented at the ML workshop [32].

6.5. Meta-programming

Participants: Nicolas Pouillard, François Pottier.

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

Nicolas Pouillard, under the supervision of François Pottier, investigated the design of a programming interface for names and binders where the representations of these two types are kept abstract. This interface is sufficiently general to enable a large body of program transformations.

This year, the de Bruijn indices approach has been investigated more in-depth, resulting in a programming interface specialized to safe programming with de Bruijn indices and providing much more precise results than those published in 2010. This work was published at the ICFP 2011 conference [27].

6.6. Formal management of package dependencies

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

Roberto Di Cosmo's current main line of research is the study and analysis of large component-based software repositories, in particular GNU/Linux-based distributions. These distributions consists of collections of dozens of thousands of *software packages*, together with metadata, installation and configuration tools, and a variety of different production processes, involving quality assurance at several levels.

Ensuring quality of software assemblies built using these components is a challenging issue: the simple question of knowing whether a single component can or not be deployed turns out to be NP-complete, and yet industry needs to deploy components all the time.

The research currently conducted within the Mancoosi FP7 european project, coordinated by Roberto Di Cosmo, adresses some of the relevant issues, by elaborating sophisticated deployment algorithms and designing specialised installation and configuration languages targeted at enabling transactional capabilities in the tools used to maintain software assemblies built out of GNU/Linux based distributions.

The results of this project are available at http://www.mancoosi.org/ and include four publications this year: one at the CBSE conference [18], which received an ACM distinguished paper award; one at the FSE conference [23], which received an ACM distinguished artifact award; one in the Science of Computer Programming journal [12]; and one at the workshop on Logics for Component Configuration [22].

7. Contracts and Grants with Industry

7.1. The Caml Consortium

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

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

The Consortium currently has 13 member companies:

- CEA
- Citrix
- Dassault Aviation
- Dassault Systèmes
- Esterel Technologies
- Jane Street
- LexiFi
- Microsoft
- MLstate
- Mylife.com
- OCamlCore
- OCamlPro
- SimCorp

For a complete description of this structure, refer to http://caml.inria.fr/consortium/. Xavier Leroy chairs the scientific committee of the Consortium.

8. Partnerships and Cooperations

8.1. National initiatives

8.1.1. ANR U3CAT

Participant: Xavier Leroy.

The Gallium project participates in the "U3CAT" project of the *Arpège* programme of *Agence Nationale de la Recherche*. This 3.5-year action (2009-2012) is coordinated by CEA LIST and focuses on program verification tools for critical embedded C codes. We 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.

8.1.2. FNRAE Ascert

Participant: Xavier Leroy.

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

8.1.3. IRILL

Participants: Roberto Di Cosmo, Didier Rémy.

Roberto Di Cosmo has been working on the creation of the IRILL (Initiative d'Innovation et Recherche sur le Logiciel Libre), also known as FSRII (Free Software Research and Innovation Institute), 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), to industry players willing to collaborate with researchers on these issues, and to educators working on improving the CS Curricula using Free and Open Source Software.

IRILL is an INRIA joint initiative with University Paris Diderot and University Pierre et Marie Curie. It was established by an agreement formally signed on November 2nd 2010, and its activity started with the IRILL Days event in October 2010. IRILL is currently hosting three major research projects (see http://www.irill.org).

8.1.4. LaFoSec

Participant: Damien Doligez.

The LaFoSec study, commissioned by ANSSI, aims at studying the security properties of functional languages, and especially of OCaml. The study is done by a consortium led by the SafeRiver company. It has produced more than 600 pages of documents. Most of these documents will be available from the ANSSI Web site (http://ssi.gouv.fr/). The study continues with the production of a prototype of a secure XML/XSD validator following the recommendations proposed in the first part of the study.

8.2. Regional initiatives

8.2.1. Digiteo Metal

Participants: François Pottier, Nicolas Pouillard.

The Metal project (2008-2011) of the Digiteo RTRA is coordinated by François Pottier. It focuses on formal foundations and static type systems for meta-programming.

8.2.2. Digiteo Hisseo

Participant: Xavier Leroy.

The Hisseo project (2008-2011) of the Digiteo RTRA is coordinated by Pascal Cuoq at CEA LIST. It studies issues related to floating-point arithmetic in static analyzers and verified compilers.

9. Dissemination

9.1. Animation of the scientific community

9.1.1. Collective responsibilities within INRIA

Damien Doligez is a member of the CUMIR committee (Commission des Utilisateurs des Moyens Informatiques, section Recherche).

Damien Doligez is a member of the COST/GTAI committee (Comité d'Orientation Stratégique – Groupe de Travail sur les Actions Incitatives).

Xavier Leroy was scientific organizer of the INRIA Evaluation Seminar for the "Programs, Verification and Proofs" theme (Paris, March).

Xavier Leroy is a member of Bureau du Comité des Projets of INRIA Paris-Rocquencourt.

Jonathan Protzenko, along with two other Ph.D. students, organizes the Junior Seminar of INRIA Paris-Rocquencourt, where Ph.D. students communicate their research work to a general audience.

9.1.2. Collective responsibilities outside INRIA

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

Xavier Leroy was a member of the hiring committees for two professor positions, one at ENS Lyon, the other at ENSEEIHT, Toulouse.

Xavier Leroy is INRIA representative on the Comité de Direction of the MPRI Master programme, and a member of the Commission des Études of this Master.

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.

9.1.4. Program committees and steering committees

Roberto Di Cosmo chaired the OSS 2011 workshop. He participated in the program committee of OpenCert 2011 and co-organized the Mancoosi International Solver Competition.

Xavier Leroy was a member of the program committees of the Practical Aspects of Declarative Languages conference (PADL 2011) and the international conference on Certified Programs and Proofs (CPP 2011).

François Pottier was a member of the program committee for the POPL 2012 symposium. He is a member of the steering commitee for the ACM TLDI workshop.

Didier Rémy co-chairs the Caml Users and Implementors Workshop, affiliated with ICFP 2012, to be held in Copenhagen, Denmark in September 2012.

Na Xu was a member of the program committee of the IFIP working conference on Domain-Specific Languages (IFIP DSL 2011) and the ACM SIGPLAN Workshop on Programming Languages meets Program Verification (PLPV 2012).

9.1.5. Ph.D. and habilitation juries

Roberto Di Cosmo was president of the jury of Cesara Dragoi (U. Paris Diderot, december); member of the Habilitation jury of Jean-Christophe Filliâtre (U. Paris Sud, december); external reviewer of the thesis of Paulo Trezentos (Instituto Superior Tecnico, Lisbonne, july); president of the jury of Grégoire Henry (U. Paris Diderot, june); president of the jury of Natalya Guts (U. Paris Diderot, january).

Xavier Leroy chaired the Habilitation jury for Xavier Rival (ENS Paris, june). He was a member of the Habilitation jury for Jean-Christophe Filliâtre (U. Paris Sud, december).

François Pottier was an external examiner for the Ph.D. theses of Mathieu Boespflug (École Polytechnique, January 18), Romain Bardou (U. Paris Sud, October 14), and Jules Villard (ENS Cachan, February 18). He was a member of the jury for the Ph.D. defense of Séverine Maingaud (Université Paris 7, December 13).

9.1.6. Learned societies

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

9.2. Teaching

Courses taught:

Licence: “Algorithmique et programmation” (INF431), 7h30, L3, École Polytechnique, France, taught by François Pottier.

Licence: “Principe de fonctionnement des machines binaires”, 33h, L1, University Paris Diderot, France, taught by Julien Cretin.

Licence: “Logique”, 26h, L3, University Paris Diderot, France, taught by Alexandre Pilkiewicz

Licence: “Introduction à la programmation”, 40h, L3, École Polytechnique, taught by Jonathan Protzenko.

Licence: “Virtual machines”, 14h, L3, University Paris Diderot, France, taught by Tahina Ramananandro.

Licence: “Compilation”, 26h, L3, University Paris Diderot, France, taught by Tahina Ramananandro.

Master: “Linear Logic”, 12h, M2, MPRI master (U. Paris Diderot and ENS Paris and ENS Cachan and Polytechnique), France, taught by Roberto Di Cosmo.

Master: “Functional programming and type systems”, 38h, M2, MPRI master (U. Paris Diderot and ENS Paris and ENS Cachan and Polytechnique), France, taught by Xavier Leroy and Didier Rémy.

Master: “Preuve de programmes”, 21h, M2, University Paris Diderot, France, taught by Alexandre Pilkiewicz

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

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

PhD & HdR:

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

PhD in progress: Alexandre Pilkiewicz, “Validation of polyhedral optimizations”, since December 2008, supervised by François Pottier.

PhD in progress: Nicolas Pouillard, “A unifying approach to safe programming with first-order syntax with binders”, since September 2008, to be defended January 13th, 2012, supervised by François Pottier.

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

PhD in progress: Tahina Ramananandro, “Mechanized formal semantics and verified compilation for C++ objects”, since September 2008, to be defended January 10th, 2012, supervised by Xavier Leroy.

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

9.3. Participation in conferences and seminars

9.3.1. Participation in international conferences and workshops

POPL: Principles of Programming Languages (Austin, Texas, USA, January).
Xavier Leroy gave an invited talk [16]. François Pottier presented [26]. Tahina Ramananandro presented [28]. Alexandre Pilkiewicz and Didier Rémy attended.

TLDI: Types in Language Design and Implementation (Austin, Texas, USA, January).
Alexandre Pilkiewicz presented [25]. François Pottier and Didier Rémy attended.

PPES: Predictability and Performance in Embedded Systems (Grenoble, France, March).
Xavier Leroy attended.

ETAPS: European Joint Conference on Theory and Practice of Software (Saarbrucken, Germany, March).
Na Xu attended.

CGO: Code Generation and Optimization (Chamonix, France, April).
Xavier Leroy gave an invited talk [15].

HCSS: High Confidence Software and Systems (Annapolis, Maryland, USA, April).
Xavier Leroy gave an invited talk.

Microsoft Software Summit (Issy-Les-Moulineaux, France, April).
Xavier Leroy gave a talk on compiler verification. François Pottier participated in a panel discussion on type systems. Didier Rémy attended.

AIM13: Agda Implementors' Meeting XIII (Göteborg, Sweden, April).
Nicolas Pouillard attended.

CADE: Conference on Automated Deduction (Wroclaw, Poland, August).
Xavier Leroy gave an invited talk.

ICFP: International Conference on Functional Programming (Tokyo, Japan, September).
Nicolas Pouillard presented [27].

AIM14: Agda Implementors' Meeting XIV (Shonan, Japan, September).
Nicolas Pouillard attended.

IFIP DSL: IFIP Working Conference on Domain-Specific Languages (Bordeaux, France, September).
Na Xu attended.

OSSC: OpenSource Software for Scientific Computation (Beijing, China, October) Roberto Di Cosmo gave a talk entitled *Free/Open Source Software: scientific opportunities and challenges for the future*.

9.3.2. Participation in national conferences

JFLA: Journées Francophones des Langages Applicatifs (La Bresse, France, January).
François Pottier gave an invited talk.

Colloquium in honor of Gérard Berry and Jean-Jacques Lévy (Gerardmer, France, February).
Damien Doligez and Didier Rémy attended.

Rencontres de la communauté française de compilation (Dinard, France, April).
Alexandre Pilkiewicz presented his work on validated polyhedral optimizations. Xavier Leroy attended.

GDR GPL: journées du GDR Génie de la Programmation et du Logiciel (Lille, June).
Roberto Di Cosmo gave a keynote address on *Research challenges from Free Software distributions*

GDR GPL: journées du GDR Génie de la Programmation et du Logiciel (Rennes, October).
Xavier Leroy gave a talk on [24].

9.3.3. Invitations and participation in seminars

Julien Cretin presented his work on Haskell's kind level at the seminar of Microsoft Research Cambridge.

Roberto Di Cosmo was invited to talk on free software at the *Formation des inspecteurs d'académie pour la discipline ISN* (Lyon, March), at INRIA Rocquencourt (March), at the *Journées nationales de la MIAGE* (Orsay, May) and at the LINA laboratory of U. Nantes (June).

Xavier Leroy gave a distinguished lecture at Texas A&M University (January). He gave a tutorial on Caml and Coq to the DO178 standardization committee (Toulouse, France, August).

François Pottier gave a talk at the students' seminar of ENS Lyon (March).

Tahina Ramananandro visited the FLINT laboratory at Yale University (New Haven, Connecticut, USA, November 14th–20th) and gave a seminar talk on a machine-checked formalization of C++ object construction and destruction.

Na Xu visited National University of Singapore and gave a talk on hybrid contract checking.

9.4. Other dissemination activities

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 has had a major role in fostering the emergence of 21 collaborative R&D projects, for a budget over 50 MEUR, which 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 active projects).

Roberto Di Cosmo organized the “Education with and to FOSS” track of the Fossa 2011 conference (Lyon, November).

Xavier Leroy gave a popular science talk on critical embedded software at the *Demi-heure de science* seminar of INRIA Rocquencourt (november).

Alexandre Pilkiewicz gave a popular science talk on verified compilation at the junior seminar of INRIA Rocquencourt (september).

Valentin Robert gave a tutorial on functional programming in Haskell at the Open World Forum (Paris, September).

10. Bibliography

Major publications by the team in recent years

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

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

[3] D. LE BOTLAN, D. RÉMY. *Recasting MLF*, in "Information and Computation", 2009, vol. 207, n^{o} 6, p. 726–785, http://dx.doi.org/10.1016/j.ic.2008.12.006.

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

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

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

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

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

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

[10] J.-B. TRISTAN, X. LEROY. *A simple, verified validator for software pipelining*, in "Proceedings of the 37th ACM Symposium on Principles of Programming Languages (POPL'10)", ACM Press, 2010, p. 83–92, http://doi.acm.org/10.1145/1706299.1706311.

Publications of the year

Articles in International Peer-Reviewed Journal

[11] A. W. APPEL, R. DOCKINS, X. LEROY. *A list-machine benchmark for mechanized metatheory*, in "Journal of Automated Reasoning", 2011, Accepted for publication, to appear, http://dx.doi.org/10.1007/s10817-011-9226-1.

[12] R. DI COSMO, D. DI RUSCIO, P. PELLICCIONE, A. PIERANTONIO, S. ZACCHIROLI. *Supporting software evolution in component-based FOSS systems*, in "Science of Computer Programming", 2011, vol. 76, n° 12, p. 1144–1160, http://dx.doi.org/10.1016/j.scico.2010.11.001.

[13] D. RÉMY, B. YAKOBOWSKI. *A Church-Style Intermediate Language for MLF*, in "Theoretical Computer Science", 2011, Accepted for publication, to appear, http://gallium.inria.fr/~remy/mlf/Remy-Yakobowski:xmlf@tcs2011.pdf.

Articles in Non Peer-Reviewed Journal

[14] X. LEROY. *Safety first! (technical perspective)*, in "Communications of the ACM", December 2011, vol. 54, n° 12, p. 122–122, http://doi.acm.org/10.1145/2043174.2043196.

Invited Conferences

[15] X. LEROY. *Formally verifying a compiler: Why? How? How far?*, in "Proceedings of CGO 2011, The 9th International Symposium on Code Generation and Optimization", IEEE, 2011, Abstract of invited talk, http://dx.doi.org/10.1109/CGO.2011.5764668.

[16] X. LEROY. *Verified squared: does critical software deserve verified tools?*, in "Proceedings of the 38th ACM Symposium on Principles of Programming Languages (POPL'11)", ACM Press, 2011, p. 1–2, Extended abstract of invited talk, http://dx.doi.org/10.1145/1926385.1926387.

International Conferences with Proceedings

[17] P. ABATE, R. DI COSMO. *Predicting upgrade failures using dependency analysis*, in "Workshops Proceedings of the 27th International Conference on Data Engineering, ICDE 2011", IEEE Computer Society Press, 2011, p. 145–150, http://dx.doi.org/10.1109/ICDEW.2011.5767626.

[18] P. ABATE, R. DI COSMO, R. TREINEN, S. ZACCHIROLI. *MPM: a modular package manager*, in "Proceedings of the 14th International ACM Sigsoft Symposium on Component Based Software Engineering, CBSE 2011", ACM Press, 2011, p. 179–188, http://doi.acm.org/10.1145/2000229.2000255.

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

[20] R. BEDIN FRANÇA, D. FAVRE-FELIX, X. LEROY, M. PANTEL, J. SOUYRIS. *Towards Formally Verified Optimizing Compilation in Flight Control Software*, in "Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES 2011)", OpenAccess Series in Informatics (OASIcs), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2011, vol. 18, p. 59–68, http://dx.doi.org/10.4230/OASIcs.PPES.2011.59.

[21] J. CRETIN, D. RÉMY. *On the Power of Coercion Abstraction*, in "Proceedings of the 39th ACM Symposium on Principles of Programming Languages (POPL'12)", ACM Press, 2012, To appear, http://gallium.inria.fr/~jcretin/papers/conf.pdf.

[22] R. DI COSMO, O. LHOMME, C. MICHEL. *Aligning component upgrades*, in "Proceedings Second Workshop on Logics for Component Configuration", Electronic Proceedings in Theoretical Computer Science, 2011, vol. 65, p. 1–11, http://dx.doi.org/10.4204/EPTCS.65.1.

[23] R. DI COSMO, J. VOUILLON. *On software component co-installability*, in "19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13rd European Software Engineering Conference (ESEC-13)", ACM Press, 2011, p. 256–266, http://doi.acm.org/10.1145/2025113.2025149.

[24] J.-H. JOURDAN, F. POTTIER, X. LEROY. *Validating LR(1) Parsers*, in "European Symposium on Programming (ESOP'12)", Springer, 2012, To appear, http://gallium.inria.fr/~xleroy/publi/validated-parser.pdf.

[25] A. PILKIEWICZ, F. POTTIER. *The essence of monotonic state*, in "6th Workshop on Types in Language Design and Implementation (TLDI 2011)", ACM Press, 2011, p. 73–86, http://dx.doi.org/10.1145/1929553.1929565.

[26] F. POTTIER. *A typed store-passing translation for general references*, in "Proceedings of the 38th ACM Symposium on Principles of Programming Languages (POPL'11)", ACM Press, 2011, p. 147–158, http://dx.doi.org/10.1145/1926385.1926403.

[27] N. POUILLARD. *Nameless, Painless*, in "Proceedings of the 16th International Conference on Functional Programming (ICFP 2011)", ACM Press, 2011, p. 320–332, http://doi.acm.org/10.1145/2034773.2034817.

[28] T. RAMANANANDRO, G. DOS REIS, X. LEROY. *Formal verification of object layout for C++ multiple inheritance*, in "Proceedings of the 38th ACM Symposium on Principles of Programming Languages (POPL'11)", ACM Press, 2011, p. 67–80, http://dx.doi.org/10.1145/1926385.1926395.

[29] T. RAMANANANDRO, G. DOS REIS, X. LEROY. *A Mechanized Semantics for C++ Object Construction and Destruction, with Applications to Resource Management*, in "Proceedings of the 39th ACM Symposium on Principles of Programming Languages (POPL'12)", ACM Press, 2012, To appear, http://gallium.inria.fr/~xleroy/publi/cpp-construction.pdf.

[30] D. N. XU. *Hybrid contract checking via symbolic simplification*, in "Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM'12)", ACM Press, 2012, To appear, http://gallium.inria.fr/~naxu/research/hcc.pdf.

[31] B. YORGEY, S. WEIRICH, J. CRETIN, JOSÉ PEDRO. MAGALHÃES, S. PEYTON JONES, D. VYTINIOTIS. *Giving Haskell a Promotion*, in "The Seventh ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI'12)", ACM Press, 2012, To appear, http://gallium.inria.fr/~jcretin/papers/fc-kind-poly.pdf.

Conferences without Proceedings

[32] P. CUOQ, D. DOLIGEZ, J. SIGNOLES. *Lightweight typed customizable unmarshaling*, in "ACM SIGPLAN Workshop on ML", ACM Press, 2011.

Research Reports

[33] J. CRETIN, D. RÉMY. *On the Power of Coercion Abstraction*, INRIA, December 2011, n^{o} RR-7587, http://hal.inria.fr/inria-00582570/en/.

[34] D. N. XU. *Hybrid Contract Checking via Symbolic Simplification*, INRIA, November 2011, n^{o} RR-7794, http://hal.inria.fr/hal-00644156/en.

Other Publications

[35] S. DAILLER. *Preuve mécanisée de correction et de complexité pour un algorithme impératif*, University Paris Diderot, August 2011, Report on Master's internship.

[36] J.-H. JOURDAN. *Validation d'analyseurs syntaxiques LR(1): Vers un analyseur syntaxique certifié pour le compilateur CompCert*, ENS Paris, August 2011, Report on Master's internship.

[37] F. POTTIER. *Syntactic soundness proof of a type-and-capability system with hidden state*, July 2011, Submitted for publication, http://gallium.inria.fr/~fpottier/publis/fpottier-ssphs.pdf.

[38] V. ROBERT. *Conception, mise en oeuvre et vérification d'une analyse d'alias pour CompCert, un compilateur C formellement vérifié*, ENSEIRB, December 2011, Report on Master's internship.

[39] J. SCHWINGHAMMER, L. BIRKEDAL, F. POTTIER, B. REUS, K. STØVRING, H. YANG. *A step-indexed Kripke Model of Hidden State*, December 2011, Submitted for publication, http://gallium.inria.fr/~fpottier/publis/sikmhs.pdf.

References in notes

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

[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] 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.

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

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

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

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

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

[48] B. MONTAGU. *Programmer avec des modules de première classe dans un langage noyau pourvu de sous-typage, sortes singletons et types existentiels ouverts*, École Polytechnique, December 2010, English title: Programming with first-class modules in a core language with subtyping, singleton kinds and open existential types, http://tel.archives-ouvertes.fr/tel-00550331/.

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

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

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

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

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