INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Project-Team Gallium

Programming languages, types, compilation and proofs

Rocquencourt

THEME SYM

Table of contents

1. Team

Team leader
Xavier Leroy [Senior research scientist (DR), INRIA]

Team vice-leader
Didier Rémy [Senior research scientist (DR), INRIA, HdR]

Administrative assistant
Nelly Maloisel [TR INRIA]

Scientific staff
Sandrine Blazy [Assistant professor, ENSIIE]
Damien Doligez [Research scientist (CR), INRIA]
Alain Frisch [Research scientist, Corps des Télécoms]
Michel Mauny [Professor, ENSTA, on secondment from INRIA]
François Pottier [Research scientist (CR), INRIA, HdR]
Andrew Tolmach [Associate professor, Portland State University, on sabbatical since September 2006]

Technical staff
Berke Durak [Ingénieur expert]
Maxence Guesdon [IR INRIA, 20% Gallium, 80% Miriad]
Virgile Prevosto [Ingénieur expert, until April 2006]
Nicolas Pouillard [Ingénieur expert, since September 2006]

Ph.D. students
Richard Bonichon [MENRT grant, university Paris 6]
Zaynah Dargaye [IDF region grant, university Paris 7]
Jean-Baptiste Tristan [INRIA grant, university Paris 7, since September 2006]
Yann Régis-Gianas [MENRT grant, university Paris 7]
Boris Yakobowski [ENS Cachan, university Paris 7]

Student interns
Michel Blockelet [classes préparatoires, July–August 2006]
Thomas Moniot [IIE and university of Évry, March–August 2006]
Nicolas Pouillard [EPITA, January–July 2006]
Jean-Baptiste Tristan [ENS Paris, March–August 2006]

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 fortitute 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 [40]. Through collaborations, we also gained experience with several domain-specific languages based on a functional core, including XML processing (XDuce, CDuce), reactive functional programming, distributed programming (JoCaml), and hardware modeling (ReFLect).

3.2. Type systems

Type systems [43] are a very effective way to improve programming language reliability. By grouping the data manipulated by the program into classes called types, and ensuring that operations are never applied to types over which they are not defined (e.g. accessing an integer as if it were an array, or calling a string as if it were a function), a tremendous number of programming errors can be detected and avoided, ranging from the trivial (mis-spelled identifier) to the fairly subtle (violation of data structure invariants). These restrictions are also very effective at thwarting basic attacks on security vulnerabilities such as buffer overflows.

The enforcement of such typing restrictions is called type checking, and can be performed either dynamically (through run-time type tests) or statically (at compile-time, through static program analysis). We favour 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 [37], [35], 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 [47], integrated in Objective Caml, extended the benefits of this approach (reusable code with no loss of typing precision) to object-oriented programming, extensible records and extensible variants. Another example is the work by Pottier on subtype polymorphism, using a constraint-based formulation of the type system [44].

3.2.3. Type inference.

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

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

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

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

3.3. Compilation

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

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

3.3.1. Formal verification of compiler correctness.

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

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

3.3.2. Efficient compilation of high-level languages.

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

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

3.4. Interface with formal methods

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

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

3.4.1. Software-proof codesign

The current industrial practice is to write programs first, then formally verify them later, often at huge costs. In contrast, we advocate a codesign approach where the program and its proof of correctness are developed in interaction, and are interested in developing ways and means to facilitate this approach. One possibility that we currently investigate is to extend functional programming languages such as Caml with the ability to state logical invariants over data structures and pre- and post-conditions over functions, and interface with automatic or interactive provers to verify that these specifications are satisfied. Another approach that we practice is to start with a proof assistant such as Coq and improve its capabilities for programming directly within Coq. Finally, we also participated in the Focal project, which designed and implemented an environment for combined programming and proving [46].

3.4.2. Mechanized specifications and proofs for programming languages components

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

4. Application Domains

4.1. 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 destructure 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, Damien Doligez, Jacques Garrigue [Kyoto University], Maxence Guesdon, 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. CDuce

Participants: Alain Frisch, Giuseppe Castagna [ENS Paris], Véronique Benzaken [LRI, U. Paris Sud].

CDuce is a functional programming language adapted to the safe and efficient manipulation of XML documents. Its type system ensures the validity (with respect to some DTD) of documents resulting from a transformation on valid inputs. CDuce features higher-order and overloaded functions, implicit subtyping, powerful pattern matching operations based on regular expression patterns, and other general purpose constructions.

Starting from the 0.2.0 release, CDuce includes a typeful interface with Objective Caml, which allows a smooth integration of the two languages in a complex project. CDuce units can use existing Objective Caml libraries without the burden of writing any stub code, and they can themselves be used from Objective Caml.

A new version released in 2006 improves various aspects of the system: a more efficient subtyping algorithm, better error messages, more precise type-checking of records.

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

5.3. 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.

5.4. XStream

Participant: Alain Frisch.

XStream is an experimental compiler for XML transformations that produces very efficient code. In particular, it allows transformations to run in streaming, that is, to compute and produce the result while parsing the input. Benchmarks are very encouraging. Plans include using XStream as a back-end for other tools such as CDuce, OCamlDuce, XSLT, and XQuery.

XStream heavily relies on the Objective Caml platform: OCaml is used both as the implementation language for the compiler, as the target language of compilation, and as the extension sub-language for XStream programs. Moreover, parsing is done using Camlp4.

Web site: http://gallium.inria.fr/~frisch/xstream.

5.5. 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/.

5.6. Alpha-Caml

Participant: François Pottier.

Cαml (pronounced "alpha-Caml") [45] 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/.

5.7. Menhir

Participants: François Pottier, 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/.

5.8. Cameleon

Participants: Maxence Guesdon, Pierre-Yves Strub [LIX, École Polytechnique].

Cameleon is a customizable integrated development environment for Objective Caml, providing a smooth integration between the Objective Caml compilers, its documentation, standard editors, a configuration management system based on CVS, and specialized code generation tools, such as DBForge (stub generator for accessing SQL databases).

Web site: http://home.gna.org/cameleon/.

6. New Results

6.1. Type systems

6.1.1. Extending ML with (impredicative) second-order types

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

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 supports only 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 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 [39]. Didier Le Botlan and Didier Rémy have continued their work on MLF focusing on the simplification of the formalism. An interesting restriction of MLF that retains most of its expressiveness while being simpler and more intuitive allows to provide a semantics of types as sets of System-F types and so to pull back the definition of type-instantiation from set inclusion on the semantics of types. 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.

Boris Yakobowski, who started his PhD under Didier Rémy's supervision in October 2004, is pursuing the investigation of MLF, and especially of MLF types, using graphs rather than terms, so as to eliminate 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. It led to the discovery of 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 will be presented at the workshop on *Types in Language Design and Implementation* (TLDI 07) [29]. An extended version with detailed proofs is available as a technical report [33].

6.2. Formal verification of compilers

6.2.1. Verification of a compiler back-end

Participant: Xavier Leroy.

Our work on compiler verification (see section 3.3.1) started in 2004 and 2005 with the formal verification of a compiler back-end, translating the Cminor intermediate language down to PowerPC assembly code and performing a few optimizations (register allocation by graph coloring and constant propagation). This work is described in a paper presented at the POPL 2006 symposium [26]. This year, Xavier Leroy implemented and proved correct one additional optimization pass: common subexpression elimination, performed by value numbering over basic blocks. He also revised the operational semantics of all source, intermediate and target languages so that they capture a *trace* of all input/output activities of the program, and proved that this trace of I/O events is preserved by all the passes of the back-end. This addition of traces leads to a significantly stronger observational equivalence between source and machine code than in our previous work.

6.2.2. Verification of a compiler front-end for a subset of the C language

Participants: Thomas Moniot, Sandrine Blazy, Xavier Leroy.

In parallel with our work on compiler back-end, we are also conducting the development and formal verification of compiler front-ends that target the Cminor intermediate language. The first such front-end generates Cminor code from a subset of the C language called Clight, similar to that used for programming critical embedded systems. Clight features all the arithmetic types and operators of C, as well as arrays, pointers, pointer arithmetic, function pointers, and all the structured control statements of C, but excludes unstructured control (`goto`, `switch`, `longjmp`).

A first prototype of a verified front-end for Clight was developed in 2005 and described in a paper presented this year at the Formal Methods conference [15].

As part of his Master's internship and under Sandrine Blazy's supervision, Thomas Moniot re-architected the prototype Clight front-end around the use of the CIL library developed at Berkeley [41]. CIL provides an industrial-strength parser and type-checker for the C language, as well as a simplifier that eliminates or explicates many difficult features of this language. The use of CIL enables our front-end to correctly handle a much larger subset of the C language, including `struct` and `union` types. Thomas Moniot and Xavier Leroy extended and adapted the Coq proofs of semantic preservation for the C front-end. As a result of this work, the Compcert compiler is now able to compile realistic examples of C source code, of a few thousand lines, with almost no modifications.

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 functions as first-class values, arithmetic, constructed data types and shallow pattern-matching, making it an adequate target for Coq's program extraction facility.

Zaynah Dargaye developed and proved correct in Coq three translation passes: numbering of data constructors, lifting of function definitions to top-level, closure conversion and generation of Cminor code, 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 accepted for presentation at JFLA 2007 [19].

6.2.4. Certified 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 [42]. During his Master's internship, Jean-Baptiste Tristan developed and proved correct a validator adequate for instruction scheduling at the basic-block level. As part of the beginning of his PhD, he currently works on extending validation to trace scheduling, where instructions can move around conditional branches but not loops.

6.2.5. Verified garbage collection

Participants: Andrew Tolmach, Xavier Leroy, Zaynah Dargaye.

High-level languages that automate memory management, such as ML or Java, prevent a large class of dangerous bugs, and are relatively amenable to formal reasoning about programs. This makes these languages a good basis for developing high-confidence software systems, including system software and theorem provers (such as Coq). However, to trust systems built on top of garbage collection, it is necessary to trust the garbage collector itself. Although correctness of GC *algorithms* is a very old subject, correctness of actual *implementations* has not been well-studied, and indeed there are few if any fielded systems containing a formally-verified collector. The goal of this research is to fill this gap, specifically by developing a verified collector within the Compcert compiler framework.

This work is just beginning, and we are still investigating several alternative approaches. Currently, the most promising idea is to code the collector directly in the Cminor intermediate language, and use mechanized proof assistance to verify correctness of this code. The collector can then be interfaced easily with code generated from existing front ends that generate Cminor; moreover, the existing certified compilation pipeline from Cminor to machine code can be used on the collector code itself. However, so far we have little experience in proving properties of Cminor programs (as opposed to properties of the compilation system). One possible mechanism is to adopt the Caduceus C-language verification-condition generator, developed by Filliâtre and others at LRI [36], to work on Cminor programs; we are undertaking experiments with the existing Caduceus to determine its suitability for verifying a collector.

6.3. Mechanized semantics

6.3.1. A separation logic for small-step Cminor

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

Andrew Appel and Sandrine Blazy have specified in Coq a Separation Logic for the Cminor intermediate language of the Compcert verified compiler. In this logic, we can prove fine-grained properties about pointers and memory footprints that are not ensured by the certified compiler.

The Separation Logic consists of an assertion language and an axiomatic semantics. The assertion language is a shallow embedding in Coq. Some operators of this language are specific to separation logic. The axiomatic semantics relies on a small-step semantics for statements, to support reasoning about input/output, non-termination and concurrency. The small-step semantics is based on continuations. Using the Coq proof assistant, we have proved the soundness of our axiomatic semantics with respect to our small-step semantics, and also the equivalence between our small-step semantics and the big-step semantics used in the verification of the Compcert compiler.

This experiment is encouraging: our axiomatic and operational semantics for Cminor are a first bridge between, on the one hand, program proof in the style of Hoare, and on the other hand the Compcert compiler verification effort. Further work remain to handle concurrency and to study the usability of the small-step semantics within proofs of compiler correctness.

A paper describing this work was submitted for publication [31].

6.3.2. 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.

Xavier Leroy formalized a coinductive natural semantics for divergence, using the Coq proof assistant, for a tiny functional language (call-by-value λ-calculus). He proved several results: equivalences with small-step semantics, semantic preservation for compilation to an abstract machine, and a novel approach to proving soundness of type systems. These results were presented at the ESOP 2006 symposium [25]. In collaboration with Hervé Grall, he is currently extending these results to trace semantics, where the natural semantics captures not only the final outcome of evaluation but also a trace of events generated during program execution.

6.4. Software-proof codesign

6.4.1. Extending ML with logical assertions

Participants: François Pottier, Yann Régis-Gianas.

François Pottier and Yann Régis-Gianas worked on generalized algebraic data types (GADTs), a modest extension to ML's algebraic data types that enables programmers to attach type equalities to values. From a propositions-as-types perspective, this extension provides a way of encoding and enforcing data structure invariants. The typechecker then plays the role of an automatic theorem prover. Yet, this approach to proving properties of programs has limited expressiveness. In a way, encoding properties of programs into ML types is an abuse of the ML typechecker, which, indeed, was meant to establish safety properties, not to prove more advanced correctness properties.

Realizing this, François Pottier and Yann Régis-Gianas extended ML with an assertion language, enabling programmers to explicitly state the intended properties of data structures and functions in first-order logic. They designed an algorithm that generates proof obligations out of programs. These proof obligations are discharged by a theorem prover such as Simplify. This approach was previously applied by other research groups to imperative and object-oriented languages (cf. ESC/Java, Caduceus, Krakatoa) but never to a functional language. The main issues in integrating logical assertions within a functional language include: finding a language design that facilitates reasoning about higher-order functional programs; and fostering the modular development of certified software components.

Yann Régis-Gianas developed a prototype implementation in order to experiment with several design choices. Several nontrivial algorithms have been proven using this tool, including the OCaml library for balanced binary search trees.

6.4.2. Focal and Zenon

Participants: Damien Doligez, Richard Bonichon, David Delahaye [CNAM], Olivier Hermant [project Logical and U. Paris 7].

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 parameterisation 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 [38]. 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 (CNAM).

Zenon version 0.4.1 was released in April. A complete overhaul of Zenon was started shortly after this release, and is still in progress. It will enhance the efficiency of Zenon by using the inverse method to find instantiations of universal hypotheses.

Richard Bonichon, in cooperation with Damien Doligez, works on implementing "deduction modulo" within the Zenon prover. Deduction modulo adds rewriting systems over terms and propositions to the deduction process. It enables the replacement of deduction steps by computational simplifications, therefore leading to shorter, easier to find proofs. Richard Bonichon defended his PhD thesis in December [10] and published two papers with Olivier Hermant on tableaux methods and deduction modulo [17], [18].

6.4.3. Tools for TLA+

Participants: Damien Doligez, Leslie Lamport [Microsoft Research], Stephan Merz [project Mosel], Georges Gonthier [Microsoft Research].

Damien Doligez is head of the "Tools for Proofs" team in the new Microsoft-INRIA Joint Center. The aim of this team is to extend the TLA+ language with a formal language for hierarchical proofs, formalizing the ideas in [38], and to build tools for writing TLA+ specifications and mechanically checking the corresponding formal proofs.

This project was started in June. At this time, it has made internal progress on the design of the proof language and the architecture of the tools. In the long term, it aims at producing a development environment specialized for TLA+, with seamless integration of both Zenon and the Isabelle proof assistant.

6.5. Meta-programming

6.5.1. Towards α-safe meta-programming

Participant: François Pottier.

Functional programming languages such as ML are intended for building, examining, and transforming complex symbolic objects, such as *programs* and *proofs*. They are rather well suited for this task because these objects are usually represented as *abstract syntax trees*, whose structure is expressed in ML via algebraic data type declarations.

However, abstract syntax trees involve *names* that can be *bound*, a reality that algebraic data type declarations do not reflect. Manipulating such trees involves a number of operations that respect the meaning of names, such as computing the set of *free* names of a term, or substituting, *without capture*, a name (or term) for a name throughout a term. ML provides no built-in support for these operations, which must instead be hand-coded, a tedious and error-prone process.

In 2005, in order to address this issue, François Pottier designed and implemented Cαml [45], a tool that turns a so-called "binding specification" into an Objective Caml compilation unit. One asset of Cαml is simplicity: it is a small tool. However, it has an inherent limitation. Because Cαml is only a code generator, as opposed to a full-fledged programming language, there is no way of ensuring that the generated code is used in a sensible way. That is, the user can write *impure* meta-programs, which construct abstract syntax trees that unintentionally contain unbound names.

In 2006, François Pottier explored a more ambitious design, which consists in designing a full-fledged meta-programming language, where the compiler statically rules out every error that could cause a name to become unbound. This design, known as Pure FreshML, is a version of Pitts and Gabbay's FreshML, equipped with a static *proof system* that guarantees *purity*. Pure FreshML relies on a rich "binding specification" language, borrowed from Cαml, on user-provided assertions (guards, preconditions, and postconditions), and on a conservative, automatic decision procedure for a logic that allows reasoning about values and about the names that they contain.

This approach is described in an as-yet-unpublished paper [32]. A prototype implementation of the programming language and proof system is being developed.

6.6. XML transformation languages

6.6.1. Streaming for XML transformations

Participants: Alain Frisch, Keisuke Nakano [U. Tokyo].

The classical processing model for XML transformations consists in loading a textual input document in memory as a tree, processing it to produce a new tree, which is then sent to the output as a new textual document. Of course, this model does not make a good use of three different resources: input channel, processing unit, output channel. By interleaving the three phases, one can start processing the document and producing the output while still parsing the input and sometimes without having to keep it entirely in memory. The advantages of this approach are evident when dealing with huge documents and/or with slow input or output channels (e.g. a transformation node in a network).

Writing streaming transformation by hand means that the programmer has to deal explicitly with buffering. This is extremely tricky and error-prone. In particular, the same conceptual behavior can result in drastically different implementations according to the "streaming context" in which it is used. Also, deciding whether a piece of transformation can be evaluated in streaming mode or requires buffering cannot generally be done statically, because the answer can depend on dynamic properties (the input document). This adds to the complexity of hand-written code.

There have been previous proposals for transformation languages to support streaming. There are all either limited in expressivity (e.g. to simple top-down transductions), overly conservative (requiring parts of the input to stay in memory when this is not necessary), or not automatic enough (requiring programmer-provided annotations).

We have designed and implemented a language, called XStream, that supports an efficient streaming evaluation model based on the theory of term rewriting. This language is purely functional and Turing-complete, and requires no annotations from the programmer. The language allows the programmer to express any computable tree transformation, including those that cannot be efficiently implemented in streaming; a "best effort" approach is taken to obtain streaming when possible, but without any optimality guarantee (optimal streaming is undecidable). It turns out that the compiler produces an optimal streaming behavior for all the examples we have tried. In practice, transformations compiled by XStream always compare well with widespread XML transformation technologies, and they are orders of magnitude more efficient when the transformation can be evaluated in streaming.

The theory behind the language and the techniques used in its implementation are described in a paper [23] to be presented at the PLAN-X 2007 workshop.

6.6.2. *Extending Caml with XML types*

Participant: Alain Frisch.

The type system of Objective Caml extends the original Hindley-Milner type system in various directions but keeps principal type inference based on first-order unification. In parallel, various type systems for manipulating XML documents in functional languages have been proposed recently. They usually build on an interpretation of types as sets of values, which induces a natural subtyping relation. Unlike Hindley-Milner type systems, they do not feature full type inference: function argument and return types have to be given by the programmer. Instead, they rely on tree automata algorithms to compute the subtyping relation and to propagate precise types through complex operations on XML documents such as regular expression pattern matching, deep iterations, and path navigation.

In order to facilitate the manipulation of XML documents in Caml, one may want to integrate some features from these domain specific languages into Objective Caml. Alain Frisch developed a type system and a typing algorithm for a language that combines OCaml and CDuce. This system preserves nice theoretical and practical properties of ML and CDuce. It was implemented in OCamlDuce, a modified version of OCaml that embeds CDuce. OCamlDuce was implemented by merging the OCaml and CDuce code bases and adding a relatively small amount of glue code.

The theoretical foundations of OCamlDuce's type system are described in two papers [21], [22] that were presented at the ICFP 2006 conference and at the PLAN-X 2006 workshop.

6.6.3. *Exact type checking for XML transformations*

Participants: Alain Frisch, Haruo Hosoya [U. 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.

There are several challenges to address if we want to integrate such techniques into programming languages: design of new programming features and paradigms, design of type system based on tree transducers, extending algorithms to deal with additional features not found in simple tree transducers.

The long-term objective is to design programming languages for XML with exact type-checking. This implies that no well-typed program is rejected even without any type annotations; that very precise errors can be produced for ill-typed programs; and that polymorphism comes for free. Exactness of type-checking cannot be obtained for a Turing-complete language; we need to introduce explicit typing approximations (presented as type annotations) to deal with that.

One of the reasons that 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 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. Most transducers that appear in practice satisfy this condition. We also started to experiment with efficient implementations of emptiness algorithm for alternating tree automata, with the hope to produce a usable type-checking tool for macro-tree transducers. In contrast, the classical algorithm produces a (non-alternating) tree automaton whose size is always exponential even for simple transducers. This algorithm can be reinterpreted as the composition of our new algorithm and a classical (exponential) translation from alternating tree automata to tree automata.

6.7. The Objective Caml system, tools, and extensions

6.7.1. The Objective Caml system

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

This year, we released three versions of the Objective Caml system: 3.09.1, 3.09.2 and 3.09.3. These are bug-fix releases for version 3.09, which was released at the end of 2005. Damien Doligez acted as release manager for these three versions. The main novelty in these updates is a port to the recent, popular MacOS X/Intel platform. In addition, about 60 minor bugs were corrected.

6.7.2. CamlP4

Participants: Nicolas Pouillard, Michel Mauny.

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.

During his internship under Michel Mauny's supervision, Nicolas Pouillard designed and implemented a new architecture for the CamlP4 subsystem, aiming at facilitating its maintenance as well as simplifying the implementation of extensions. As a result, CamlP4 now features a simpler bootstrap mechanism, a more efficient generic lexical analyzer, more conventional pretty-printers, a better handling of source locations, and a more modular architecture. Nicolas Pouillard also added hooks for filters, allowing to insert arbitrary program transformations of OCaml parse trees before they are sent to the compiler, and redesigned the the quotation system to make it as extensible as the input language itself.

6.7.3. Automatic tools to orchestrate the compilation process

Participants: Nicolas Pouillard, Alain Frisch, Berke Durak.

The process of compiling a software project is sometimes tricky: one must usually respect some kind of dependency ordering between compilation units, and inferring the dependencies is not trivial (it can depend on some parts of the project being already built); some source files might be automatically generated (or preprocessed) from other files, sometimes by generators which are themselves part of the project; the rules defining how the project is built can themselves change frequently during development or depend on some properties of the environment. Developers also appreciate not having to recompile all the project when only a part has been changed (incremental recompilation). The standard compilation technology, based on the `make` tool, quickly shows its limits.

We have started to study and categorize the difficulties related to orchestrating the compilation process, with a special (but not exclusive) focus on Objective Caml projects, and to develop tools to experiment with new ideas, such as high-level descriptions of software projects or dynamic discovery of dependencies by analyzing how compilers interact with the file system. The final objective of this work is to produce robust tools to make it easier to develop and deploy complex OCaml software. We also expect this work to help us design guidelines for compiler makers. For instance, a light cooperation with the compiler is often enough to obtain precise dependency information, without the need of necessarily approximative and sometimes unsound external tools.

6.7.4. ReFLect over Ocaml

Participants: Virgile Prevosto, Nicolas Pouillard, Michel Mauny, Damien Doligez.

Virgile Prevosto (until May 2006), and Nicolas Pouillard (since July 2006), in collaboration with Damien Doligez and Michel Mauny, develop a new implementation of a functional programming language called ReFLect. This language is used at Intel Corporation to perform model-checking and circuit verification. ReFLect supports both lazy evaluation and imperative features, provides binary decision diagrams as a generalization of boolean values, and includes reflective aspects. This work, supported by a contract between INRIA and Intel, started in September 2005.

The main goal of the first twelve months was to provide a non-ambiguous specification of the semantics of the existing ReFLect interpreter from Intel Strategic Lab and to implement a compiler for the core language. Virgile Prevosto implemented a prototype compiler called CaFL (or "ReFLect over OCaml") that was demonstrated to Intel during the summer 2006. Nicolas Pouillard carried on Virgile's work from July 2006 and continued the development, generalizing the usage of AlphaCaml for managing the scope of identifiers and adapting CUDD, a free BDD package, to CaFL. Michel Mauny started to implement the ReFLect overloading machanism into CaFL.

6.7.5. Type-safety of unmarshaled OCaml values

Participants: Grégoire Henry [PPS, U. Paris 7], Michel Mauny, Emmanuel Chailloux [PPS, U. Paris 7 and LIP6, U. Paris 6].

Unmarshaling (reading OCaml values from disk files, for instance) is not a type-safe operation in the current OCaml implementation. Indeed, there is currently no way to provide an unmarshaling function with a polymorphic (parametric) type. Furthermore, marshaled values written to disk by OCaml do not carry type information. Generally, unmarshaling functions in staticallly typed programming languages have monomorphic types and return values containing explicit type information. Some form of dynamic type checking is then necessary, as a programming language construct, in order to recover a statically typed value from such an unmarshaled value.

Grégoire Henry and Michel Mauny devised an extension of ML-like languages where unmarshaling functions receive a representation of some type t as argument and return values of type t, or fail. This is obtained by extending the language with a predefined parameterized abstract type α `tyrepr` whose values are representations of the current instance of the type parameter α. These type representations are then passed at run-time to an algorithm able to test whether an unmarshaled value can be safely used with type t or not: a relatively simple task, except when parts of the value are polymorphic, shared or cyclic. Grégoire Henry and Michel Mauny formalized this algorithm, and proved its correctness and completeness. This result was presented at the JFLA 2006 conference [24] and a prototype implementation was released as a patch to the OCaml distribution.

This work provides safe unmarshaling of all kinds of OCaml values excepted those involving functions. The extension of this algorithm to functions is under consideration, and uses a new formalization using a presentation of type verification as collecting and solving constraints, following Pottier and Rémy's présentation of ML type inference [7].

6.7.6. The Caml development environment

Participant: Maxence Guesdon.

Maxence Guesdon took part in the development of LablGtkSourceview, the OCaml bindings with the GtkSourceview library which provides Gtk text widgets with color highligthing.

Maxence Guesdon continues his work on Cameleon, the integrated development environment for Objective Caml described in section 5.8. This year, Chamo was added to Cameleon. Chamo is a powerful source code editor based on LablGtkSourceview and offering complete customization possibilities using OCaml code, the same way as Emacs and Lisp.

Maxence Guesdon developed a graphical editor of OCamlyacc files based on Yacclib, the small library to parse and print OCamlyacc files. This editor is included in the Yacclib distribution.

6.8. Formal management of software dependencies

Participants: Berke Durak, Xavier Leroy, Jaap Boender [U. Paris 7], Roberto Di Cosmo [U. Paris 7], Fabio Mancinelli [U. Paris 7], Jérôme Vouillon [U. Paris 7].

In the context of the European project EDOS (Environment for the development and Distribution of Open Source software), we participate in an effort led by University Paris 7 to formally understand the dependencies between the software packages that typically constitute a Linux or BSD distribution. The goal is to develop efficient algorithms and tools to improve the quality of distributions, for instance by detecting package inconsistencies and potential upgrading problems.

In 2005, we developed a formal model of package dependencies and designed algorithms based on reductions to Boolean satisfiability problems. In 2006, these results were presented at the Automated Software Engineering conference, the FRCSS workshop, and the Workshop on Free Software [27], [20], [16].

This year, Berke Durak's work on the EDOS project concentrated on the design and development of a complete toolchain for downloading, storing, indexing, verifying, browsing and searching Debian package metadata.

Berke Durak tested various database backends and found severe performance limitations in RDBMS for storing the large graph-like structure of package metadata. He and Jaap Boender developed an alternative storage backend (`dosebase`).

Tools for interconverting package dependency problems with boolean satisfiability formulas were developed. Limitations in the constraint solving logic of existing package management tools (such as `APT` or `urpmi`) were found. Well-known SAT solvers (`sat-grasp`, `zchaff`) were used to assess the feasibility of the SAT approach for veryfing the integrity of package distributions with respect to installability. Since these solvers are not free software and therefore not suitable for inclusion in most Linux distributions, Berke Durak designed, implemented and benchmarked custom SAT-solving algorithms based on simple backtracking, the Davis-Putnam approach, and conflict propagation.

Distribution editors are routinely faced with the problem of fitting a large number of packages in a given number of physical media (typically DVDs or CDs) while respecting size, dependency and disk order constraints. Berke Durak developed an algorithm that solves this problem while maximizing the total utility of the selected packages. This algorithm extends the conflict-propagating SAT-solving algorithm previously mentioned. It was implemented as a standalone tool called `tart`.

Finally, Berke Durak designed a functional query language for operating on Linux package metadata using Boolean set operators and dependency operators (such as dependency closure of a set). An implementation of this query language was developed as a library and integrated in two tools, a command-line interface (`history`) and a Web interface (`console`). This library integrates the dependency solving algorithm developed by Jérôme Vouillon at PPS, and initially used in his `debcheck` and `rpmcheck` tools.

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, and Microsoft. For a complete description of this structure, refer to http://caml.inria.fr/consortium/. Michel Mauny (until september 2006) and Xavier Leroy (since september 2006) represent INRIA in the scientific committee of the Consortium.

7.2. ReFLect

We have a contract with Intel Corporation that supports Virgile Prevosto, Nicolas Pouillard, Michel Mauny and Damien Doligez's work on formally defining and implementing the ReFLect functional programming language designed at Intel. (See section 6.7.4.) 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, Logical and Protheo projects participate in an *Action Concertée Incitative "Sécurité et Informatique"* (ACI SI) named "Modulogic", coordinated by the LIP6 laboratory at university Paris 6 and also involving the Cedric laboratory at CNAM. The theme of this action is the design of a development environment for certified software, emphasizing modular development and proof. Our participation to Modulogic is centered around the Focal language (see section 6.4.2), with the long-term goals of incorporating rewriting into Focal and studying how to apply the Focal methodology to security-sensitive applications.

8.2. European initiatives

The Gallium and Gemo projects participate 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 is coordinated by INRIA Futurs and involves 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.).

The Gallium project was involved in the European Esprit working group "Applied Semantics II". The purpose of this working group, and its predecessor "Applied Semantics", is to foster communication between theoretical research in semantics and practical design and implementation of programming languages. It ended in june 2006. Didier Rémy was the INRIA coordinator for this working group.

9. Dissemination

9.1. Interaction with the scientific community

9.1.1. Learned societies

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

9.1.2. Collective responsibilities within INRIA

Didier Rémy was a member of the hiring committees for the Rocquencourt and Futur CR2 competitions. He is also a member of the committee for *Délégations et Détachements*.

9.1.3. Collective responsibilities outside INRIA

Sandrine Blazy is a member of the Commission de spécialistes (hiring committees) of CNAM.

Michel Mauny chairs the scientific board of the French-Moroccan program *Réseaux STIC*: a cooperation program managed by INRIA and funded by the French Ambassy in Rabat.

9.1.4. Editorial boards and program committees

Xavier Leroy is an associate editor of the Journal of Functional Programming.

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.

Sandrine Blazy co-chaired the program committee for the Journées Francophones des Langages Applicatifs (JFLA 2007).

Damien Doligez participated in the program committee of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2007).

Alain Frisch served on the jury of the SPECIF PhD thesis award and on the program committees for ICFP 2006 (ACM SIGPLAN International Conference on Functional Programming), PLAN-X 2007 (ACM SIGPLAN Workshop on Programming Language Technologies for XML), and JFLA 2007 (Journées Francophones des Langages Applicatifs).

Xavier Leroy is a member of the steering committee of the ACM Symposium on Principles of Programming Languages (POPL). Xavier Leroy served on the program committees for the Compiler Optimization meets Compiler Verification workshop (COCV 2006), the Workshop on Mechanized Metatheory (WMM 2006), the Formal Methods symposium (FM 2006), the Asian Computing Science conference (ASIAN 2006), and the European Symposium on Programming (ESOP 2007).

François Pottier was a joint program chair for the 2006 ACM SIGPLAN Workshop on ML (ML'06) and the general chair for the 2007 ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI'07). He is a member of the steering committee for the ACM International Conference on Functional Programming (ICFP).

Didier Rémy was a member of the program committee of the European Symposium on Programming (ESOP 2006) and of the international workshop Foundations and Developments of Object-Oriented Languages (FOOL/WOOD 2006).

9.1.5. PhD and habilitation juries

Damien Doligez was a member of the PhD jury of his student Richard Bonichon (University Paris 6, december 2006).

Xavier Leroy was a reviewer (*rapporteur*) for the PhD theses of June Andronick (University Paris Sud, march 2006) and Tamara Rezk (University of Nice, november 2006).

Didier Rémy was a member of the PhD jury of Tomasz Blanc (École Polytechnique, november 2006).

François Pottier was a reviewer (*rapporteur*) for the PhD theses of Julien Signoles (Université Paris 11, july 2006) and Vincent Cremet (EPFL, may 2006).

9.1.6. The Caml user community

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.

9.2. Teaching

9.2.1. Supervision of PhDs and internships

Damien Doligez is co-advisor of Richard Bonichon's PhD, with Thérèse Hardin (U. Paris 6).

François Pottier supervises the PhD thesis of Yann Régis-Gianas.

Sandrine Blazy supervised the master's internship of Thomas Moniot (ENSIIE, 6 months).

Alain Frisch supervises the PhD thesis of Kim Nguyen, together with Giuseppe Castagna and Véronique Benzaken.

Xavier Leroy is PhD advisor for Zaynah Dargaye and Jean-Baptiste Tristan. He supervised the Master's internship of Jean-Baptiste Tristan.

Michel Mauny supervised Nicolas Pouillard's internship for his engineering diploma at EPITA. In cooperation with Emmanuel Chailloux (PPS & LIP6), Michel Mauny supervises Grégoire Henry's PhD.

Didier Rémy supervises Boris Yakobowski's PhD.

9.2.2. Graduate courses

The Gallium project-team is strongly involved in the *Master Parisien de Recherche en Informatique* (MPRI), a 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 about 30 students.

Xavier Leroy gave a 6-hour lecture on abstract machines and compilation of functional languages at the "École Jeune Chercheurs en Programmation" (Toulouse, june 2006), a summer school attended by about 40 first-year PhD students.

9.2.3. Undergraduate courses

François Pottier is a part-time assistant professor (*professeur chargé de cours*) at École Polytechnique. He taught the Compilation course to third-year students and one of the student groups ("*petite classe*") for the Foundations of Computer Science course.

Didier Rémy was a part-time professor at École Polytechnique until September 2006. He taught a course on Operating systems principles and programming to third-year students. Maxence Guesdon was teaching assistant for this course.

Yann Régis-Gianas was teaching assistant for a programming module for second-year students at university Paris 7, from January to June.

Boris Yakobowski was teaching assistant for a project module for first-year students at university Paris 7, from January to June. He was also teaching assistant on a course on logical tools for second-year students, from September to December.

9.3. Participation in conferences and seminars

9.3.1. Participation in conferences

POPL: Principles of Programming Languages (Charleston, USA, january).
Xavier Leroy presented [26]. Yann Régis-Gianas presented [28]. François Pottier attended.

PLAN-X: Workshop on Programming Language Technologies for XML (Charleston, USA, january).
Alain Frisch presented [22].

JFLA: Journées Francophones des Langages Applicatifs (Pauillac, France, january).
Attended by Sandrine Blazy, Damien Doligez, and Michel Mauny.

ESOP: European Symposium on Programming (Vienna, Austria, march).
Xavier Leroy presented [25]. Sandrine Blazy and Didier Rémy attended.

FRCSS: Workshop on Future Research Challenges for Software and Services (Vienna, Austria, april).
Xavier Leroy presented [20]. Berke Durak attended.

RMLL: Rencontres Mondiales du Logiciel Libre (Nancy, France, july).
Attended by Berke Durak and Xavier Leroy.

FM: Formal Methods symposium (Hamilton, Canada, august).
Xavier Leroy presented [15]. Sandrine Blazy attended.

ICFP: Int. Conf. on Functional Programming (Portland, USA).
Alain Frisch presented [21]. Michel Mauny and Nicolas Pouillard attended.

LPAR: Logic for Programming, Artificial Intelligence, and Reasoning (Phnom Penh, Cambodia, November). Richard Bonichon presented [17].

PARISTIC: Panorama des Recherches Incitatives en STIC (Nancy, France, november).
Xavier Leroy presented an overview talk and a poster on the Compcert project.

9.3.2. Invitations and participation in seminars

Alain Frisch gave a talk on practical uses of the OCamlDuce langage during a visit at the Jane Street Capital company (New York). He presented his work on streaming XML transformation during a meeting of the Tralala ACI (XML Transformation Languages: logic and applications) in Marseille, in a seminar at the INRIA-Microsoft Joint Reserach Laboratory, and in the students' seminar at ENS Paris.

Alain Frisch presented his work on practical exact type-checking for macro-tree transducers in the seminar of the MOSTRARE group at INRIA Futurs (Lille) and in a seminar at the TUM technical university (Munich, three day visit).

Alain Frisch visited the Protheo group in LORIA (Nancy) and gave one talk about the CDuce language and another one about the theory of semantic subtyping underpinning this language.

Michel Mauny and Nicolas Pouillard visited Jim Grundy and John O'Leary at Intel's Strategic CAD Labs in Hillsboro.

François Pottier visited Andrew Pitts at the University of Cambridge, where he gave a talk.

Yann Régis-Gianas gave a talk about Menhir at PPS (U. Paris 7) and another talk about the design of programming languages for certified software at LRI (U. Paris Sud).

Didier Rémy participated to the IFIP working group in Boston in June. At this occasion, he visited Benjamin Pierce and his group at the University of Pennsylvania and David Walker at Princeton University.

Andrew Tolmach visited Microsoft Research, Cambridge, in october, as part of an ongoing collaboration with Simon Peyton Jones on lightweight concurrency in the Glasgow Haskell compiler.

Boris Yakobowski presented his PhD thesis work at the student seminar of ENS Paris.

Sandrine Blazy gave a talk about separation logic for C minor at PPS (U. Paris 7).

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-Verlag, 2002, p. 413–537.

Year Publications

Doctoral dissertations and Habilitation theses

[10] R. BONICHON. *Tableaux et Déduction Modulo*, Ph. D. Thesis, University Paris 6, December 2006.

Articles in refereed journals and book chapters

[11] E. CHAILLOUX, M. MAUNY. *Programmation fonctionnelle*, in "Encyclopédie des systèmes d'information", Éditions Vuibert, 2006, p. 1016–1027.

[12] F. POTTIER, N. GAUTHIER. *Polymorphic Typed Defunctionalization and Concretization*, in "Higher-Order and Symbolic Computation", vol. 19, n^{o} 1, March 2006, p. 125–162, http://gallium.inria.fr/~fpottier/publis/fpottier-gauthier-hosc.pdf.

Publications in Conferences and Workshops

[13] A. W. APPEL, X. LEROY. *A list-machine benchmark for mechanized metatheory*, in "Proc. Int. Workshop on Logical Frameworks and Meta-Languages (LFMTP'06)", Electronic Notes in Theoretical Computer Science, 2006, http://gallium.inria.fr/~xleroy/publi/listmachine-lfmtp.pdf.

[14] Y. BERTOT, B. GRÉGOIRE, X. LEROY. *A structured approach to proving compiler optimizations based on dataflow analysis*, in "Types for Proofs and Programs, Workshop TYPES 2004", Lecture Notes in Computer Science, vol. 3839, Springer-Verlag, 2006, p. 66-81, http://gallium.inria.fr/~xleroy/publi/proofs_dataflow_optimizations.pdf.

[15] 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-Verlag, 2006, p. 460–475, http://gallium.inria.fr/~xleroy/publi/cfront.pdf.

[16] J. BOENDER, R. DI COSMO, B. DURAK, X. LEROY, F. MANCINELLI, M. MORGADO, D. PINHEIRO, R. TREINEN, P. TREZENTOS, J. VOUILLON. *News from the EDOS project: improving the maintenance of free software distributions*, in "VIIth Workshop on Free Software", 2006, http://www.edos-project.org/xwiki/bin/download/Main/Publications/wsl06.pdf.

[17] R. BONICHON, O. HERMANT. *A semantic completeness proof for TaMeD*, in "Logic for programming, artificial intelligence, and reasoning, LPAR 2006", Lecture Notes in Artificial Intelligence, vol. 4246, Springer-Verlag, 2006, p. 167–181.

[18] R. BONICHON, O. HERMANT. *On Constructive Cut Admissibility in Deduction Modulo*, in "TYPES 2006 conference", Lecture Notes in Computer Science, Accepted for publication in the post-workshop proceedings, Springer-Verlag, 2006.

[19] Z. DARGAYE. *Décurryfication certifiée*, in "Journées Francophones des Langages Applicatifs (JFLA'07)", Accepted for publication, to appear, INRIA, 2007.

[20] R. DI COSMO, B. DURAK, X. LEROY, F. MANCINELLI, J. VOUILLON. *Maintaining large software distributions: new challenges from the FOSS era*, in "Proceedings of the FRCSS 2006 workshop", 2006, http://gallium.inria.fr/~xleroy/publi/edos-frcss06.pdf.

[21] A. FRISCH. *OCaml + XDuce*, in "Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming", ACM Press, September 2006, p. 192–200, http://doi.acm.org/10.1145/1159803.1159829.

[22] A. FRISCH. *OCaml + XDuce*, in "Workshop Programming Language Technologies for XML (PLAN-X) 2006", January 2006.

[23] A. FRISCH, K. NAKANO. *Streaming XML Transformations Using Term Rewriting*, in "Workshop Programming Language Technologies for XML (PLAN-X 2007)", January 2007.

[24] G. HENRY, M. MAUNY, E. CHAILLOUX. *Typer la dé-sérialisation sans sérialiser les types*, in "Journées francophones des langages applicatifs", INRIA, January 2006.

[25] X. LEROY. *Coinductive big-step operational semantics*, in "European Symposium on Programming (ESOP'06)", Lecture Notes in Computer Science, vol. 3924, Springer-Verlag, 2006, p. 54-68, http://gallium.inria.fr/~xleroy/publi/coindsem.pdf.

[26] 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.

[27] F. MANCINELLI, R. DI COSMO, J. VOUILLON, J. BOENDER, B. DURAK, X. LEROY, R. TREINEN. *Managing the Complexity of Large Free and Open Source Package-Based Software Distributions*, in "21st IEEE Int. Conf. on Automated Software Engineering (ASE 2006)", IEEE Computer Society Press, 2006, p. 199–208, http://doi.ieeecomputersociety.org/10.1109/ASE.2006.49.

[28] F. POTTIER, Y. RÉGIS-GIANAS. *Stratified type inference for generalized algebraic data types*, in "33rd ACM symposium on Principles of Programming Languages", ACM Press, January 2006, p. 232–244, http://gallium.inria.fr/~fpottier/publis/pottier-regis-gianas-05.pdf.

[29] 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", To appear, ACM Press, January 2007, http://pauillac.inria.fr/~remy/publications.html#Remy/mlf-graphic-types.

Internal Reports

[30] A. W. APPEL, X. LEROY. *A list-machine benchmark for mechanized metatheory*, Research report, n^{o} 5914, INRIA, 2006, http://hal.inria.fr/inria-00077531.

Miscellaneous

[31] A. W. APPEL, S. BLAZY. *Separation Logic for Small-step C Minor*, Draft, October 2006, http://gallium.inria.fr/~blazy/AppelBlazy06.pdf.

[32] F. POTTIER. *Static Name Control for FreshML*, Draft, July 2006, http://gallium.inria.fr/~fpottier/publis/fpottier-pure-freshml.pdf.

[33] D. RÉMY, B. YAKOBOWSKI. *A graphical presentation of MLF types with a linear-time incremental unification algorithm.*, Extended version of the TLDI'07 article, September 2006, http://gallium.inria.fr/~remy/project/mlf.

[34] J.-B. TRISTAN. *Certification d'un validateur de transformations*, Master's dissertation (mémoire de stage de Master 2), ENS Paris, September 2006, http://gallium.inria.fr/~tristan/rapport.pdf.

References in notes

[35] 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.

[36] J.-C. FILLIÂTRE, C. MARCHÉ. *Multi-Prover Verification of C Programs*, in "Sixth International Conference on Formal Engineering Methods (ICFEM)", Lecture Notes in Computer Science, vol. 3308, Springer-Verlag, 2004, p. 15–29.

[37] 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.

[38] L. LAMPORT. *How to write a proof*, in "American Mathematical Monthly", vol. 102, n^{o} 7, August 1993, p. 600–608.

[39] 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.

[40] X. LEROY, D. DOLIGEZ, J. GARRIGUE, D. RÉMY, J. VOUILLON. *The Objective Caml system, documentation and user's manual – release 3.09*, INRIA, October 2005, http://caml.inria.fr/pub/docs/manual-ocaml/.

[41] G. C. NECULA, S. MCPEAK, S. P. RAHUL, W. WEIMER. *CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs*, in "Compiler Construction : 11th International Conference, CC 2002", Lecture Notes in Computer Science, vol. 2304, Springer-Verlag, 2002, p. 213-228.

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

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

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

[45] 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.

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

[47] 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.