INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Project-Team Cristal

Type-safe programming, modularity and compilation

Rocquencourt

THEME SYM

Activity Report

2005

Table of contents

1. Team

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

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

Administrative assistant
Nelly Maloisel [TR INRIA]

Scientific staff
Sandrine Blazy [Assistant professor, CNAM/IIE]
Damien Doligez [Research scientist (CR), INRIA]
Alain Frisch [Research scientist, Corps des Télécoms]
Gérard Huet [Senior research scientist (DR), INRIA, also affiliated with project Signes]
Michel Mauny [Senior research scientist (DR), INRIA, on secondment as professor at ENSTA since August 2005]
François Pottier [Research scientist (CR), INRIA]
Pierre Weis [Senior research scientist (DR), INRIA]

Technical staff
Berke Durak [Ingénieur expert]
Maxence Guesdon [IR INRIA, 40% Cristal, 60% Miriad]
Virgile Prevosto [Ingénieur expert]

Ph.D. students
Richard Bonichon [MENRT grant, university Paris 6]
Daniel Bonniot [university Paris 7, until November 2005]
Zaynah Dargaye [IDF grant, university Paris 7, since October 2005]
Nadji Gauthier [MENRT grant, university Paris 7]
Benoît Razet [MENRT grant, university Paris 6, since October 2005]
Yann Régis-Gianas [MENRT grant, university Paris 7]
Boris Yakobowski [ENS Cachan, university Paris 7]

Student interns
Arthur Charguéraud [ENS Lyon, April–June 2005]
Zaynah Dargaye [University Paris 7, March–August 2005]
Grégoire Henry [University Paris 6, March–August 2005]
Benoît Razet [University Paris 6, April–September 2005]

2. Overall Objectives

2.1. Overall Objectives

The research carried out in the Cristal group is centered on type systems and related program analyses, applied to functional, object-oriented, and modular programming languages. The Caml language embodies many of our research results. Our work spans the whole spectrum from theoretical foundations and formal semantics to language design, efficient and robust implementations, and applications to real-world problems. The conviction of the Cristal group is that high-level, statically-typed programming languages greatly enhance program reliability, security and efficiency, during both development and maintenance.

3. Scientific Foundations

3.1. Type systems

Typing is a fundamental concept in programming: it allows the specification and automatic verification of consistent handling of data in programs. Moreover, when applied to functions, classes, and modules, typing helps structuring and managing large and complex programs.

The Cristal group studies type systems and typing techniques with the aim of designing safe programming languages and tools, whose properties are formally established.

3.1.1. The Hindley-Milner type system

The ML language, designed circa 1978, supports automated compile-time type inference and checking. Its type system supports a form of *parametric polymorphism* that allows giving types to generic algorithms, that is, algorithms that work uniformly on a variety of data types. This type system, as well as its type inference algorithm, are formally specified. Its main property is expressed through a soundness theorem: *Well-typed programs cannot go wrong*.

The ML type system has been the subject of numerous studies and extensions during the last two decades. Even if they are meant to be reusable in other contexts, many of the results of the Cristal group in this area are primarily applied in the Caml language, a dialect of ML developed in the Cristal group.

3.1.2. Typing objects

Although object-oriented programming is widely used in industry, very few OO programming languages have a clear semantics and a formal type system. Different approaches have been proposed in order to lay firm type-theoretic foundations for OO programming.

The main approach followed by the Cristal group relies on Didier Rémy's PhD work [57] that proposed a type system for extensible records allowing type inference. This approach was first extended to objects by Rémy, in his ML-ART prototype [58]. Rémy and Vouillon [59] later added a fully-fledged class-based object layer to Caml, which then became Objective Caml (OCaml). This OO layer is compatible with type inference, and supports advanced OO concepts, such as multiple inheritance, parameterized classes, and "my type" specialization. It can statically type idioms, such as container classes or binary methods, that require dynamic type-checking in conventional OO languages, such as Java.

Another approach to type inference for object-oriented programming, featuring implicit subsumption and based on subtyping constraints, was studied by François Pottier in his PhD dissertation [6], [7]. Our current efforts are directed towards multi-methods (that is, method dispatching based on all the arguments of a method) and the introduction of objects in concurrent programming languages, such as the ones based on the join-calculus [49].

3.1.3. Typing modules

Module systems provide an alternative to classes for structuring and decomposing large programs. The Caml module system [54] provides advanced mechanisms for packaging data types with associated operations, maintaining multiple, possibly abstract interfaces for these packages, and parameterizing a module over other modules. It is based on the Standard ML module system but offers improved support for separate compilation, through strict adherence to purely syntactic module types.

While modules are fundamentally different from objects and classes, it would be preferable for modules to take advantage of the possibilities offered by classes such as inheritance and overriding, as well as mutual recursion. These issues are currently studied by the Cristal group in the general framework of *mixin modules* [46].

Finally, extensional polymorphism provides identifier overloading and generic functions, whose behavior is guided by the type of their arguments. Extensional polymorphism [50] also allows modelling computations that dynamically depend on types (input/output, values with dynamic types).

3.1.4. First-class polymorphism and higher-order types

On the one hand, the Hindley-Milner type system has a restricted form of polymorphism: its types are first-order (they do not contain universal quantifiers), and universal quantification is allowed only at the top level, as part of type schemes. This restriction allows ML to support automated type inference, without the need for type annotations in programs.

On the other hand, system F allows universal quantifiers to occur arbitrarily deep within a type expression. This “first-class” polymorphism provides greater expressiveness, at the expense of type inference, which becomes undecidable.

System F’s types are second order, i.e., universal quantification is limited to types. In addition, system F_ω offers higher-order types: it also allows universal quantification over type functions (or, in other words, over parameterized types). This provides even greater expressiveness, making it possible to express notions in the core language, which in ML must be relegated to the level of the module language.

The search for type systems that combine some form of first-class and possibly higher-order polymorphism with a decidable or tractable type inference problem, constitute an important research objective at Cristal [2], [4]. Results in this area would increase the expressiveness of ML-style languages, strengthen their abstraction capabilities, and therefore facilitate code reuse.

3.1.5. Typing and confidentiality

Numerous programs manipulate sensitive corporate or private data. A static *information flow* analysis provides a way of preventing such confidential information from being leaked to untrusted third parties. The Cristal group applies type inference techniques to this problem [8].

3.2. The Caml programming language

Caml belongs to the ML family of languages. Like all languages of this family, Caml supports both functional and imperative programming styles, Hindley-Milner static typing with type inference, and features a powerful module system. Caml also supports class-based object-oriented programming, and has an efficient compiler.

Initially built around a functional kernel with imperative extensions, the ML language evolved in two independent ways during the 80’s. First, a group headed by Robin Milner designed Standard ML (SML), whose most innovative feature was its module system, designed by David MacQueen. In parallel, Caml was first designed and developed in the Formel group at INRIA, in collaboration with members of the C.S. Lab. of École Normale Supérieure in Paris, and then in the Cristal group. Xavier Leroy designed and implemented a module system similar to that of SML, but with a greater focus on separate compilation. Didier Rémy and Jérôme Vouillon built the object layer. The version of Caml including these features is called Objective Caml (OCaml).

From an implementation point of view, Caml has advanced the state of the art in compiling functional languages. Initially based on the Categorical Abstract Machine, which was implemented on top of a Lisp runtime system, Caml’s execution model was first changed to a high-performance bytecoded virtual machine designed by Xavier Leroy and complemented by Damien Doligez’s generational and incremental garbage collector [48]. The resulting implementation, Caml-Light, has low memory requirements and high portability, which made it quite popular for education.

Then, on the way from Caml-Light to Objective Caml, Xavier Leroy complemented the bytecode generator with a high-performance native code compiler featuring optimizations based on control-flow analyses, good register allocation, and code generators for 9 different processors. The combination of the two compilers provides portability and short development cycle, thanks to the bytecode compiler, as well as excellent performance, thanks to the native code generator.

4. Application Domains

4.1. Software reliability

One of the aims of static typing is early detection of programming errors. In addition, typed programming languages encourage programmers to structure their code in ways that facilitate debugging and maintenance. Judicious uses of type abstraction and other encapsulation mechanisms allow static type checking to enforce program invariants.

Typed functional programming is also an excellent match for the application of formal methods: functional programs lend themselves very well to program proof, and the Coq proof assistant can extract Caml code directly from Coq specifications and proofs.

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 XDuce and CDuce 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 perfectly suitable for fast prototyping and the gradual evolution of software prototypes into final applications, as advocated by the popular “extreme programming” methodology.

4.4. Programming secure applications

Strongly-typed programming languages are inherently well-suited to the development of high-security applications, because by design they prevent a number of popular attacks (buffer overflows, executing network data as if it were code, etc). Moreover, the methods used in designing type systems and establishing their properties are also applicable to the specification and verification of security policies.

4.5. Web programming

Web programming is an application domain on which the Cristal group has worked in the past, e.g. via the MMM applet-enabled browser [56] and the V6 smart Web proxy. Currently, the use of XML documents, online or offline, and the need for typing their transformations opens an application area that perfectly matches the strengths of languages such as OCaml. Our recent work in this area concentrates on statically-typed XML transformation languages such as XDuce and CDuce, where XML DTDs and Schemas are interpreted as types and statically enforced through type-checking of XML transformations. We also study the integration of these domain-specific languages for XML processing with general-purposes languages such as OCaml.

4.6. 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*. Caml Light and OCaml are also used in engineering schools, colleges and universities in France, the US, and Japan.

4.7. Computational linguistics

Computational linguistics focuses on the processing of natural languages by computer programs. This rapidly expanding field is multi-disciplinary in nature, and involves several areas that are extensively represented at INRIA: syntactic analysis, computational logic, type theory. During the last two years, Gérard Huet has been investigating this new domain, and has shown that OCaml and its Camlp4 preprocessor have been highly effective for the development of natural language processing components.

5. Software

5.1. Advanced software

The following software developments are publicly distributed (generally under Open Source licenses), actively supported, and used outside our group.

5.1.1. Caml Light

Participants: Xavier Leroy, Damien Doligez, Pierre Weis.

Caml Light is a lightweight, portable implementation of the core Caml language. It is still actively used in education, but most other users have switched over to its successor, Objective Caml.
Web site: http://caml.inria.fr/index.en.html.

5.1.2. Objective Caml

Participants: Xavier Leroy, Damien Doligez, Jacques Garrigue [Kyoto University], Maxence Guesdon, Luc Maranget [project Moscova], Jérôme Vouillon [CNRS, U. Paris 7], Pierre Weis.

Objective Caml is our main 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 comprehensive standard library, as well as a replay debugger, lexer and parser generators, and a documentation generator.
Web site: http://caml.inria.fr/index.en.html.

5.1.3. Camlp4

Participant: Michel Mauny.

Camlp4 is a source pre-processor for Objective Caml that enables defining extensions to the Caml syntax (such as syntax macros and embedded languages), redefining the Caml syntax, pretty-printing Caml programs, and programming 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.
Web site: http://caml.inria.fr/index.en.html.

5.1.4. 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, a 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 integratation 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.

Several version have been released in 2005. They include improvements over run-time efficiencies, a stronger support for XML Schema and for XML Namespaces, and various additions to the surface language and improvements of error messages.

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

5.1.5. Cameleon

Participants: Maxence Guesdon, Pierre-Yves Strub [ingénieur expert MIRIAD].

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/.

5.1.6. ActiveDVI

Participants: Pierre Weis, Didier Rémy, Roberto Di Cosmo [U. Paris 7], Jun Furuse [U. Tokyo], Alexandre Miquel [U. Paris 7].

ActiveDVI is a programmable viewer for DVI files produced by the TeX and LaTeX text processors. It provides many fancy graphic effects and can use any X Windows application as plug-in, therefore allowing a demo to be embedded in a presentation, for instance. ActiveDVI provides an excellent Unix/LaTeX-based alternative to PowerPoint presentations. In particular, it supports time recording in slide shows: a lecture can be post-synchronized with the speaker's words. ActiveDVI uses the Camlimages library developed by J. Furuse and P. Weis for displaying embedded images.

Web site: http://pauillac.inria.fr/advi/.

5.1.7. WhizzyTeX

Participant: Didier Rémy.

WhizzyTeX is an Emacs extension that allows previewing of a LaTeX document in real-time during editing.

Web site: http://cristal.inria.fr/whizzytex/.

5.2. Prototype software

The following software developments are prototypes used mostly within our group, either for experimental purposes or to support our specific needs. Nonetheless, they are all publicly distributed.

5.2.1. OCamlDuce

Participant: Alain Frisch.

OCamlDuce is a modified version of OCaml which integrates most features from CDuce: 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 which also need support for XML, with the same advantages as CDuce (syntactic support for XML and XML Namespaces, complex pattern matching over XML, efficient evaluation, strong typing guarantees).

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

5.2.2. *Cαml*

Participant: François Pottier.

Cαml (pronounced “alpha-Caml”) [32] 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/.

5.2.3. *The Zenon theorem prover*

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).

5.2.4. *The Zen computational linguistics toolkit*

Participant: Gérard Huet.

Zen is a toolkit for computational linguistics developed in Objective Caml by Gérard Huet. It powers Gérard Huet’s Sanskrit Site, at http://sanskrit.inria.fr/.

Web site: http://sanskrit.inria.fr/ZEN/.

5.2.5. *Htmlc*

Participant: Pierre Weis.

Htmlc is an HTML template file expander that produces regular HTML pages from source files containing generated text fragments. These fragments can be the results of variable expansions, the output of an arbitrary Unix command (for instance, the last modification date of a page), or shared HTML files (such as a common page header or footer). Htmlc offers a server-independent way of defining templates that factor out the repetitive parts of HTML pages.

Web site: http://pauillac.inria.fr/htmlc/.

6. New Results

6.1. Type systems

6.1.1. *Type inference for generalized algebraic data types*

Participants: François Pottier, Yann Régis-Gianas, Nadji Gauthier.

Generalized (or *guarded*) algebraic data types subsume the concepts known in the literature as *indexed types*, *guarded recursive datatype constructors* and *first-class phantom types*, and are closely related to *inductive types* as found in the Coq proof assistant. They have the distinguishing feature that, when typechecking a function defined by cases, every branch may be checked under different assumptions about the type variables in scope. This mechanism allows exploiting the presence of dynamic tests in the code to produce extra static type information. This enables rich invariants over data structures to be expressed in the data type declarations and enforced automatically by typing.

Continuing preliminary work by Simonet and Pottier [39], François Pottier and Yann Régis-Gianas studied the type inference problem in the presence of generalized algebraic data types. They designed a solution that modularly combines a traditional constraint-based type inference layer with an approximate local type inference layer [33]. Yann is now planning to implement this proposal into the Objective Caml compiler.

François Pottier and his students studied several applications of generalized algebraic data types, two of which were published this year: static typing of pushdown automata for LR parsing [34] and static typing of defunctionalized forms of higher-order functional programs [18].

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

Participants: Didier Rémy, Boris Yakobowski, Didier Le Botlan [École des Mines de Nantes].

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 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. Therefore, the user must 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. 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 [53]. Didier Le Botlan and Didier Rémy have continued their work on MLF focussing on the simplification of the formalism. An interesting restriction of MLF that retains much of expressivity while been much 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.

As part of his PhD work, Boris Yakobowski 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. His work has already led to two interesting results. First, graph types appear to be much more canonical that syntactic types; in particular the instance relations can be defined by a few simple, atomic rules, which significantly increases our confidence in the instance relation. Second, graph types exposed a new, linear-time unification algorithm that is simpler than previous proposals. Graph types may be split into the superposition of a structure graph and a binding graph. Unification on graph types can be decomposed into standard first-order unification on the structure graphs and a simple computation on their bindings graphs. Preliminary results have been presented at the third APPSEM II workshop [35]; a conference submission is in preparation.

6.1.3. Exploring partial type inference for System F based on predicative type-containment

Participants: Didier Rémy, François Pottier, Arthur Charguéraud.

In parallel with MLF, Didier Rémy has also explored another approach to perform partial type inference in System F based on predicative type-containment. Type containment is an implicit structural instance relation on types that allows some binders to be moved implicitly in types. This is particularly useful for type inference, as it makes type instantiation commute with applications. However, since the type-containment relation is undecidable, we restrict to implicit predicative type-containment and recover impredicative instantiations using explicit annotations.

For predicative System F, we identify a type system with a simple logical specification for which we can perform ML-style type inference. However, too many type annotations are still required in practice. Therefore, we also proposed a *stratified type inference* approach, which consist of a preliminary elaboration phase that solely propagates source type annotations in some simple but incomplete manner followed by type-inference for the elaborated term. This approach works remarkably well for the predicative fragment of System-F. However, elaboration for impredicative instantiations remains unsatisfactory.

These results have been presented at the *International Conference on Functional Programming* (ICFP) in September 2005 [36]. A prototype implementation was developed by Arthur Charguéraud (ENS-Lyon) during his Spring internship in our group.

6.2. XML transformation languages

6.2.1. Extending Caml with XML types

Participant: Alain Frisch.

Objective Caml type system 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 manipulation XML documents in functional languages have been proposed recently. They usually builds 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, path navigation.

In order to facilitate the manipulation of XML documents in Caml, one may want to integrates some features from these domain specific languages into Objective Caml. However, it seems quite challenging to merge in a single type systems ML-like type inference and XML types based on tree automata. A natural approach would be to turn to the HM(X) constraint-based formulation of the ML type system. Unfortunately, the constraint domain naturally arising from embedding XML types into HM(X) is undecidable, because it can express the inclusion of context-free languages. Moreover, turning to a constraint-based approach might not be desirable from a practical point of view (for efficiency, readability of error messages and displayed types, and integration with existing implementations).

Alain Frisch has developed a type system and a typing algorithm for a merger between OCaml and CDuce. This system preserves nice theoretical and practical properties from ML and CDuce. It has been implemented in OCamlDuce, a modified version of OCaml which embeds CDuce. OCamlDuce was implemented by merging OCaml and CDuce code bases and adding only a relatively small amount of glue code.

The theoretical foundation of OCamlDuce's type system are described in a paper [27] to be presented at the PLAN-X 2006 (Programming Language Technologies for XML) workshop.

6.2.2. Error mining for XML pattern matching

Participants: Alain Frisch, Dario Colazzo [LRI, U. Paris Sud], Giuseppe Castagna [ENS Paris].

In the design of type systems for XML programming languages based on regular expression types and patterns, the focus has been over *result analysis*, with the main aim of statically checking that a transformation always yields data of an expected output type. While being crucial for correct program composition, result analysis is not sufficient to guarantee that patterns used in the transformation are correct. We studied more subtle notions of errors in patterns in order to detect "dead" or useless sub-patterns. Compared to e.g. ML pattern matching, these notions become tricky because of boolean connectives (union, intersection, negation) in types and patterns, and because patterns are required to comply with a first-match disambiguation policy. We formalized the errors by instrumenting an operational semantics for patterns, and we designed a sound and complete algorithm to detect those errors. These results have been presented in [24].

6.2.3. Exact type checking for XML transformations

Participant: Alain Frisch.

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 want to introduce explicit typing approximations (under the appearance of type annotations) to deal with that.

One of the reasons which can explain the relative lack of interest from the programming language community for tree transducer techniques is that algorithms are very expensive (towers of exponentials). We hope that a reformulation of the algorithms will yield usable algorithms for practical cases. As a first step and as a proof of concept, we implemented and released `xtrans`, an exact type-checker for a tiny XML transformation language based on top-down tree transducers with regular look-ahead. At least for the small and medium-sized transformations we tried, `xtrans` is very efficient (it takes well below one second to type-check all the examples). We plan to extend it with accumulators so as to reach the expressive power of so-called macro tree transducers.

6.3. Certified compilation

Formal methods are increasingly being applied to safety-critical software in order to increase their reliability and meet the requirements of the highest levels of software certification. However, formal methods are applied to the source code of the software, written in C or higher-level languages; but what actually runs in the safety-critical computer is the machine code generated from the sources by a compiler. Making sure that the compiler does not introduce bugs is a difficult process: extensive testing of the executable code can help, but the highest levels of certification in e.g. avionics actually require manual inspection of the assembly code produced by the compiler, which is painful and costly.

A better solution to this problem is to apply formal methods to the compiler itself. We are currently exploring the feasability of developing a *realistic, certified compiler*, that is, a compiler usable in the context of critical embedded software and that comes with a proof that the generated assembly code has the same semantics as the source code. This proof is developed and machine-checked using the Coq proof assistant. These explorations started in the context of an INRIA *Action de Recherche Coordonnée* and continue in the context of an ANR *Action de Recherche Amont*, involving projects Cristal and Marelle as well as CNAM (Cedric lab) and University Paris 7 (PPS lab).

6.3.1. Certification of a compiler back-end

Participants: Xavier Leroy, Damien Doligez, Laurence Rideau [project Marelle].

Since summer 2004, Xavier Leroy has developed and certified a compiler back-end that translates from Cminor (a low-level imperative language with structured control) to assembly language for the PowerPC processor, using 4 intermediate languages. This compiler generates compact, reasonably efficient PowerPC code, thanks in particular to a state of the art register allocator and a couple of optimization passes (constant propagation and common subexpression elimination).

Last year, only two translation phases of this compiler were certified: translation of Cminor to a control-flow graph of register transfer instructions, and register allocation with introduction of function calling conventions. This year, Xavier Leroy proved semantic preservation for the remaining translation phases: constant propagation, common subexpression elimination, linearization of the control-flow graph, placement of stack-allocated temporaries in memory-resident stack frames (with insertion of matching “reload” and “spill” instructions), and final generation of PowerPC assembly code. Therefore, we now have a complete, certified translation chain from Cminor to PowerPC.

Damien Doligez contributed Coq correctness proofs for the “finite map” data structure and for type reconstruction over the register transfer intermediate language. Laurence Rideau integrated in this development her earlier certification (with Bernard Serpette) of the “parallel assignment” compilation algorithm, which plays a crucial role in the explicitation of function calling conventions. This certification is described in a paper presented at JFLA 2005 [55].

Most of the compiler back-end is written directly in the Coq specification language, as pure functions. Using the extraction facility of Coq (which translates Coq specifications to Caml code), and adding a Cminor parser and PowerPC printer written directly in Caml, we obtained Caml source code for the whole compiler, making it directly executable. Benchmarking on a set of small hand-written Cminor program shows that the performance of the generated PowerPC code is entirely acceptable: performance is much better than that of the `gcc` compiler at optimization level 0, and only slightly inferior to that of `gcc` at optimization level 1.

This work is described in a paper [31] that will be presented at the POPL conference in january 2006.

6.3.2. *Certification of a compiler front-end for a subset of the C language*

Participants: Sandrine Blazy, Zaynah Dargaye, Xavier Leroy.

In parallel with the certification of the back-end, we are conducting several experiments in developing and certifying compiler front-ends that target the Cminor intermediate language. The first such experiment generates Cminor code from a subset of the C language similar to that used for programming critical systems: it features all the arithmetic operators of C, pointers and arrays with pointer arithmetic, function pointers, and all the structured control statements of C, but excludes unstructured control (`goto`, `switch`, `longjmp`), dynamic allocation, "struct" and "union" types, and block-local variables.

Sandrine Blazy formalized the dynamic semantics (evaluation rules) for this subset of the C language. This dynamic semantics is written in “big-step” style, also called natural semantics. To give meanings to C’s type-dependent operations, it assumes that the expressions are well-typed and annotated by their types. Sandrine Blazy and Zaynah Dargaye also formalized the static semantics of the subset of the C language: typing rules and a type-checking algorithm producing type-annotated terms.

Zaynah Dargaye (as part of her Master’s internship) and Xavier Leroy developed and proved correct (in Coq) a translation from this subset of C to the Cminor intermediate language. This translation performs resolution of operator overloading, explicitation of address computations, translation of C’s control structures into Cminor’s simpler control structures, and explicit stack-allocation of local arrays and local variables whose addresses are taken (using the `&` operator). Proving the latter transformation involves complex reasoning over the memory states.

This preliminary experiment is encouraging: Cminor appears adequate as a target intermediate language, and the certification could be completed within the time limits of a Master’s internship. However, significant work remains to be done to handle a larger subset of the C language and validate our semantics against those used by other research groups working on the verification and static analysis of C source code.

6.3.3. *Certification of a compiler front-end for a small functional language*

Participants: Zaynah Dargaye, Xavier Leroy.

For the beginning of her PhD thesis, Zaynah Dargaye, under the supervision of Xavier Leroy, investigates the development and certification of a translator from a small call-by-value functional language (mini-ML) to Cminor. This work is just beginning and currently focuses on representation strategies for function closures.

6.3.4. Adequate on-machine operational semantics

Participants: Xavier Leroy, Sandrine Blazy.

A recurring issue in the formal certification of compilers is the development of appropriate operational semantics for the source, intermediate and target languages. Most of our semantics so far are of the "big-step operational" kind, also called natural semantics. These semantics relate formally a program to its final result, and lend themselves well to the proof of compiler-like program transformations. However, these semantics apply only to terminating programs, and do not allow observing intermediate states during program execution. These two features are distinct disadvantages for the intended application domain: embedded software is typically reactive in nature, meaning that programs do not normally terminate and their interactions with the outside world is what matters, not their final result. We are therefore investingating other forms of on-machine semantics.

An obvious choice is "small-step operational semantics" based on (finite or infinite) sequences of elementary reductions of the program source. Sandrine Blazy wrote a small-step semantics of the subset of the C language, and proved the equivalence between small-step and big-step semantics for terminating programs. However, such reduction semantics are difficult to exploit when proving the correctness of compiler transformations such as the generation of a control-flow graph from a structured program.

Another direction that we investigated is co-inductive big-step semantics, where evaluation rules still relate programs and program fragments to their final outcome, but some of the rules are interpreted co-inductively (infinite derivation trees) instead of inductively (finite derivation trees) as usual. The co-inductive interpretation enables these semantics to describe the evaluation of non-terminating programs. Xavier Leroy formalized this approach in Coq for a tiny functional language (call-by-value λ-calculus) and 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 are described in a paper [30] accepted at the ESOP 2006 conference.

6.4. Software-proof codesign

6.4.1. Focal

Participants: Damien Doligez, Richard Bonichon, Virgile Prevosto, Pierre Weis.

Focal — a joint effort with the LIP6 laboratory (U. Paris 6) and the CEDRIC laboratory (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 supports the modular and incremental design of certified software libaries: starting from a set of pure-specification modules, the programmer uses inheritance to add code and proofs in a structured way. This system promotes reusability by making it easy to use a set of abstract and semi-abstract modules to embody the general aspects of the problem at hand, before going into more specialised coding.

The Focal compiler takes Focal source code, performs type checking and dependency checking to ensure the consistency of inheritance and parameterisation, and produces two outputs: OCaml code and an incomplete Coq proof script. The OCaml code can be compiled and executed; it corresponds to the computational part of the Focal source. The Coq script must be completed before it can be passed to Coq for verification.

Focal proofs are done in a hierarchical language invented by Leslie Lamport in 1994. 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) and Virgile Prevosto.

Focal version 0.3.1 was released in May 2005. The release comprises the `focc` compiler, the `zenon` prover, a library of computer algebra functions, and a tool for producing documentation in XML format.

6.4.2. The Zenon theorem prover

Participants: Damien Doligez, Richard Bonichon.

Zenon is an automatic theorem prover based on the tableaux method, developed by Damien Doligez. (See section 5.2.3 for a description.) This year, Damien Doligez extended Zenon with a general treatment of transitive relations, generalizing earlier work specialized to the equality relation. He also added support for reasoning over `if`...`then`...`else`... constructs, and is currently working on generalizing this support to pattern-matching over inductive data structures.

Damien Doligez also benchmarked Zenon over the TPTP (Thousands of Problems for Theorem Provers) collection of standard problems. Currently, about 720 of the 1724 problems are solved automatically.

Initially developed as part of the Focal project, Zenon is finding other uses. At project Proval, Jean-Christophe Filliâtre and a Master's student packaged Zenon as a decision procedure for the Coq proof assistant, and used it to simplify proof obligations generated by the Why and Caduceus program provers.

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. The theoretical framework for this integration was published in [45].

6.4.3. Evaluating theorem provers for programming language verification tasks

Participants: Xavier Leroy, Andrew Appel [project Moscova].

Xavier Leroy participated in two challenges designed to compare the usability of various theorem provers and proof assistants for the purpose of mechanizing specifications and proofs in the area of programming languages semantics, type systems, and compilation.

The first is the POPLmark challenge proposed by Pierce, Weirich and Zdancewic, which proposes to formalize and prove the soundness of a type system based on system $F_{<:}$. This challenge focuses on proper handling of binders and α-conversion — a difficult task for most theorem provers. Xavier Leroy submitted a partial solution to the POPLmark challenge, developed in Coq and using the McKinna-Pollack mixed representation of variables (free variables are represented by names and bound variables by de Bruijn indices) [43].

The second challenge was designed by Andrew Appel with contributions by Xavier Leroy. It focuses on machine-level languages and proposes to prove the correctness of a type-checker for such a language. Xavier Leroy provided a solution in Coq, while Andrew Appel solved it in Twelf. A draft technical report [40] describes the challenge and gives a comparative evaluation of the two solutions.

6.5. The Objective Caml system, libraries and tools

6.5.1. The Objective Caml system

Participants: Damien Doligez, Jacques Garrigue [Kyoto university], Xavier Leroy, Maxence Guesdon, Luc Maranget [project Moscova], Michel Mauny, Pierre Weis.

This year, we released version 3.09 of the Objective Caml system, along with two updates 3.08.3 and 3.08.4 to the previous version. Damien Doligez acted as release manager for these three versions. The main novelties in these releases are:

- The introduction of private row types [51], which offer new possibilities for abstracting over the types of objects and polymorphic variants where only some of the methods and constructors are statically known.
- Additional compile-time warnings for unused local variables and unintentional applications of non-returning functions.
- A revised implementation of the `printf` formatted output facility that has a more intuitive behaviour with respect to partial applications.

- Improvements in the quality of the compiled code for class expressions, recursive module definitions, and large mutually-recursive function definitions.

In addition, about 80 minor bugs were corrected.

6.5.2. *Automating object-level α-conversion in meta-programs*

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.

To address this issue, François Pottier designed and implemented Cαml (pronounced “alpha-Caml”) [32] a tool that turns a so-called “binding specification” into an Objective Caml compilation unit. 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 code generated by Cαml offers safe and efficient implementations of the operations listed above (and more).

6.5.3. *ReFLect over Ocaml*

Participants: Virgile Prevosto, Michel Mauny, Damien Doligez.

Virgile Prevosto, in collaboration with Damien Doligez and Michel Mauny, is developing 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, and is supposed to include reflective aspects. This work, supported by a contract between INRIA and Intel, started in September.

The main goal of the first twelve months is 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. At the end of 2005, we have a first specification of core ReFLect. We devised a translation scheme for encoding ReFLect’s evaluation strategy into Ocaml, based on Michel Mauny’s previous work, as well as a few optimizations that should minimize the overhead inherent to this encoding. A prototype compiler is being developed and should be operational at the beginning of 2006.

6.5.4. *Type-safety of unmarshaled OCaml values*

Participants: Grégoire Henry, Michel Mauny.

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 studied the possibility of having unmarshaling functions that receive a representation of some type t as argument and return values of type t. This is obtained by extending the programming language with a predefined parameterized abstract type α *tyrepr* whose values are representations of the current instance of the type parameter α. With such a mechanism, the representation of type t has type t *tyrepr*. Given such a representation of type t, an unmarshaling function is then able to check that the value being unmarshaled indeed belongs to type t. When, as it is the case in OCaml, no type information is attached to marshaled values, the verification can be performed via a recursive traversal of the value, with particular care brought to sharing and polymorphism, to mutable data structures, and to cycles.

Grégoire Henry and Michel Mauny formalized their dynamic type checking algorithm for marshaled values, and proved its correctness and completeness: unmarshalled values successfully checked at type t can safely be used with this type, and ummarshalling at type t a value whose static type was t always succeed. This result is accepted for publication [28].

6.5.5. Data types with relations

Participants: Pierre Weis, Frédéric Blanqui [project Protheo].

In 2003, Pierre Weis designed and integrated in OCaml 3.07 the notion of data types with private constructors, or “private data types” for short. Data structures of these types can be inspected directly by client code, but not constructed directly: construction functions from the defining module must be invoked; these construction functions can in turn enforce some invariants over the data structure at creation time.

This year, Pierre Weis proposed to extend private data types towards *data types with relations* or *relational data types*. A relational data type is a private data type with the additional possibility to declare invariants or relations that must be verified by the values of the type. In particular, equations between these values, such as commutativity or associativity equations, can be given. These equations, interpreted as rewrite rules, are used at construction time to ensure that values of the relational data type are always in normal form with respect to these rules. Given such a definition of a relational data type, a special-purpose compiler generates Caml constructor functions that maintain this invariant.

Relational data types can be used to implement quotient types, ensuring that the values of the type are canonical representatives of their equivalence classes. More generally, the *generators and relations* presentation of mathematical structures can easily be implemented within the relational data type framework.

In collaboration with Frédéric Blanqui from the Protheo project, Pierre Weis has implemented the `mocac` compiler that generates the normalisation functions for a class of relational data types with algebraic relations. The prototype compiler handles purely algebraic properties, such as commutativity, associativity, distributivity, neutral or absorbing elements, and distributivity of an operation over another. General rewrite rules are also available to express other relations.

6.5.6. Caml for numerical programming

Participants: Pierre Weis, Hend Ben Ameur [Lamsin/ENIT], François Clément [project Estime], Roberto Di Cosmo [university Paris 7], Jérôme Jaffré [project Estime], Zheng Li [university Paris 7].

An INRIA *Action de Recherche Concertée* called Moprosco started this year. Coordinated by project Estime and involving U. Paris 7 and U. Pisa, this ARC aims at promoting the use of OCamlP3L, an OCaml-based framework for distributed parallel computing, in the scientific computation community. An early success in using OCamlP3L for code coupling problems is described in [47].

With Hend Ben Ameur and François Clément (project Estime), Pierre Weis started to write a library to exchange data between languages used in scientific computing libraries. The library will provide APIs in Fortran77, C++, Java and Caml. The Caml and C++ versions are already written and we are working on the Fortran version.

François Clément and Pierre Weis wrote a draft of a complete inverse problem resolution algorithm via the “refinement indicators” methodology. Application to visualisation and images of this algorithm leads to an interesting segmentation program for images.

6.5.7. The Caml development environment

Participants: Maxence Guesdon, Pierre-Yves Strub [ingénieur expert MIRIAD].

Maxence Guesdon continues his work on Cameleon, the integrated development environment for Objective Caml described in section 5.1.5. This year, Topcameleon, a graphical toplevel allowing the browsing of result values, was also ported to LablGTK2 and included in Cameleon. A new view was added to Cameleon, displaying the dependencies between modules and enabling access to the code from this view. Caml-get, a tool that facilitates the reuse of Caml source code that is not packaged as libraries, was integrated in Cameleon as a plug-in.

Maxence Guesdon developed Yacclib, a small library to parse and print OCamlyacc files. A `dot` output is also possible. A graphical editor for OCamlyacc files is in development, based on this library.

6.6. Formal management of software dependencies

Participants: Berke Durak, Xavier Leroy, Roberto Di Cosmo [university Paris 7], Ralf Treinen [ENS Cachan].

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 help with automatic installation and upgrading of packages on the users' side and with distribution building and detection of broken packages on the distributors' side.

This year, Roberto Di Cosmo (Paris 7) and Ralf Treinen (ENS Cachan) formalized a dependency model based on three kinds of dependencies: “requires”, “provides” and “conflicts with”. They showed that the installability problem (“is it possible to satisfy all dependencies of a given package?”) is NP-complete and that existing tools solve it either very incompletely (Debian's `APT` tool) or can take exponential time (Mandriva's `Smart` tool).

Di Cosmo and Treinen encoded the installability problem in constraint logic programming, using integer unary constraints. Attempts to solve this problem using the Mozart/Oz environment were unsuccessful (exponential time). Xavier Leroy then proposed a different encoding as a satisfiability problem over boolean formulae, and implemented a solver based on the Grasp SAT solver that appears efficient in practice: it takes a few seconds to solve the hardest instances found in the Debian package pool. The solver was successfully applied to the whole Debian pool and found a number of broken packages (not installable in any configuration).

Berke Durak was hired in september 2005 to work on the EDOS project. He pursues the boolean satisfiability approach to the problem, in particular the development of custom SAT solvers that should provide explanations of failures. He also develops other tools for package management, such as the Ara package indexer and tools to graphically display dependency graphs.

6.7. Computational linguistics

6.7.1. Computational linguistics

Participant: Gérard Huet.

Gérard Huet continued his work on developing a computational linguistics platform adapted to Sanskrit, based on applicative programming in Ocaml. The main effort in 2005 concerned curbing the overgeneration of the segmenter by a semantic analysis. Each segmentation solution, represented as a list of morphological items (inflected words tagged with their lemmatization as a root entry together with a morphological generator carring its various features), is translated into a sequence of semantic role scripts. Verbal forms become sites of actions/situations, expecting complements as role assignments. These roles depend on the regime of the verb, given its voice. For instance, a transitive verb in the active voice demands a subject in the nominative and an object in the accusative for its role saturation. Dually, nominal phrases provide the corresponding roles. Matching opposite polarities gives rise to a constraint satisfaction problem over the role features. This corresponds, in Western linguistics, to the construction of the *dependency structure* in the sense of Tesnière, as computed in computational systems based on dependency grammars (and having their analogues in feature logical programming platforms such as HPSG or LFG). In the terminology of Indian linguists like Pāṇini, we do the analysis of *kaarakas*. The constraint satisfaction problem is similar to proper typing of categorial grammar parse trees, or to the construction of a proof net in non-commutative linear logic. However, non-linear phenomena are frequent. For instance, agreement of an adjective and its qualifying noun is a kind of contraction.

The constraint satisfaction engine proceeds as a sequence of stream processors applied to the tagged sentence stream, going from right to left. Tool words are treated as postfix stream combinators - they are allowed to compute only in the past of their utterance. From this work arises the notion of a *linguistic tool* as a feature structure stream transducer. Pronouns are linguistic tools in this sense, since their purpose is to link to their anaphoric antecedent. In Sanskrit, a case study for coordination led to the implementation of the *ca* tool. This postfix conjunction has the effect of merging antecedent noun phrases with three semantic upper bound operations, respectively for gender, number, and person. These operations compute a sum in abstract interpretations on the morphological features domains. For instance, it has the effect of transducing the sequence of tagged items for “two girls and one boy” into one tag for “several male persons”, paving the way to the proper recognition of this compound item as a proper subject to a verb conjugated in the plural. After iteration of stream combinators, a final phase attempts to solve government and binding constraints, computing a compound bonus-malus score.

The constraint engine, still under design, demonstrates a remarkable filtering capacity. Very often, sentences with several hundred potential phonemic segmentations are processed successfully, in the sense that most segmentation candidates are rejected as dubious, their bonus-malus score being below some threshold, while the intended meaning is retained. Rejection scores of 98% are frequent. This is rather encouraging, and it is expected that in early 2006 the prototype system will be released as a Sanskrit corpus tagging assistant. This application is entirely distributed as a Web service. The user on his client machine types in a sentence, calls remotely the parser, inspects the small number of surviving taggings, then may inspect each one in order to peruse the semantic analysis, presented as a pseudo-English paraphrase. Some non-determinism may remain — typically, a given segment may be lemmatized in several ways, either by homonymy, or by morphological ambiguity. Each path in the semantic dependency matrix is shown with its bonus-malus, and the user may select the one he prefers, yielding a completely disambiguated analysis which he may then store on his client machine, as an hypertext document indexing in the Sanskrit Heritage Dictionary (our structured lexical database). This service has no equivalent worldwide.

This Sanskrit platform was presented at the ATALA workshop on “Traitement automatique des langues anciennes”, on May 21st in Paris. It was the topic of an invited lecture at the 5th International Conference on Logical Aspects of Computational Linguistics (LACL 2005) in Bordeaux on April 28th.

6.7.2. A new applicative model for finite state machines

Participants: Gérard Huet, Benoît Razet.

This work builds on the Zen toolkit[37] for lexical processing designed by Gérard Huet and distributed as free software as an OCaml library. It investigates a notion of mixed automaton or *aum*, first presented in 2003 in Gérard Huet’s “Automata Mista” article [52]. This work is being pursued as a general model for the modular construction of finite state machines, possibly non-deterministic, and possibly transducing their input on an output tape, in a purely applicative inductive data type whose operations model constructions of regular relations.

This year a new generic layer was abstracted for compiling control for the *reactive engine,* implementing an original notion of *modular transducer*. The user provides a system of regular expression over phases, as well as specific aum recognizers for each phase. A meta-programming tool, implementing the Berry-Sethi algorithm for regular expression compiling, yields a sequential *dispatcher* tailored to the specific application, as a stand-alone ML module, linked as a plug-in to the generic Zen toolkit. This was the topic of the summer internship of Benoît Razet for his 2nd year Master project at University Paris 6 [44]. This work lead to the release of version 2 of the Zen computational linguistics toolkit, as a free software Pidgin ML library. A joint article on the design of modular tranducers has been submitted for publication [42].

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.

This year, Microsoft Corporation joined the Caml Consortium. The three other members are : Athys (a subsidiary of Dassault Systèmes), Dassault Aviation, and LexiFi. For a complete description of this structure, refer to http://caml.inria.fr/consortium/index.en.html. Michel Mauny acts as the INRIA representative in the scientific committee of the Caml Consortium.

7.2. ReFLect

We have a contract with Intel Corporation that supports Virgile Prevosto, Michel Mauny, and Damien Doligez's work on formally defining and implementing the ReFLect functional programming language designed at Intel. (See section 6.5.3.) Michel Mauny is the principal investigator for this contract.

8. Other Grants and Activities

8.1. National initiatives

The Cristal, Logical, Miró 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 6.4.1), with the long-term goals of incorporating rewriting into Focal and studying how to apply the Focal methodology to security-sensitive applications.

The Cristal 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 called Compcert and involves the Cristal and Marelle INRIA projects as well as CNAM (Cedric laboratory) and University Paris 7 (PPS laboratory). The theme of this action is the on-machine certification of compilers and the development of supporting tools for specification and proof of programs. Xavier Leroy is the principal investigator for this action.

8.2. European initiatives

The Cristal project is involved in the European Esprit working group 26142 named "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. This group is coordinated by the Ludwig-Maximilians University in Munich and comprises both European academic teams and industrial research labs. The yearly meetings of this working group allow us to keep good connections with the European semantics community. Didier Rémy is the INRIA coordinator for this working group.

The Cristal 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.).

9. Dissemination

9.1. Interaction with the scientific community

9.1.1. Learned societies

Gérard Huet is Member of the French Academy of Sciences.

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

9.1.2. Prizes

Alain Frisch received the 2005 SPECIF prize (the *Société des Personnels Enseignants et Chercheurs en Informatique de France*) for his PhD work on XML processing languages, done at ENS under the supervision of Giuseppe Castagna.

9.1.3. Collective responsibilities within INRIA

Xavier Leroy was chairman of the *Section locale d'Auditions* (local hiring committee) for the INRIA Rocquencourt CR2 hiring competition (56 candidates for 2 positions).

Michel Mauny was member of the management team of the INRIA-Rocquencourt research unit as delegate for scientific affairs until july 2005. He was one of INRIA's representatives at Forum USA'05 (http://www.forumusa.org/?lang=en), held in Boston and San Francisco in april.

Pierre Weis is a member of the *Comité d'UR de Rocquencourt*. He is *membre titulaire élu* (elected member) of the *Comité Technique Paritaire* and the *Comité de Concertation*.

9.1.4. Collective responsibilities outside INRIA

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

Gérard Huet was invited to become member of the International Advisory Board of NII (National Institute of Informatics) in Tokyo, Japan. He participated to the first meeting of this board on June 2nd, and was subsequently offered to write a tribune in NII's journal[17].

Xavier Leroy is a member of the steering committee of the ACM Symposium on Principles of Programming Languages (POPL).

Michel Mauny was a member of the executive office of the *Réseau National de Recherche en Télécommunications* (RNRT) until november 2005. He is a member of the scientific board of the *Groupement de recherche Algorithmique, Langage et Programmation* (GDR ALP). He chairs the scientific board of the French-Moroccan cooperation program *Réseaux STIC*.

Pierre Weis is on the steering comittee of the JFLA conference.

9.1.5. 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 member of the editorial board of the Journal of Functional Programming.

Sandrine Blazy participated in the program committee of the Journées Francophones des Langages Applicatifs (JFLA 2006).

Alain Frisch was a member of the program committees of the International Symposium on Database Programming Languages (DBPL 2005) and the workshop on Programming Language Technologies for XML (PLAN-X 2006).

Xavier Leroy co-chaired the program committee for the ML Workshop 2005. He also participated in the program committees for the Virtual Execution Environments conference (VEE 2005) and the MetaOCaml Workshop 2005.

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).

9.1.6. PhD and habilitation juries

Xavier Leroy was reviewer (*rapporteur*) for the PhD theses of Antoine Galland (University Paris 6, july 2005) and David Pichardie (University Rennes 1, december 2005). He participated in the PhD jury of Xavier Rival (École Polytechnique, october 2005).

Michel Mauny was reviewer (*rapporteur*) for the PhD thesis of Éric Moretti (University Paris 11, february 2005).

Didier Rémy was a member of the PhD jury of his student Daniel Bonniot (École des Mines, november 2005).

9.1.7. The Caml user community

Maxence Guesdon maintains the Caml Humps (http://caml.inria.fr/humps/), a comprehensive Web index of about 450 Caml libraries, tools and tutorials contributed by Caml users. This Web site contributes significantly to the visibility of the Caml language.

Maxence Guesdon developed (with Vincent Simonet) and deployed a new Web site for the Caml programming language (http://caml.inria.fr/index.en.html). As part of the new site, Maxence Guesdon installed a modern bug tracking system for Caml, based on the Mantis Bug Tracker, and developed tools to automatically import all the bug reports from the previous system.

9.2. Teaching

9.2.1. Supervision of PhDs and internships

Alain Frisch co-supervises the PhD thesis of Kim Nguyen (ENS), together with Giuseppe Castagna and Véronique Benzaken.

Sandrine Blazy supervised the Master's internship of Agnès Gonnet (6 months) in cooperation with the BFD company. Sandrine Blazy and Xavier Leroy supervised the Master's internship of Zaynah Dargaye (5 months). Since october 2005, Zaynah Dargaye pursues a PhD under the supervision of Xavier Leroy.

In cooperation with Emmanuel Chailloux (PPS, univ Paris 6), Michel Mauny supervised the Master's internship of Grégoire Henry (6 months). Grégoire Henry pursues a PhD, still under Michel Mauny's co-supervision.

François Pottier supervises the PhD studies of Nadji Gauthier and Yann Régis-Gianas.

Didier Rémy supervises the PhD theses of Daniel Bonniot and Boris Yakobowski. Daniel Bonniot defended in november 2005 [11].

9.2.2. Graduate courses

The Cristal 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. The second year of this master's program succeeds the DEA *Programmation* and the DEA *Algorithmique* previously organized by these institutions. 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*.

In the context of the second year of the MPRI, Gérard Huet taught a 12-hour course entitled *Structures Informatiques et Logiques pour la Modélisation Linguistique* (logic and computational linguistics). François Pottier taught a 15-hour course entitled *Langages de programmation: sémantique et typage* (type systems and operational semantics). Alain Frisch taught a 3-hour mini-course on exact type-checking of tree transformations. Xavier Leroy taught a remedial course (*cours de remise à niveau*) on functional programming, operational semantics and type systems (7.5 hours).

In january 2005, Xavier Leroy gave a 7-hour lecture on the language-based approach to computer security at the TCS Excellence in Computer Week, a winter school organized in Pune (India) by Tata Consultancy Services and attended by about 50 Indian graduate students and lecturers.

In january 2005, François Pottier taught a 5 hour course entitled *Type-based information flow analyses* at the CIMPA School on Security of Computer Systems and Networks in Bangalore (India).

Xavier Leroy gave a 6-hour lecture on abstract machines and compilation of functional languages at the “École Jeune Chercheurs en Programmation” (Dinard, june 2005), a summer school attended by about 30 first-year PhD students.

In september 2005, François Pottier taught a 5 hour course entitled *A modern eye on ML type inference: old techniques and recent developments* at the APPSEM Summer School in Frauenchiemsee, Germany.

9.2.3. Undergraduate courses

Richard Bonichon is teaching assistant at University Paris 6 for the course “Advanced compilation” (M1, 40h) and the programming project “Tarski’s world in OCaml” (L2, 14h).

Michel Mauny gave two courses to engineering students, one on functional programming at ISIA (20 hours), the other on the Unix system at ETGL (10 days). Since august 2005, Michel Mauny is professor (on secondment) at ENSTA.

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

Didier Rémy is a part-time professor at École Polytechnique. This year, he taught a course on “Operating Systems: principles and programming” to third year students. Maxence Guesdon was teaching assistant for this course.

9.3. Participation in conferences and seminars

9.3.1. Participation in conferences

Alain Frisch and Yann Régis-Gianas attended the Symposium on Principles of Programming Languages (POPL) (Long Beach, California, USA, January 2005). At the co-located Programming Language Technologies for XML (PLAN-X) workshop, Alain Frisch presented a demonstration of CDuce.

Sandrine Blazy and Pierre Weis attended the Journées Francophones des Langages Applicatifs (Obernay, France, march 2005).

Sandrine Blazy, Xavier Leroy and François Pottier attended the European Joint Conferences on Theory and Practice of Software (ETAPS) (Edinburgh, Scotland, april 2005). Xavier Leroy was invited speaker at the satellite Workshop on Bytecode Semantics, Verification, Analysis and Transformation; he lectured on a unified framework for bytecode verification algorithms.

Gérard Huet was invited to give a talk at the International Conference on Rewriting Theory and Applications (RTA’05) (Nara, Japan, april 2005). He talked on “Rewriting before RTA”.

Didier Rémy participated in the International Conference on Typed Lambda Calculi and Applications (TLCA’05) (Nara, Japan, april 2005). He presented his work on subtyping recursive types modulo associative commutative products co-authored with François Pottier and Roberto Di Cosmo [26].

Xavier Leroy gave an invited lecture entitled “From Krivine’s machine to the Caml implementations” at the Workshop on the Krivine and ZINC Abstract Machines (KAZAM) (Birmingham, UK, may 2005).

Sandrine Blazy attended the International Conference on Theorem Proving in Higher Order Logics (TPHOL) (Oxford, UK, august 2005).

Richard Bonichon attended the TABLEAUX conference (Koblenz, Germany, august 2005).

Nadji Gauthier, Yann Régis-Gianas and Boris Yakobowski attended the APPSEM II Summer School on Applied Semantics, and the APPSEM II Workshop (Frauenchiemsee, Germany, september 2005). Boris Yakobowski presented his work with Didier Rémy on extending ML with impredicative second-order types at the workshop [35].

Alain Frisch, Xavier Leroy, Michel Mauny, François Pottier, Virgile Prevosto, Yann Régis-Gianas, Didier Rémy and Boris Yakobowski attended the ACM SIGPLAN International Conference on Functional Programming (ICFP) (Tallinn, Estonia, september 2005) and some of the co-located workshops: Workshop on ML, Haskell Workshop, Commercial Users of Functional Programming. François Pottier gave an invited talk at ICFP titled “From ML type inference to stratified type inference”. He also presented his work on Cαml at the ML Workshop [32]. Didier Rémy presented his work on partial type inference in the predicative fragment of System F [36] at ICFP. Yann Régis-Gianas presented his work with François Pottier on well-typed LR parsers at the ML Workshop [34].

Xavier Leroy participated in the meeting of IFIP Working Group 2.8 on Functional Programming (Kalvi, Estonia, october 2005). He presented his ongoing work on coinductive big-step semantics [30].

Sandrine Blazy attended the International Conference on Formal Engineering Methods (ICFEM) (Manchester, England, november 2005) where she presented a paper on the formalisation of a memory model [23].

9.3.2. Invitations and participation in seminars

Sandrine Blazy and Xavier Leroy presented their ongoing work on formal semantics and precompilation of a subset of the C language at the Logical-Proval seminar of the PRCI laboratory (Orsay, november 2005).

Richard Bonichon gave a talk at the Focal PPF seminar about the ongoing implementation of deduction modulo using CiME (Paris 6, december 2005).

Alain Frisch attended the Dagstuhl Seminar "Foundations of Semistructured Data" (February 2005) and the Seminar on Web Programming (Paris 13, April 2005), where he presented his work on integrating OCaml and CDuce type systems.

Alain Frisch and Xavier Leroy were invited to the Links Meeting (Edinburgh, April 2005) dedicated to new trends in programming of Web applications. Alain Frisch presented recent advances on XML types in the programming language community. Xavier Leroy presented speculations on future trends in programming language research.

In January, Gérard Huet participated in the annual TECS Excellence week in Pune, India, as member of the International Advisory Board of TRDDC (Tata Consultancy Services).

Gérard Huet was invited to deliver the Robin Milner lecture at University of Edinburgh in April. He talked on "Design of a Computational Linguistics Platform".

In January 2005, François Pottier gave a talk at ENS Lyon about an application of generalized algebraic data types to typechecking LR parsers.

François Pottier took part in and gave a talk at the Dagstuhl Seminar "Types for tools" in june 2005.

In november 2005, François Pottier and Yann Régis-Gianas visited LORIA at Frédéric Blanqui's invitation. Each of them gave a talk.

Virgile Prevosto gave a talk on Focal at the Plan Pluri-Formation "Développement des logiciels sûrs" (seminar for students and IT professionals from CNAM and University Paris 6) in december.

Didier Rémy visited Jacques Garrigue at the University of Kyoto in april 2005. He gave a talk on partial type inference based on predicative type-containment.

Pierre Weis participated in the Linux-Expo 2005 exhibition in Paris. He gave a talk about the design and construction of CD-ROMs with free material (in particular free software) in the public research field. He also participated in the Perl'05 workshop in Marseille Luminy, where he presented a "lightning" talk about functional programming and Caml.

9.4. Industrial relations

Sandrine Blazy is scientific collaborator at BFD, a company that experiments with Coq and OCaml to formally specify a domain-specific language devoted to information systems for financial institutions.

Pierre Weis is scientific collaborator at LexiFi, a start-up company that use Caml to design and implement a domain-specific language devoted to formal description and pricing of financial contracts.

9.5. Other dissemination activities

Michel Mauny and Emmanuel Chailloux (PPS Laboratory, univ. Paris 6) wrote a chapter on Functional Programming for the "*Encyclopédie des systèmes d'information*" [12], to appear (Vuibert Éditions).

François Pottier and Didier Rémy contributed a chapter on the essence of ML type inference to a graduate textbook edited by Benjamin Pierce [19].

With help from Guillaume Rousse (project Atoll) and François Clément (project Estime), Pierre Weis prepared the 2005 release of the CD-ROM "Free Software available at INRIA".

10. Bibliography

Major publications by the team in recent years

[1] G. COUSINEAU, M. MAUNY. *The Functional Approach to Programming*, Cambridge University Press, 1998.

[2] J. GARRIGUE, D. RÉMY. *Extending ML with semi-explicit higher-order polymorphism*, in "Information & Computation", vol. 155, n° 1/2, 1999, p. 134–169.

[3] T. HIRSCHOWITZ, X. LEROY. *Mixin modules in a call-by-value setting*, in "Programming Languages and Systems – ESOP'2002", D. LE MÉTAYER (editor). , Lecture Notes in Computer Science, vol. 2305, Springer-Verlag, 2002, p. 6–20, http://pauillac.inria.fr/~xleroy/publi/mixins-cbv-esop2002.pdf.

[4] 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://pauillac.inria.fr/~remy/work/mlf/icfp.pdf.

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

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

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

[8] F. POTTIER, V. SIMONET. *Information Flow Inference for ML*, in "ACM Transactions on Programming Languages and Systems", vol. 25, n° 1, January 2003, p. 117–158, http://pauillac.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.

[10] P. WEIS, X. LEROY. *Le langage Caml*, second edition, Dunod, July 1999.

Doctoral dissertations and Habilitation theses

[11] D. BONNIOT. *Typage modulaire des multi-méthodes*, Ph. D. Thesis, École des Mines de Paris, November 2005.

Articles in refereed journals and book chapters

[12] E. CHAILLOUX, M. MAUNY. *Programmation fonctionnelle*, in "Encyclopédie des systèmes d'information", To appear, Éditions Vuibert, 2006.

[13] D. DELAHAYE, M. JAUME, V. PREVOSTO. *Coq, un outil pour l'enseignement*, in "Technique et Science Informatique", vol. 24, n° 9, 2005, p. 1139-1160.

[14] P. FLAJOLET, G. HUET. *Mathématiques et informatique*, in "Les mathématiques dans le monde contemporain", J.-C. YOCCOZ (editor). , Rapport sur la science et la technologie n°20, Académie des sciences, 2005.

[15] T. HIRSCHOWITZ, X. LEROY. *Mixin modules in a call-by-value setting*, in "ACM Transactions on Program-

ming Languages and Systems", vol. 27, nº 5, 2005, p. 857–881, http://pauillac.inria.fr/~xleroy/publi/mixins-cbv-toplas.pdf.

[16] G. HUET. *A Functional Toolkit for Morphological and Phonological Processing, Application to a Sanskrit Tagger*, in "J. Functional Programming", vol. 15, nº 4, 2005, p. 573–614, http://pauillac.inria.fr/~huet/PUBLIC/tagger.pdf.

[17] G. HUET. *Internet Challenges for Informatics Research*, in "Progress in Informatics", vol. 1, nº 2, 2005.

[18] F. POTTIER, N. GAUTHIER. *Polymorphic Typed Defunctionalization and Concretization*, in "Higher-Order and Symbolic Computation", To appear, May 2005, http://cristal.inria.fr/~fpottier/publis/fpottier-gauthier-hosc.pdf.

[19] 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.

[20] F. POTTIER, C. SKALKA, S. SMITH. *A Systematic Approach to Static Access Control*, in "ACM Transactions on Programming Languages and Systems", vol. 27, nº 2, March 2005, http://cristal.inria.fr/~fpottier/publis/fpottier-skalka-smith-toplas.ps.gz.

Publications in Conferences and Workshops

[21] S. ABITEBOUL, C. BRYCE, R. DI COSMO, K. R. DITTRICH, S. FERMIGIER, S. LAURIÈRE, F. LEPIED, X. LEROY, T. MILO, E. PANTO, R. POP, A. SAGI, Y. SHTOSSEL, F. VILLARD, B. VRDOLJAK. *EDOS: Environment for the Development and Distribution of Open Source Software*, in "First International Conference on Open Source Systems (OSS 2005)", 2005, http://oss2005.case.unibz.it/Papers/37.pdf.

[22] 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, To appear, Springer-Verlag, 2005, http://pauillac.inria.fr/~xleroy/publi/proofs_dataflow_optimizations.pdf.

[23] S. BLAZY, X. LEROY. *Formal verification of a memory model for C-like imperative languages*, in "International Conference on Formal Engineering Methods (ICFEM 2005)", Lecture Notes in Computer Science, vol. 3785, Springer-Verlag, 2005, p. 280–299, http://www.springerlink.com/openurl.asp?genre=article&id=doi:10.1007/11576280_20.

[24] G. CASTAGNA, D. COLAZZO, A. FRISCH. *Error Mining for Regular Expression Patterns*, in "9th Italian Conference on Theoretical Computer Science", October 2005, p. 160-172.

[25] G. CASTAGNA, A. FRISCH. *A Gentle Introduction to Semantic Subtyping*, in "International Conference on Principles and Practice of Declarative Programming", Abstract of invited lecture, ACM Press, July 2005, p. 198-199.

[26] R. DI COSMO, F. POTTIER, D. RÉMY. *Subtyping Recursive Types modulo Associative Commutative Products*, in "Seventh International Conference on Typed Lambda Calculi and Applications (TLCA'05), Nara, Japan", Lecture Notes in Computer Science, vol. 3461, Springer Verlag, April 2005, p. 179–193, http://pauillac.inria.fr/~fpottier/publis/dicosmo-pottier-remy-tlca05.pdf.

[27] A. FRISCH. *OCaml + XDuce*, in "Programming Language Technologies for XML (PLAN-X) 2006", Accepted for publication, to appear, January 2006.

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

[29] H. HOSOYA, A. FRISCH, G. CASTAGNA. *Parametric Polymorphism for XML*, in "32nd ACM symposium on Principles of Programming Languages", ACM Press, January 2005, p. 50-62, http://www.cduce.org/papers/polyx.ps.gz.

[30] X. LEROY. *Coinductive big-step operational semantics*, in "European Symposium on Programming (ESOP'06)", Accepted for publication, to appear, Springer-Verlag, 2006, http://pauillac.inria.fr/~xleroy/publi/coindsem.pdf.

[31] 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", Accepted for publication, to appear, ACM Press, 2006, p. 42–54, http://pauillac.inria.fr/~xleroy/publi/compiler-certif.pdf.

[32] F. POTTIER. *An overview of Cαml*, in "ACM Workshop on ML", Electronic Notes in Theoretical Computer Science, September 2005, p. 27–51, http://cristal.inria.fr/~fpottier/publis/fpottier-alphacaml.pdf.

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

[34] F. POTTIER, Y. RÉGIS-GIANAS. *Towards efficient, typed LR parsers*, in "ACM Workshop on ML", Electronic Notes in Theoretical Computer Science, September 2005, p. 149–173, http://cristal.inria.fr/~fpottier/publis/fpottier-regis-gianas-typed-lr.pdf.

[35] D. RÉMY, B. YAKOBOWSKI. *MLF with Graphs*, in "Proceedings 3rd APPSEM II Worshop", sept 2005.

[36] D. RÉMY. *Simple, partial type-inference for System F based on type-containment*, in "Proceedings of the tenth International Conference on Functional Programming", ACM Press, September 2005, http://pauillac.inria.fr/~remy/publications.html#Remy/fml.

Internal Reports

[37] G. HUET. *The Zen Computational Linguistics Toolkit, Version 2.0*, 2005, http://sanskrit.inria.fr/ZEN/.

[38] 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/.

[39] V. SIMONET, F. POTTIER. *Constraint-Based Type Inference for Guarded Algebraic Data Types*, Research Report, n° 5462, INRIA, January 2005, http://www.inria.fr/rrrt/rr-5462.html.

Miscellaneous

[40] A. W. APPEL, X. LEROY. *A List-Machine Benchmark for Mechanized Metatheory*, Draft, circulated on the poplmark mailing list, 2005.

[41] Z. DARGAYE. *Éléments de preuve pour un compilateur certifié de C*, Master's thesis, Université Paris 7, 2005.

[42] G. HUET, B. RAZET. *The Reactive Engine for Modular Transducers*, Submitted for publication, 2005.

[43] X. LEROY. *A locally nameless solution to the POPLmark challenge*, October 2005, http://pauillac.inria.fr/~xleroy/POPLmark/locall nameless/.

[44] B. RAZET. *Automates modulaires*, Master's thesis, Université Paris 7, 2005.

Bibliography in notes

[45] R. BONICHON. *TaMeD: A Tableau Method for Deduction Modulo*, in "Automated Reasoning: Second International Joint Conference, IJCAR 2004", Lecture Notes in Computer Science, vol. 3097, Springer-Verlag, 2004, p. 445-459.

[46] G. BRACHA. *The programming language Jigsaw: mixins, modularity and multiple inheritance*, Ph. D. Thesis, University of Utah, 1992.

[47] F. CLÉMENT, V. MARTIN, A. VODICKA, R. DI COSMO, P. WEIS. *Domain decomposition for flow simulation around a waste disposal site: direct computation versus code coupling using OCamlP3l*, in "International Conference on Supercomputing in Nuclear Applications (SNA'2003)", September 2003.

[48] D. DOLIGEZ, X. LEROY. *A concurrent, generational garbage collector for a multithreaded implementation of ML*, in "Proc. 20th symp. Principles of Programming Languages", ACM press, 1993, p. 113–123, http://pauillac.inria.fr/~xleroy/publi/concurrent-gc.ps.gz.

[49] C. FOURNET, L. MARANGET, C. LANEVE, D. RÉMY. *Inheritance in the join calculus*, in "Foundations of Software Technology and Theoretical Computer Science – FSTTCS 2000", Lecture Notes in Computer Science, vol. 1974, Springer-Verlag, 2000.

[50] J. FURUSE. *Extensional polymorphism: theory and applications*, Ph. D. Thesis, University Paris 7, December 2002.

[51] J. GARRIGUE. *Private rows: abstracting the unnamed*, Draft, 2005, http://www.math.nagoya-u.ac.jp/~garrigue/papers/privaterows.pdf.

[52] G. HUET. *Automata Mista*, in "Verification: Theory and Practice: Essays Dedicated to Zohar Manna on the Occasion of His 64th Birthday", N. DERSHOWITZ (editor). , Lecture Notes in Computer Science, vol. 2772, Springer-Verlag, 2004, p. 359–372, http://pauillac.inria.fr/~huet/PUBLIC/zohar.pdf.

[53] 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.

[54] X. LEROY. *A syntactic theory of type generativity and sharing*, in "Journal of Functional Programming", vol. 6, n^o 5, 1996, p. 667–698, http://pauillac.inria.fr/~xleroy/publi/syntactic-generativity.ps.gz.

[55] L. RIDEAU, B. P. SERPETTE. *Coq à la conquête des moulins*, in "Journées Francophones des Langages Applicatifs", INRIA, 2005.

[56] F. ROUAIX. *A Web navigator with applets in Caml*, in "Proceedings of the 5th International World Wide Web Conference, in Computer Networks and Telecommunications Networking", vol. 28, Elsevier, May 1996, p. 1365–1371.

[57] D. RÉMY. *Type Inference for Records in a Natural Extension of ML*, in "Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design", C. A. GUNTER, J. C. MITCHELL (editors). , MIT Press, 1993.

[58] D. RÉMY. *Programming Objects with ML-ART: An extension to ML with Abstract and Record Types*, in "Theoretical Aspects of Computer Software", M. HAGIYA, J. C. MITCHELL (editors). , Lecture Notes in Computer Science, vol. 789, Springer-Verlag, April 1994, p. 321–346.

[59] 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.