INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Project-Team Gallium

Programming languages, types, compilation and proofs

Paris - Rocquencourt

THEME SYM

Activity Report

2007

Table of contents

1. Team

Head of project-team
Xavier Leroy [Senior research scientist (DR), INRIA]

Vice-head of project-team
Didier Rémy [Senior research scientist (DR), INRIA, HdR]

Administrative assistant
Nelly Maloisel [TR INRIA]

Research scientists
Damien Doligez [Research scientist (CR), INRIA]
Alain Frisch [Research scientist, Corps des Télécoms, until June 2007]
François Pottier [Senior research scientist (DR), INRIA, HdR]

Ph.D. students
Arthur Charguéraud [ENS Lyon, université Paris 7, since September 2007]
Zaynah Dargaye [Île de France region grant, université Paris 7]
Benoît Montagu [AMX grant, université Paris 7, since September 2007]
Tahina Ramananandro [ENS Paris, since September 2007]
Jean-Baptiste Tristan [INRIA grant, université Paris 7]
Yann Régis-Gianas [MENRT grant, université Paris 7, until November 2007]
Boris Yakobowski [AMN grant, université Paris 7]

Post-doctoral fellow
Keiko Nakata [CNAM, part time with CNAM/ENSIIE, since April 2007]

Visiting scientist
Andrew Tolmach [Associate professor, Portland State University, until August 2007]

Technical staff
Berke Durak [Ingénieur expert, until February 2007]
Nicolas Pouillard [Ingénieur expert]

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

Student interns
Arthur Charguéraud [ENS Lyon, March–August 2007]
Benoît Montagu [Polytechnique, March–August 2007]
Tahina Ramananandro [ENS Paris, March–August 2007]

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

3.2. Type systems

Type systems [50] 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 [46], [43], 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 [54], 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 [51].

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 [53].

3.4.2. Mechanized specifications and proofs for programming languages components

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

4. Application Domains

4.1. Software safety and security

A large part of our work on programming languages and tools focuses on improving the reliability of software. Functional programming and static type-checking contribute significantly to this goal. Because of its proximity with mathematical specifications, pure functional programming is well suited to program proof. Static typing detects programming errors early and prevents a number of popular security attacks: buffer overflows, executing network data as if it were code, etc. On the safety side, judicious uses of type abstraction and other encapsulation mechanisms allow static type checking to enforce program invariants. On the security side, the methods used in designing type systems and establishing their soundness are also applicable to the specification and automatic verification of some security policies such as non-interference for data confidentiality.

4.2. Processing of complex structured data

Like most functional languages, Caml is very well suited to expressing processing and transformations of complex, structured data. It provides concise, high-level declarations for data structures; a very expressive pattern-matching mechanism to de-structure data; and compile-time exhaustiveness tests. Languages such as CDuce and OCamlDuce extend these benefits to the handling of semi-structured XML data. Therefore, Caml is a suitable match for applications involving significant amounts of symbolic processing: compilers, program analyzers and theorem provers, but also (and less obviously) distributed collaborative applications, advanced Web applications, financial analysis tools, etc.

4.3. Fast development

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

4.4. Teaching programming

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

5. Software

5.1. Objective Caml

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

Objective Caml is our flagship implementation of the Caml language. From a language standpoint, it extends the core Caml language with a fully-fledged object and class layer, as well as a powerful module system, all joined together by a sound, polymorphic type system featuring type inference. The Objective Caml system is an industrial-strength implementation of this language, featuring a high-performance native-code compiler for 9 processor architectures (IA32, PowerPC, AMD64, Alpha, Sparc, Mips, IA64, HPPA, StrongArm), as well as a bytecode compiler and interactive loop for quick development and portability. The Objective Caml distribution includes a standard library and a number of programming tools: replay debugger, lexer and parser generators, documentation generator, and the Camlp4 source pre-processor.

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

5.2. OCamlDuce

Participant: Alain Frisch.

OCamlDuce is a modified version of OCaml that integrates most of CDuce features: XML regular expression types and patterns, XML iterators, XML Namespaces. The OCaml toplevel, byte-code and native compiler, and various tools have been adapted. OCamlDuce can use any library compiled by OCaml.

Whereas CDuce targets mostly XML-oriented applications (such as XML transformation), OCamlDuce makes it possible to develop large OCaml applications that also need XML support, with the same advantages as CDuce (syntactic support for XML and XML Namespaces, complex pattern matching over XML, efficient evaluation, strong typing guarantees).

OCamlDuce was first released in 2005. A key constraint in the design of OCamlDuce was to make it possible to follow the development of OCaml itself. In respect to this constraint, we can now report a preliminary successful conclusion: we have been able to follow the evolutions of the OCaml type checker without any problem. Upgrading OCamlDuce to a new OCaml release is a matter of minutes. This is important for the viability of the project.

Some users started to adopt OCamlDuce and reported positive feedback. A typical example of usage is the use of OCamlDuce in the development of an OCaml library to export OCaml functions as SOAP web services (SOAP being based on XML).

Web site: http://www.cduce.org/ocaml.

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. Alpha-Caml

Participant: François Pottier.

Cαml (pronounced "alpha-Caml") [52] is an OCaml code generator that turns a so-called "binding specification" into safe and efficient implementations of the fundamental operations over terms that contain bound names. A binding specification resembles an algebraic data type declaration, but also includes information about names and binding constructs: where are names bound in the data structure? what is the scope of such a binding? The automatically generated operations include substitution, computation of free names, and mechanisms to traverse and transform terms. This tool helps writers of interpreters, compilers, or other programs-that-manipulate-programs deal with α-conversion in a safe and concise style.

Web site: http://cristal.inria.fr/~fpottier/alphaCaml/.

5.5. Menhir

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

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

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

6. New Results

6.1. Type systems

6.1.1. Partial Type inference with first-class polymorphism

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

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

Didier Le Botlan and Didier Rémy have proposed a type system, called MLF, that enables type synthesis as in ML while retaining the expressiveness of System F [2]. Only type annotations on parameters of functions that are used polymorphically in their body are required. All other type annotations, including all type abstractions and type applications are inferred. Remarkably, type inference in MLF reduces to a new form of unification that amounts to performing first-order unification in the presence of second-order types.

The initial study of MLF was the topic of Didier Le Botlan's PhD dissertation [48]. Didier Le Botlan and Didier Rémy have continued their work on MLF focusing on the simplification of the formalism. There is an interesting restriction of MLF that retains most of its expressiveness while being simpler and more intuitive for which types can be interpreted as sets of System-F types and type-instantiation becomes set inclusion on the semantics. This justifies a posteriori the type-instance relation of MLF that was previously defined only by syntactic means. This work has been submitted for journal publication [32].

Boris Yakobowski, who started his Ph.D. under Didier Rémy's supervision in October 2004, is pursuing the investigation of MLF, aimed at simplifying the presentation. The use of graphs rather than terms to represent types, has permitted the elimination of most of the notational redundancies. Graph types are the superposition of a dag representation of first-order terms and a binding tree that describes *where* and *how* variables are bound. This representation is much more canonical than syntactic types. This exposed a linear-time unification algorithm on graph types, which can be decomposed into standard first-order unification on the underlying dags and a simple computation on the underlying binding trees. These results have been be presented at the workshop on *Types in Language Design and Implementation* [28].

Didier Rémy presented both of these new results in an invited talk at the ML workshop.

Graphic types can also be extended to represent type inference constraints internally. This is a stepping stone for efficient type inference for MLF. This also allows for a direct specification of type inference via generation of typing constraints and a more direct proof of type soundness by showing that typing constraints of a program entails the typing constraints of its reduct. These two new results are to be submitted for publication. Boris Yakobowski has also implemented a prototype implementation both for pedagogical purposes and for verifying the efficiency of the implementation in practice—*i.e.* that the constant overhead of the graph representation is quite small.

6.1.2. 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 case. However, they remain surprisingly hard to formalize and also often become harder to use in larger, more complex but practical examples. In fact, useful features such as recursive modules or mixins are technically challenging—and still an active topic of research.

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

Our goal is to provide a new presentation of modules that is conceptually more economical while retaining (or increasing) the expressiveness and conciseness of the actual approaches. We rely on first-class modules to avoid duplications of constructs, (a new form of) *opened* existential types to represent type abstraction, and a new form of paths in types that do not depend on values to preserve the conciseness of writing.

Preliminary investigations, presented in Benoit Montagu's master dissertation, are promising. This work has also been described in an article to be submitted for publication.

In the future, we should exploit the first-class nature of our approach to increase expressiveness and conciseness of the module sub-language and exploit the simplicity of the theoretical formalism to tackle recursive modules and mixins.

6.1.3. Exact type checking for XML transformations

Participants: Alain Frisch, Haruo Hosoya [University of Tokyo].

Type systems for programming languages are usually sound but incomplete: they reject programs that would not cause type errors at run-time. There are good reasons for this incompleteness: for most programming languages, exact (complete) typing is undecidable. However, this might not be the case for type systems for XML transformation languages, which usually rely on tree automata and regular tree languages to precisely constrain the structure.

We are interested in importing results from the theory of tree transducers into programming languages for XML. There is a strong analogy between top-down tree transducers and functional programs (top-down traversal of values through pattern matching and mutually recursive functions). There is a rich literature about tree transducers. Many of the existing formalisms enjoy a property of exact type-checking: given two regular tree languages interpreted as input and output constraints, it is possible to decide without any approximation whether a given tree transducer is sound with respect to this specification.

One of the reasons which can explain the relative lack of interest from the programming language community for tree transducer techniques is that most of the problems are EXPTIME-complete and algorithms are quite complex. We believe that a proper reformulation of the algorithms will allow us to define interesting classes of transformations that support efficient type-checking and to experiment with original implementation techniques.

We are particularly interested in the formalism of so-called macro-tree transducers, which directly capture the essence of top-down functional transformations with accumulators. We have obtained a new backward type-inference algorithm for this kind of transducers. From a deterministic bottom-up tree automaton describing the output type, this algorithm produces an alternating tree automaton that represents all the valid input trees, in polynomial time (alternating tree automata can have both conjunctive and disjunctive transitions, which makes them exponentially more succinct than normal tree automata). The type-checking problem then reduces to checking emptiness of alternating tree automata, which is an DEXPTIME-complete problem in general, but some algorithms are efficient for many common situations. In particular, we have established that a transducer that traverses the input tree a bounded number of times results in an alternating tree automata whose emptiness can be checked in polynomial time (and most transducers that appear in practice satisfy this condition).

We have also developed various optimization heuristics and have implemented an efficient emptiness check for alternating tree automata. Combined with the backward type-inference algorithm, this gives the first usable type-checking tool for realistic macro-tree transducers. We have benchmarked our tool with several XML transformations on the XHTML DTD (which is quite large). For all our examples, the type-checking takes at most a few seconds and usually a few milliseconds.

These results were presented at the DBLP 2007 conference [24].

6.2. Formal verification of compilers

6.2.1. The Compcert verified compiler for the C language

Participants: Xavier Leroy, Sandrine Blazy [ENSIIE], Laurence Rideau [project Marelle], Bernard Serpette [project CMission].

In the context of our work on compiler verification (see section 3.3.1), since 2005 we have been developing and formally verifying a moderately-optimizing compiler for a large subset of the C programming language, generating assembly code for the PowerPC architecture. This compiler comprises a back-end part, translating the Cminor intermediate language to PowerPC assembly and reusable for source languages other than C [4], and a front-end translating the Clight subset of C to Cminor [44]. 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: the semantics of the generated PowerPC code matches that of the source Clight program.

The main activity on the Compcert compiler this year was to extend the proof of semantic preservation to the case of source programs that do not terminate. The original proof was restricted to terminating source programs because the semantics given to the source and intermediate languages were formalized using natural semantics (big-step operational semantics), which cannot describe non-terminating executions. To lift this limitation, Xavier Leroy reformulated these semantics using transition semantics (small-step operational semantics) for the low-level intermediate languages (RTL, LTL, Mach) and coinductive natural semantics (see section 6.4.2) for the source language Clight and the high-level intermediate languages C#minor and Cminor. Xavier Leroy then reworked the proofs of semantic preservation for every pass of the Compcert compiler, adapting them to the new semantics and extending them to the non-terminating case. This was a major overhaul of the Compcert development, requiring one third of its 40000 lines to be rewritten, but many of these rewrites were semi-systematic in nature and proof reuse was globally higher than expected.

In addition, two parts of the Compcert development that are potentially reusable in other projects were modularized and documented in articles submitted to the *Journal of Automated Reasoning* [37], [42]. The first of these papers, by Sandrine Blazy and Xavier Leroy, describes the Coq formalization of the C-like memory model used throughout the Compcert development, and its uses to reason about program transformations that modify the memory behavior of the program. The second paper, by Laurence Rideau, Bernard Serpette and Xavier Leroy, focuses on the compilation algorithm for parallel assignments used in Compcert, and its surprisingly delicate Coq proof of correctness.

6.2.2. Verified translation validation

Participants: Jean-Baptiste Tristan, Xavier Leroy.

Certified 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, is investigating this approach in the case of *instruction scheduling* transformations. Instruction scheduling is a family of low-level optimizations that reorder the program instructions so as to exploit instruction-level parallelism and reduce overall execution time. The validation algorithm for instruction scheduling is based on symbolic execution of the original and transformed programs [49]. This year, Jean-Baptiste Tristan successfully built and proved correct a validator for trace scheduling, a scheduling optimization that moves instructions across basic block boundaries. This result is detailed in a paper that will be presented at POPL 2008 [29]. It provides evidence that verified translation validation is a viable alternative to the formal verification of program transformations.

We are now trying to extend this approach in two different directions. On one hand, we would like to show that we can apply this method on even more challenging program transformations: technically, when code modifications cross loop boundaries. On the other hand we would like to show that a validator can be reused for a family of transformations, thus reducing the cost of its formal verification. In this perspective, we are now trying to validate a family of transformations that globally eliminate partial redundancies.

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

Participants: Zaynah Dargaye, Xavier Leroy.

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

Last year, Zaynah Dargaye developed and proved correct in Coq a prototype compiler from Mini-ML to Cminor, featuring lifting of function definitions to top-level and closure conversion, as well as one optimization pass: the transformation of curried functions into n-ary functions. A paper describing this optimization and its proof of correctness was presented at JFLA 2007 [22].

This year, Zaynah Dargaye attacked the issue of interfacing the generated code with an exact (non-conservative) garbage collector. This required extensive changes to the compiler and its proofs of correctness. In particular, the generated code must be instrumented to register memory roots with the garbage collector, and the correctness proofs must be re-done to account for the fact that memory blocks unreachable from these roots can be reclaimed at any allocation point. The need for root registration led her to investigate intermediate representations where all such roots are named and can be communicated to the garbage collector via a dynamically-maintained data structure.

The first such intermediate representation is continuation-passing style (CPS). Zaynah Dargaye and Xavier Leroy developed a CPS transformation and its Coq proof of correctness, utilizing a novel proof technique based on natural semantics. This result was presented at the LPAR 2007 conference [23].

Zaynah Dargaye and Xavier Leroy then developed an alternate named intermediate representation based on monadic form and inspired by ANF. Compared with ANF, this monadic intermediate representation has the advantage to remain compatible with direct-style function invocations. Zaynah Dargaye conducted Coq proofs of correctness of the transformation that puts programs in monadic form, and of the translation from this monadic form to a variant of the Cminor target language that features abstract notions of garbage collection and memory roots.

6.2.4. *Verified garbage collection*

Participants: Tahina Ramananandro, Andrew Tolmach, Xavier Leroy.

High-level languages that automate memory management, such as ML or Java, are a good basis for developing high-assurance software systems. But to have confidence in systems built on top of garbage collection, we must have confidence in the garbage collector itself. The goal of this research is to produce a fully verified, realistic GC implementation, specifically for the a collector in written the Cminor intermediate language and linked into the existing Compcert compiler framework.

Verifying GC code is a special case of the more general problem of verifying pointer-based imperative programs. Andrew Tolmach has been exploring an approach to this problem based on a monadic encoding of imperative code within Gallina, Coq's term language (the calculus of inductive constructions). Code written in Gallina can be verified directly using the full power of Coq; it can then be extracted to an executable form. Coq's existing extraction mechanism only generates pure code, but it can be adjusted or replaced to generate imperative code instead. As an initial proof of concept, Tolmach has used this technique to verify and extract code for a number of simple imperative programs, including a very simple mark-and-sweep GC. Further work is required to determine whether this approach will scale to more realistic collectors, and to address the formal verification of the extraction process itself.

As part of his Master's internship [41] and under the supervision of Xavier Leroy and Andrew Tolmach, Tahina Ramananandro investigated the formal verification of a "mark and sweep" garbage collector. To enable a future integration of this collector in the verified compiler from Mini-ML to Cminor described above, this garbage collector was written directly in the Cminor intermediate language. Its Coq proof of correctness is challenging, involving several layers of refinement from an abstract model of garbage collection down to the very concrete memory model of Cminor. Tahina Ramananandro investigated approaches to reason about code fragments directly written in Gallina (the specification language of the Coq proof assistant), and proving that these fragments do the same memory operations as the actual implementation code with respect to the semantics of Cminor. Tahina Ramananandro also modeled a concrete heap structure within the Compcert memory model.

6.2.5. Formal verification of register allocation algorithms

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

Benoît Robillard, as part of his Master's internship and under Sandrine Blazy's supervision, studied a new algorithm for register allocation and spilling, based on integer linear programming (ILP). A greedy graph coloring algorithm that is adapted to the Compcert register allocator has been specified in Coq. The optimality of the solution computed by this algorithm (when there is one) has also been proved in Coq. A paper describing these results was accepted for presentation at JFLA 2008 [19].

In the CompCert certified compiler, the register validation is performed by untrusted Caml code, whose results are validated *a posteriori* in Coq. Benoît Robillard and Sandrine Blazy developed an interface between a commercial ILP solver and the *a posteriori* validation of CompCert. Three mathematical programs modeling register allocation were implemented in this solver and tested on large examples provided by Andrew Appel. These encouraging results were presented at the *Journées Graphes et Algorithmes* [18]. As part of the beginning of his PhD, Benoît Robillard currently works on improving these mathematical programs, based on graph reduction and also on cutting-plane algorithms.

6.3. Program specification and proof

6.3.1. Certification of purely functional programs

Participants: François Pottier, Yann Régis-Gianas.

François Pottier and Yann Régis-Gianas have developed a discipline that allows specifying and certifying strict, purely functional programs.

More specifically, they consider Core ML, a strict, purely functional programming language, equipped with higher-order functions, algebraic data structures, and polymorphism. They allow Core ML programs to be decorated with logical assertions, which serve as pre- or post-conditions, and define an algorithm that extracts proof obligations out of annotated programs. The proof obligations are then fed to an external, off-the-shelf theorem prover.

This approach is inspired by earlier work, such as ESC/Java, Caduceus, or Krakatoa. However, it has never been applied to a programming language in the ML family. A technical difficulty lies in the specification of higher-order functions, which requires higher-order logic. The treatment of algebraic data structures and polymorphism is relatively straightforward.

This work is presented in an as-yet-unpublished paper [40], and a prototype tool has been implemented by Yann Régis-Gianas. The tool has been used to specify and check OCaml's balanced binary tree implementation. While the specifications must be hand-written, as in earlier work by Filliâtre and Letouzey [45], the proof is almost entirely automatic, thanks to the automated first-order theorem prover Ergo.

6.3.2. Analysis of imperative programs

Participants: Arthur Charguéraud, François Pottier.

Arthur Charguéraud and François Pottier have developed a type system, featuring regions, capabilities, and notions of linearity, which allows fine-grained reasoning about aliasing and ownership in imperative programs with dynamic memory allocation.

The type system is closely inspired by earlier work, such as the Calculus of Capabilities, Alias Types, or Adoption and Focus. Charguéraud and Pottier's principal contribution is a type-directed translation of imperative programs into a purely functional calculus. Like the well-known monadic translation, this is a store-passing translation. It is, however, much more fine-grained, because the store is partitioned into multiple fragments, which are threaded through a computation only if they are relevant to it. Furthermore, the decomposition of the store into fragments can evolve dynamically to reflect ownership transfers.

This work is presented in an as-yet-unpublished paper [36]. Charguéraud and Pottier's long-term objective is to define a system for specifying and certifying imperative programs on top of this type system. The system would be analogous to the one developed by Pottier and Régis-Gianas (see section 6.3.1), but would support side effects.

6.3.3. Focal and Zenon

Participants: Damien Doligez, Richard Bonichon [project Protheo], David Delahaye [CNAM], Pierre Weis [project AT-Roc].

Focal — a joint effort with LIP6 (U. Paris 6) and Cedric (CNAM) — is a programming language and a set of tools for software-proof codesign. The most important feature of the language is an object-oriented module system that supports multiple inheritance, late binding, and parameterization with respect to data and objects. Within each module, the programmer writes specifications, code, and proofs, which are all treated uniformly by the module system.

Focal proofs are done in a hierarchical language invented by Leslie Lamport [47]. Each leaf of the proof tree is a lemma that must be proved before the proof is detailed enough for verification by Coq. The Focal compiler translates this proof tree into an incomplete proof script. This proof script is then completed by Zenon, the automatic prover provided by Focal. Zenon is a tableau-based prover for first-order logic with equality. It is developed by Damien Doligez with the help of David Delahaye.

Version 0.5.0 of Zenon was released in November. A paper describing Zenon was presented at the LPAR 2007 conference [20].

A complete rewrite of Zenon is in progress. It will enhance the efficiency of Zenon by using purely functional data structures and by implementing a better heuristic for finding instantiations of universal hypotheses and existential conclusions.

6.3.4. Tools for TLA+

Participants: Damien Doligez, Leslie Lamport [Microsoft Research], Stephan Merz [project Mosel], Georges Gonthier [Microsoft Research], Kaustuv Chaudhuri [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 [47], and to build tools for writing TLA+ specifications and mechanically checking the corresponding formal proofs.

We have finished the design of the proof language and started implementing the front-end processor (parser and type-checker) and updating the existing TLA+ tools to deal with the new language. Kaustuv Chaudhuri was hired in November for a 2-year post-doctoral position; he will be the main architect and implementer of the "proof manager", a development environment for developing TLA+ specification along with their proofs. The "proof manager" will use Isabelle and Zenon as back-ends provers.

6.4. Mechanization of type systems and axiomatic and operational semantics

6.4.1. Engineering formal metatheory

Participants: Arthur Charguéraud, Brian Aydemir [University of Pennsylvania], Benjamin C. Pierce [University of Pennsylvania], Randy Pollack [University of Edinburgh], Stephanie Weirich [University of Pennsylvania].

Machine-checked proofs of properties of programming languages have become a critical need, both for increased confidence in large and complex designs and as a foundation for technologies such as proof-carrying code. However, constructing these proofs remains a black art, involving many choices in the formulation of definitions and theorems that make a huge cumulative difference in the difficulty of carrying out large formal developments. The representation and manipulation of terms with variable binding is a key issue.

We propose a novel style for formalizing metatheory which combines a locally nameless representation of terms (bound variables are represented using de Bruijn indices while free variables are represented using names) and a cofinite quantification of free variable names in inductive definitions of relations on terms. Following this style obviates the need for reasoning about alpha-conversion (the fact that alpha-equivalent terms need to be identified) and about equivariance (the fact that free names can be renamed in derivations).

Although many of the underlying ingredients of our technique have been used before, their combination here yields a significant improvement over other methodologies using first-order representations, leading to developments that are faithful to informal practice, yet require no external tool support and little infrastructure within the proof assistant.

We have carried out several large developments in this style using the Coq proof assistant, proving type soundness for System-Fsub and core ML (with references, exceptions, datatypes, recursion, and patterns), as well as subject reduction for the Calculus of Constructions. Not only do these developments demonstrate the comprehensiveness of our approach; they have also been optimized for clarity and robustness, making them good templates for future extension.

This work is described in a paper to be presented at the POPL 2008 symposium [16]. This work also gave rise to a tutorial day affiliated with POPL 2008 and organized by the University of Pennsylvania Programming Language Club. The tutorial is entitled: "Using Proof Assistants for Programming Language Research, or: How to write your next POPL paper in Coq".

6.4.2. Coinductive natural semantics

Participants: Xavier Leroy, Hervé Grall [École des Mines de Nantes].

Natural semantics, also called big-step operational semantics, is a well-known formalism for describing the semantics of a wide variety of programming languages. It lends itself well to proving the correctness of compilers and other program transformations. Our ongoing work on compiler verification makes heavy use of natural semantics. A limitation of natural semantics, as commonly used, is that it can only describe the semantics of terminating programs. However, it turns out that non-termination can be captured by inference rules similar to those of natural semantics, provided that these rules are interpreted coinductively instead of the usual inductive interpretation, or in other terms provided that infinite evaluation derivations are considered in addition to the usual finite evaluation derivations.

This year, Hervé Grall and Xavier Leroy completed a study and Coq formalization of this approach on the particular case of the call-by-value λ-calculus. New results include an extension of the coinductive approach to trace semantics, as well as new equivalence results with a simple form of denotational semantics. These results are accepted for publication in a special issue of *Information and Computation* [12].

6.4.3. A separation logic for a subset of the C language

Participants: Keiko Nakata, Sandrine Blazy, Xavier Leroy.

It is highly valuable for the development of critical software to validate the correctness of implementations using formal methods. Among the most important correctness properties that the implementation should satisfy is memory safety. Separation logic is a powerful tool to reason about memory safety. The logic has a primitive construct for expressing separation properties between memory areas, facilitating reasoning about programs with pointers and aliases.

Keiko Nakata, under guidance of Sandrine Blazy and Xavier Leroy, is developing a separation logic for the Clight subset of the C language that serves as input language to the Compcert compiler, formalizing this logic within the Coq proof assistant, and proving its soundness with respect to the operational semantics of Clight.

The intended use for this logic is to prove, using the Coq assistant, correctness properties of programs written in Clight, then compiled using the Compcert compiler. Since this compiler is formally proved, using Coq as well, to preserve the observational behavior programs, the guarantees offered by the source-level proof extend to the final executable code. An issue with this approach is that Coq is a mostly interactive prover offering little proof automation. This makes program proofs using the Clight separation logic tedious. To alleviate this issue, Keiko Nakata is considering the use of fully automated decision procedures: her separation logic is designed to support the use of a first-order decision procedure (the Ergo automatic theorem prover) by choosing suitable primitive constructs for the logic.

6.4.4. A separation logic for small-step Cminor

Participants: Andrew Appel [project Moscova and Princeton University], Sandrine Blazy.

Last year, Andrew Appel and Sandrine Blazy specified in Coq a separation logic for the Cminor intermediate language of the Compcert certified compiler. In this logic, we can prove fine-grained properties about pointers and memory footprints that are not ensured by the certified compiler. In 2007, this work was adapted to follow the evolutions of the Cminor language. Additional proofs were conducted to establish the soundness of the separation logic with respect to a small-step semantics for Cminor. These results were presented at the TPHOL 2007 conference [14]. Andrew Appel and his group are currently extending this work in order to handle shared-memory concurrency, based on the ideas of concurrent separation logic.

6.5. The Objective Caml system, tools, and extensions

6.5.1. The Objective Caml system

Participants: Damien Doligez, Alain Frisch, Jacques Garrigue [University of Nagoya], Xavier Leroy, Maxence Guesdon [team SED], Luc Maranget [project Moscova], Pierre Weis [team AT-Roc].

This year, we released version 3.10.0 of the Objective Caml system. This version, distributed in may 2007, is a major release featuring in particular:

- A major overhaul of the Camlp4 preprocessor and pretty-printer, re-structured in a more modular and extensible manner. (See section 6.5.3.)
- A new development tool: the OCamlBuild compilation manager. (See section 6.5.4.)
- The introduction of virtual instance variables in classes. Like methods, these instance variables can be re-defined in sub-classes.
- The extension of the stack backtrace mechanism (printing context for uncaught exceptions) to native-code programs.
- Two new ports to the AMD/Intel 64-bit architecture running Windows 64, and to the PowerPC 64-bit architecture running MacOS X.

A bug-fix release, version 3.10.1, is in preparation and should be distributed in january 2008. Xavier Leroy and Damien Doligez acted as release manager for 3.10.0 and 3.10.1, respectively.

6.5.2. CaFL: ReFLect over OCaml

Participants: Michel Mauny, Nicolas Pouillard, Damien Doligez.

Nicolas Pouillard, in collaboration with Damien Doligez and Michel Mauny, develops a new implementation of a functional programming language called reFLect. Used at Intel Corporation to perform model-checking and verification of circuits, reFLect supports both lazy evaluation and imperative features and uses binary decision diagrams as a generalization of boolean values. The language also includes an advanced overloading system and reflective aspects. This work, supported by a contract between INRIA and Intel, started in September 2005.

The first year of the project has been devoted to produce a specification of the language and a prototype of compiler for the core language, called CaFL. Nicolas Pouillard carried on Virgile Prevosto's work from July 2006 and continued the development of CaFL compiler and language features. Michel Mauny and Nicolas Pouillard designed the CaFL overloading system and resolution mechanism, along the lines of reFLect and Nicolas Pouillard integrated it into the CaFL compiler.

Nicolas Pouillard then integrated a reflection mechanism that allows to write programs manipulating other programs. This fairly advanced feature is used by Intel to build a theorem prover on top of it and use a "proof by evaluation" technique.

Nicolas Pouillard also designed and implemented other CaFL features such as active patterns, type abbreviations, syntactic macros and improvements to the OCaml debugger.

6.5.3. The Camlp4 pre-processor

Participant: Nicolas Pouillard.

Camlp4 is a source pre-processor for Objective Caml that enables programmers to define extensions to the Caml syntax (such as syntax macros and embedded languages), redefine the Caml syntax, pretty-print Caml programs, and program recursive-descent, dynamically-extensible parsers. For instance, the syntax of OCaml streams and recursive descent parsers is defined as a Camlp4 syntax extension. Camlp4 communicates with the OCaml compilers via pre-parsed abstract syntax trees. Originally developed by Daniel de Rauglaudre, Camlp4 has been maintained since 2003 by Michel Mauny, and is now developed by Nicolas Pouillard.

Starting during his internship in 2006, Nicolas Pouillard designed and implemented a new, modular architecture for the Camlp4 subsystem, aiming at facilitating its maintenance as well as simplifying the implementation of extensions. This reimplementation was finished this year, and released as part of OCaml 3.10.0. Nicolas Pouillard continues to maintain the new Camlp4 and provides support to developers of Camlp4 syntax extensions, helping them to transition to the new version.

6.5.4. The OCamlBuild compilation manager

Participants: Nicolas Pouillard, Berke Durak, Alain Frisch.

The OCamlBuild compilation manager was added to the OCaml distribution in the 3.10.0 release. OCamlBuild automates the recompilation and dependency analysis for OCaml projects. It implements the so-called "cut-off" strategy, reducing the amount of recompilation performed. For simple projects, OCamlBuild's built-in compilation rules entirely automate the build process: no "Makefile" is needed. Variations on the built-in rules can be specified compactly through a "tag" mechanism. For more advanced projects, OCamlBuild can be extended through plug-ins written in OCaml itself.

OCamlBuild has been initially developed by Nicolas Pouillard and Berke Durak in collaboration with Alain Frisch. Nicolas Pouillard continues to maintain it.

6.5.5. Dynamic linking of native code for Objective Caml

Participant: Alain Frisch.

Modern applications are often structured around a kernel which can be dynamically extended with plugins. Since its first version, Objective Caml has come with a Dynlink module that supports this kind of decomposition. It allows Caml modules to be loaded in a running Caml program, without compromising static type-safety guarantees. The Camlp4 preprocessor, for instance, relies on this module to dynamically load syntax extensions and pretty printers.

The Dynlink module, however, has been available only for the bytecode compiler. This year, Alain Frisch extended the Dynlink mechanism to work with the native-code optimizing compiler as well. The high-level design of the Dynlink native module – which includes coherence checks between the plugin and the main application – is very close to the bytecode one, and the Caml interface is in fact strictly the same. Some care had to be taken to ensure that internals structures related to the garbage collector and other aspects of the runtime system were properly handled.

Internally, the native Dynlink module relies on the underlying operating system's support for dynamic libraries. Some operating systems were easy to deal with because their dynamic libraries support naturally the Dynlink model: plugins can freely refer to symbols defined in the main program or in previously loaded plugins. Other operating systems required more work. For instance, the Linux AMD64 port required some tweaks in the back end of the Objective Caml compiler so as to produce position-independent code.

More difficult was the work needed to make the native Dynlink module available for the three Windows ports (Microsoft, Cygwin and MinGW toolchains). The problem is that Windows DLLs cannot refer to symbols provided in the surrounding dynamic environment; instead, they must statically refer to specific import libraries so as to resolve at link-time all the external symbol references. After a first working implementation that required heavy surgery on the back end of the Objective Caml compiler, Alain Frisch decided to adopt another strategy and we wrote a tool, called FlexDLL, which emulates the classical POSIX `dlopen` API on top of Windows DLLs. FlexDLL works as a wrapper around the underlying linker that turns unresolved symbol references into data in order to postpone their resolution until run-time.

6.6. Other results on the implementation of functional programming languages

6.6.1. Lightweight concurrency for GHC

Participants: Andrew Tolmach, Peng Li [Univ. of Pennsylvania], Simon Peyton Jones [Microsoft Research Cambridge], Simon Marlow [Microsoft Research Cambridge].

Andrew Tolmach participated in a collective effort to design a set of lightweight concurrency primitives for the Glasgow Haskell Compiler (GHC). GHC supports a large set of concurrency operations; these are currently implemented in C within the runtime system, which is complex, error-prone, and difficult to maintain or extend. The goal of this work is to define a new, minimal runtime system substrate for concurrency that supports the implementation of most concurrency features within Haskell itself. Initial results of this work were described in a paper presented at the ACM Haskell Workshop 2007 [26].

6.6.2. Lazy recursive modules

Participant: Keiko Nakata.

The goal of this work is to extend the ML module system, the theoretical underpinning of the OCaml module system, with recursion by designing a suitable evaluation strategy for modules. The introduction of recursion in the module system increases its expressive power and enables it to express widely-adopted programming idioms, often found in object-oriented programming. that are useful for extensible software development. Combined with other flexible features of the module system, the potential expressive power gained by a recursive module extension can go beyond that of object systems with respect to the extensibility.

Keiko Nakata is investigating a new evaluation strategy for modules, inspired by the lazy evaluation mechanism. The lazy evaluation mechanism, already available for instance in the OCaml core language, carries two aspects. On the one hand, it allows flexible treatment of recursion, which is hard to obtain under the eager evaluation mechanism. On the other hand, it requires runtime tests to determine whether the evaluation is still suspended or has been forced, causing additional run-time costs. Keiko Nakata is exploring disciplined uses of the lazy evaluation mechanism to keep a good balance between expressiveness and efficiency. Preliminary results have been submitted for publication [39].

7. Contracts and Grants with Industry

7.1. The Caml Consortium

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

The current members of the Consortium are: Athys (a subsidiary of Dassault Systèmes), Dassault Aviation, LexiFi, Intel, Jane Street Capital, Microsoft and XenSource. For a complete description of this structure, refer to http://caml.inria.fr/consortium/. Xavier Leroy chairs the scientific committee of the Consortium.

7.2. ReFLect over Caml

We have a contract with Intel Corporation that supports Nicolas Pouillard, Michel Mauny and Damien Doligez's work on formally defining and implementing the reFLect functional programming language designed at Intel. (See section 6.5.2.) Michel Mauny is the principal investigator for this contract.

8. Other Grants and Activities

8.1. National initiatives

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

The Gallium, AT-Roc and Protheo teams participate in an *Action de Recherche Collaborative* named "Quotient" (2007–2008). Its purpose is to study and develop an intermediate between concrete data types and abstract data types in an extension of OCaml.

8.2. European initiatives

The Gallium and Gemo projects participated in an European 6th Framework Programme STREP project named "Environment for the Development and Distribution of Open Source Software" (EDOS). The main objective of this project is to develop technology and tools to support and streamline the production, customization and updating of distributions of Open Source software. This STREP was coordinated by INRIA Futurs and involved both small and medium enterprises from the Open Source community (Mandriva, Caixa Magica, Nuxeo, Nexedi) and academic research teams (U. Paris 7, U. Genève, Zurich U., Tel-Aviv U.). It ended in September 2007.

9. Dissemination

9.1. Interaction with the scientific community

9.1.1. Collective responsibilities within INRIA

In september 2007, François Pottier was appointed a member of INRIA's COST (*Conseil d'Orientation Scientifique et Technologique*).

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

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

Didier Rémy is a member of the committee for *Délégations et Détachements*.

9.1.2. Collective responsibilities outside INRIA

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

Xavier Leroy was a member of the scientific committee of the SeSUR 2007 (*Sécurité et sûreté informatique*) ANR programme.

9.1.3. Editorial boards

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

François Pottier is an associate editor for the ACM Transactions on Programming Languages and Systems.

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

9.1.4. Program committees and steering committees

Sandrine Blazy chaired the program committee for the Journées Francophones des Langages Applicatifs (JFLA 2008).

Xavier Leroy is a member of the steering committee of the ACM Workshop on ML and the ACM Workshop Commercial Users of Functional Programming.

François Pottier served on the program committee for the PLPV 2007 workshop. He is a member of the steering committees for the ICFP conference and for the ML workshop.

Andrew Tolmach served on the program committee for the ICFP 2007 conference.

9.1.5. PhD and habilitation juries

Xavier Leroy was external examiner (*rapporteur*) for the Ph.D. thesis of Yann Hodique (university of Lille, april 2007).

François Pottier was a member of the jury (as advisor) for the Ph.D. defense of Yann Régis-Gianas (university Paris 7, november 2007).

9.1.6. Learned societies

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

9.1.7. Honors and distinctions

Xavier Leroy was awarded the Michel Monpetit prize of the French Academy of Sciences.

9.2. Teaching

9.2.1. Supervision of PhDs and internships

In cooperation with Eric Soutif (CNAM), Sandrine Blazy supervises Benoît Robillard's Ph.D. and supervised his Master's internship (ENSIIE and U. Paris 6). In cooperation with Marc Shapiro (project-team Regal), Sandrine Blazy supervised the Master's internship of Sonia Ksouri (U. Paris 6 and CNAM).

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

Xavier Leroy and Andrew Tolmach co-supervised the Master's internship of Tahina Ramananandro.

François Pottier supervised Yann Régis-Gianas, who defended his Ph.D. thesis on November 29th, 2007.

François Pottier supervised Arthur Charguéraud's master's internship over a period of six months. Arthur Charguéraud started a Ph.D. in september 2007, under the supervision of François Pottier,

Didier Rémy supervised Benoît Montagu's master's internship. He currently supervises Benoît Montagu's Ph.D. since september 2007.

Didier Rémy has been supervising Boris Yakobowski's Ph.D. since september 2004.

9.2.2. Graduate courses

The Gallium project-team is involved in the *Master Parisien de Recherche en Informatique* (MPRI), a research-oriented graduate curriculum co-organized by University Paris 7, École Normale Supérieure Paris, École Normale Supérieure de Cachan and École Polytechnique.

Xavier Leroy participates in the organization of the MPRI, as INRIA representative on its board of directors and as a member of the *commission des études*.

Xavier Leroy and François Pottier taught a 24-hour lecture on functional programming languages and type systems at the MPRI, attended by 30 students.

Alain Frisch gave a 3-hour lecture on compilation techniques for XML languages and tree automata at the Master of University Paris Sud.

Xavier Leroy gave a 6-hour lecture on abstract machines and compilation of functional languages at the "École Jeune Chercheurs en Programmation" (Dinard, may 2007), a summer school attended by about 30 first-year Ph.D. students.

9.2.3. Undergraduate courses

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

Yann Régis-Gianas and Boris Yakobowski were teaching assistants at university Paris 7.

9.3. Participation in conferences and seminars

9.3.1. Participation in conferences

POPL: Principles of Programming Languages (Nice, France, january).
Zaynah Dargaye, Damien Doligez, Alain Frisch, Xavier Leroy, François Pottier, Nicolas Pouillard, Didier Rémy, Andrew Tolmach, Jean-Baptiste Tristan and Boris Yakobowski attended.

TLDI: Types in Language Design and Implementation (Nice, France, january).
Boris Yakobowski presented a paper [28]. Didier Rémy attended.

DAMP: Workshop on Declarative Aspects of Multicore Programming (Nice, France, january).
Xavier Leroy attended.

PLAN-X: Programming Language Technologies for XML (Nice, France, january).
Alain Frisch presented [25].

JFLA: Journées Francophones des Langages Applicatifs (Aix-les-Bains, France, january).
Andrew Tolmach gave an invited talk about Operating Systems in Haskell. Sandrine Blazy attended.

ESOP; European Symposium on Programming (Braga, Portugal, march).
Andrew Tolmach attended.

Mathematical Theories of Abstraction, Substitution and Naming in Computer Science (Edinburgh, United Kingdom, may).
François Pottier gave a presentation.

MEMOCODE: Formal Methods and Models for Codesign (Nice, France, may).
Xavier Leroy gave an invited talk on compiler verification.

RTA: Rewriting Techniques and Applications (Paris, France, june).
Xavier Leroy gave an invited talk on compiler verification.

TLCA: Typed Lambda Calculi and Applications (Paris, France, june).
Keiko Nakata attended.

WST: International Workshop on Termination (Paris, France, june).
Keiko Nakata gave a presentation.

IFIP Working Group 2.8 "Functional Programming" (Selfoss, Iceland, july).
Xavier Leroy gave a talk on the ongoing verification of a Mini-ML compiler. Didier Rémy gave a talk on MLF type inference.

LICS: Logic in Computer Science (Wroclaw, Poland, july).
François Pottier presented a paper [27].

C/C++ Verification Workshop (Oxford, United Kingdom, july).
Sandrine Blazy presented [17].

TPHOLs: Theorem Proving in Higher-Order Logics (Kaiserslautern, Germany, september).
Sandrine Blazy presented [14]. Xavier Leroy gave an invited talk on compiler verification.

ICFP: International Conference on Functional Programming (Freiburg, Germany, october).
Arthur Charguéraud, Zaynah Dargaye, Xavier Leroy, Benoît Montagu, Keiko Nakata, François Pottier, Nicolas Pouillard, Didier Rémy, Jean-Baptiste Tristan and Boris Yakobowski attended.

CUFP: Commercial Users of Functional Programming (Freiburg, Germany, october).
Xavier Leroy gave an invited talk on industrial uses of Caml. Boris Yakobowski attended.

WMM: Workshop on Mechanizing Metatheory (Freiburg, Germany, October).
Arthur Charguéraud, François Pottier and Boris Yakobowski attended.

PLPV: Programming Languages meet Program Verification (Freiburg, Germany, October).
Arthur Charguéraud and François Pottier attended. François Pottier participated in a panel discussion.

MLW: ACM Workshop on ML (Freiburg, Germany, october).
Didier Rémy gave an invited talk on MLF. Xavier Leroy attended.

LPAR: Logic for Programming, Artificial Intelligence and Reasoning (Yerevan, Armenia, october).
Xavier Leroy presented [23].

JGA: Journées Graphes et Algorithmes (Paris, France, november).
Sandrine Blazy and Benoît Robillard presented [18].

Colloque STIC ANR (Paris, France, november).
Xavier Leroy gave an overview talk and progress report on the Compcert project. Sandrine Blazy attended.

EffTT: Workshop on Effects and Type Theory (Tallinn, Estonia, december).
Arthur Charguéraud gave a talk on his work with François Pottier on analysis of imperative programs.

9.3.2. Invitations and participation in seminars

Sandrine Blazy gave a talk about the Compcert C front-end compiler at the seminar of the *Plan Pluri-Formation* on "Secure software" of CNAM and LIP6 (Paris, june).

Alain Frisch participated in the Dagstuhl seminar “Programming Paradigms for the Web: Web Programming and Web Services” (Dagstuhl, Germany, january-february).

Alain Frisch was invited to give a talk on Practical Typechecking for Macro Tree Transducers at a meeting of the TRALALA ACI on XML transformation languages, logic and applications (Nice, january).

Alain Frisch gave a talk on Streaming for XML transformations at the Gemo seminar (Saclay, France, april).

Xavier Leroy gave a talk on verified compilers at the IRMAR/CELAR seminar on cryptography (Rennes, march).

Michel Mauny and Nicolas Pouillard visited the Intel research laboratory in Haifa (Israel) during one week in december.

Benoît Montagu gave a talk on first class module systems at the seminar of the PPS laboratory (Paris, november).

Andrew Tolmach gave a talk about Operating systems in Haskell at the seminar of LIP6 (Paris, june).

9.4. Other dissemination activities

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

After providing computer expertise for an experimental study of a wild tit population in 2003, Damien Doligez helped write the corresponding paper [11].

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

Boris Yakobowski wrote a tutorial paper on how to use polymorphic variants to reuse code and statically guarantee more properties [30]. This paper is to be presented at Journées Francophones des Langages Applicatifs in January.

10. Bibliography

Major publications by the team in recent years

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

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

[3] X. LEROY. *A modular module system*, in "Journal of Functional Programming", vol. 10, n^{o} 3, 2000, p. 269–303, http://gallium.inria.fr/~xleroy/publi/modular-modules-jfp.ps.gz.

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

[5] F. POTTIER. *A Versatile Constraint-Based Type Inference System*, in "Nordic Journal of Computing", vol. 7, n^{o} 4, 2000, p. 312–347.

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

[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] 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.

[9] 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.

Year Publications

Articles in refereed journals and book chapters

[10] S. BLAZY. *Comment gagner confiance en C ?*, in "Technique et Science Informatiques", vol. 26, n^{o} 9, 2007, p. 1195-1200, http://www.ensiie.fr/~blazy/TSI07Blazy.pdf.

[11] B. DOLIGEZ, A. BERTHOULY, D. DOLIGEZ, M. TANNER, V. SALADIN, D. BONFILS, H. RICHNER. *Spatial scale of local breeding habitat quality and adjustment of breeding decisions in a wild tit population*, in "Ecology (ISSN 0012-9658)", Accepted for publication, 2008.

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

[13] 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, http://gallium.inria.fr/~fpottier/publis/simonet-pottier-hmg-toplas.ps.gz.

Publications in Conferences and Workshops

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

[15] A. W. APPEL, X. LEROY. *A list-machine benchmark for mechanized metatheory (extended abstract)*, in "Proc. Int. Workshop on Logical Frameworks and Meta-Languages (LFMTP'06)", Electronic Notes in Computer Science, vol. 174/5, 2007, p. 95–108, http://gallium.inria.fr/~xleroy/publi/listmachine-lfmtp.pdf.

[16] B. AYDEMIR, A. CHARGUÉRAUD, B. C. PIERCE, R. POLLACK, S. WEIRICH. *Engineering Formal Metatheory*, in "Proceedings of the 35th ACM Symposium on Principles of Programming Languages (POPL'08)", Accepted for publication, to appear, ACM Press, January 2008.

[17] S. BLAZY. *Experiments in validating formal semantics for C*, in "Proceedings of the C/C++ Verification Workshop", Technical report ICIS-R07015, Radboud University Nijmegen, 2007, p. 95–102, http://www.ensiie.fr/~blazy/C07Blazy.pdf.

[18] S. BLAZY, B. ROBILLARD, E. SOUTIF. *Coloration avec préférences dans les graphes triangulés*, in "Journées Graphes et Algorithmes (JGA'07)", nov 2007.

[19] S. BLAZY, B. ROBILLARD, E. SOUTIF. *Vérification formelle d'un algorithme d'allocation de registres par coloration de graphes*, in "Journées Francophones des Langages Applicatifs (JFLA'08), Étretat, France", Accepted for publication, to appear, January 2008, http://www.ensiie.fr/~blazy/JFLABRS08.pdf.

[20] R. BONICHON, D. DELAHAYE, D. DOLIGEZ. *Zenon: an Extensible Automated Theorem Prover Producing Checkable Proofs*, in "Logic for Programming, Artificial Intelligence and Reasoning, 14th Int. Conf. LPAR 2007", Lecture Notes in Artificial Intelligence, vol. 4790, Springer, 2007, p. 151–165, http://focal.inria.fr/zenon/zenlpar07.pdf.

[21] R. BONICHON, O. HERMANT. *On Constructive Cut Admissibility in Deduction Modulo*, in "Types for Proofs and Programs, International Workshop, TYPES 2006, Revised Selected Papers", Lecture Notes in Computer Science, vol. 4502, Springer, 2007, p. 33-47, http://dx.doi.org/10.1007/978-3-540-74464-1_3.

[22] Z. DARGAYE. *Décurryfication certifiée*, in "Journées Francophones des Langages Applicatifs (JFLA'07)", INRIA, 2007, p. 119–134, http://gallium.inria.fr/~dargaye/publications/certdec.pdf.

[23] Z. DARGAYE, X. LEROY. *Mechanized verification of CPS transformations*, in "Logic for Programming, Artificial Intelligence and Reasoning, 14th Int. Conf. LPAR 2007", Lecture Notes in Artificial Intelligence, vol. 4790, Springer, 2007, p. 211–225, http://gallium.inria.fr/~xleroy/publi/cps-dargaye-leroy.pdf.

[24] A. FRISCH, H. HOSOYA. *Towards Practical Typechecking for Macro Tree Transducers*, in "Database Programming Languages, 11th International Symposium, DBPL 2007", Lecture Notes in Computer Science, vol. 4797, Springer, 2007, p. 246–260, http://dx.doi.org/10.1007/978-3-540-75987-4_17.

[25] A. FRISCH, K. NAKANO. *Streaming XML Transformations Using Term Rewriting*, in "Workshop Programming Language Technologies for XML (PLAN-X 2007)", January 2007, p. 2–13, http://www.plan-x-2007.org/plan-x-2007.pdf.

[26] P. LI, S. MARLOW, S. PEYTON JONES, A. TOLMACH. *Lightweight concurrency primitives for GHC*, in "Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell workshop", ACM Press, 2007, p. 107–118, http://doi.acm.org/10.1145/1291201.1291217.

[27] F. POTTIER. *Static Name Control for FreshML*, in "Twenty-Second Annual IEEE Symposium on Logic In Computer Science (LICS'07)", IEEE Computer Society Press, July 2007, p. 356–365, http://gallium.inria.fr/~fpottier/publis/fpottier-pure-freshml.ps.gz.

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

[29] 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)", Accepted for publication, to appear, ACM Press, January 2008, http://gallium.inria.fr/~xleroy/publi/validation-scheduling.pdf.

[30] B. YAKOBOWSKI. *Le caractère "backquote" à la rescousse – Factorisation et réutilisation de code grâce aux variants polymorphes*, in "Journées Francophones des Langages Applicatifs (JFLA'08), Étretat, France", Accepted for publication, to appear, January 2008, http://gallium.inria.fr/~yakobows/jfla08.html.

Internal Reports

[31] A. W. APPEL, S. BLAZY. *Separation logic for small-step Cminor (extended version)*, 29 pages, Research report, n^{o} 6138, INRIA, 2007, http://hal.inria.fr/inria-00134699/.

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

[33] X. LEROY, D. DOLIGEZ, J. GARRIGUE, D. RÉMY, J. VOUILLON. *The Objective Caml system, documentation and user's manual – release 3.10*, INRIA, May 2007, http://caml.inria.fr/pub/docs/manual-ocaml/.

[34] X. LEROY. *A locally nameless solution to the POPLmark challenge*, Research report, n^{o} 6098, INRIA, January 2007, http://hal.inria.fr/inria-00123945.

Miscellaneous

[35] A. CHARGUÉRAUD. *Proof of imperative programs*, Master's dissertation (mémoire de stage de Master 2), ENS Lyon, September 2007.

[36] A. CHARGUÉRAUD, F. POTTIER. *Functional Translation of a Calculus of Capabilities*, Submitted, November 2007, http://gallium.inria.fr/~fpottier/publis/chargueraud-pottier-capabilities.pdf.

[37] X. LEROY, S. BLAZY. *Formal verification of a C-like memory model and its uses for verifying program transformations*, Accepted modulo revisions to the Journal on Automated Reasoning, October 2007, http://gallium.inria.fr/~xleroy/publi/memory-model-journal.pdf.

[38] B. MONTAGU. *Modules de première classe*, Master's dissertation (mémoire de stage de Master 2), École Polytechnique, September 2007, http://gallium.inria.fr/~montagu/publications/2007/modules.pdf.

[39] K. NAKATA. *Frozen Modules: A lazy evaluation strategy for more recursive initialization patterns*, Submitted, December 2007.

[40] F. POTTIER, Y. RÉGIS-GIANAS. *Extended Static Checking of Call-by-Value Functional Programs*, Draft, July 2007, http://gallium.inria.fr/~fpottier/publis/pottier-regis-gianas-escfp.pdf.

[41] T. RAMANANANDRO. *Vérification formelle d'une implémentation d'un gestionnaire de mémoire pour un compilateur certifié*, Master's dissertation (mémoire de stage de Master 2), ENS Paris, September 2007, http://www.eleves.ens.fr/~ramanana/travail/gc.

[42] L. RIDEAU, B. P. SERPETTE, X. LEROY. *Tilting at windmills with Coq: Formal verification of a compilation algorithm for parallel moves*, Accepted modulo revisions to the Journal on Automated Reasoning, September 2007, http://gallium.inria.fr/~xleroy/publi/parallel-move.pdf.

References in notes

[43] 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.

[44] 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.

[45] J.-C. FILLIÂTRE, P. LETOUZEY. *Functors for Proofs and Programs*, in "Programming Languages and Systems, 13th European Symposium on Programming, ESOP 2004", Lecture Notes in Computer Science, vol. 2986, Springer, 2004, p. 370-384.

[46] 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.

[47] 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.

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

[49] G. C. NECULA. *Translation validation for an optimizing compiler*, in "Programming Language Design and Implementation 2000", ACM Press, 2000, p. 83–95.

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

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

[52] F. POTTIER. *An overview of Cαml*, in "ACM Workshop on ML", Electronic Notes in Theoretical Computer Science, vol. 148(2), September 2005, p. 27–52, http://gallium.inria.fr/~fpottier/publis/fpottier-alphacaml.pdf.

[53] 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.

[54] 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.