INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Project-Team Cristal

Type-safe programming, modularity and compilation

Rocquencourt

THEME 2A

Table of contents

1. Team **1**
2. Overall Objectives **1**
3. Scientific Foundations **1**
3.1. Type systems 1
3.1.1. The Hindley-Milner type system 2
3.1.2. Typing objects 2
3.1.3. Typing modules 2
3.1.4. First-class polymorphism and higher-order types 2
3.1.5. Typing and confidentiality 3
3.2. The Caml programming language 3
4. Application Domains **3**
4.1. Software reliability 3
4.2. Processing of complex structured data 4
4.3. Fast development 4
4.4. Programming secure applications 4
4.5. Interoperability 4
4.6. Web programming 4
4.7. Teaching programming 4
4.8. Computational linguistics 5
5. Software **5**
5.1. Advanced software 5
5.1.1. Caml Light 5
5.1.2. Objective Caml 5
5.1.3. Camlp4 5
5.1.4. Cameleon 5
5.1.5. ActiveDVI 6
5.1.6. SpamOracle 6
5.1.7. WhizzyTeX 6
5.2. Prototype software 6
5.2.1. Wallace 6
5.2.2. Flow Caml 6
5.2.3. GCaml 6
5.2.4. The Zen computational linguistics toolkit 7
5.2.5. Htmlc 7
6. New Results **7**
6.1. Type systems 7
6.1.1. Extending ML with second order types 7
6.1.2. Type inference with structural subtyping 8
6.1.3. Extending HM(X) with existential and universal data-types 8
6.2. Type isomorphisms 8
6.2.1. Type isomorphism in the presence of subtyping and recursive types 8
6.2.2. Type isomorphisms, normalization by evaluation, and partial evaluation 9
6.3. Modularity 9
6.3.1. Mixin modules 9
6.3.2. Recursive modules 9
6.3.3. Modular type checking of multi-methods 10
6.3.4. Records and modules 10

6.3.5. Algebraic concrete data types 10
6.4. Principled compilation 11
6.4.1. Polymorphic Typed Defunctionalization 11
6.4.2. Compilation of extended recursion 11
6.4.3. Certified compilation 11
6.5. Static analysis 12
6.5.1. Information Flow Analysis 12
6.6. The Objective Caml system, libraries and tools 12
6.6.1. The Objective Caml system 12
6.6.2. The Caml development environment 13
6.6.3. The OCaml-SOAP library 13
6.7. Extensions of Objective Caml 13
6.7.1. OCamlDuce: typed XML processing in OCaml 13
6.7.2. OcamlP3L: high level parallel functional programming and code coupling 14
6.7.3. MetaOCaml: multi-staged computations in OCaml 14
6.7.4. Data persistence 14
6.8. Computational linguistics 15
6.8.1. Lexical, phonological and morphological tools 15
6.8.2. Applicative representation of finite automata and transducers 15
6.8.3. Sanskrit computational linguistics 15
7. Contracts and Grants with Industry 16
7.1. The Caml Consortium 16
8. Other Grants and Activities 16
8.1. National initiatives 16
8.2. European initiatives 16
9. Dissemination 16
9.1. Interaction with the scientific community 16
9.1.1. Learned societies 16
9.1.2. Collective responsibilities within INRIA 16
9.1.3. Collective responsibilities outside INRIA 17
9.1.4. Editorial boards and program committees 17
9.1.5. PhD juries 17
9.1.6. The Caml user community 18
9.2. Teaching 18
9.2.1. Supervision of PhDs and internships 18
9.2.2. Graduate courses 18
9.2.3. Undergraduate courses 18
9.2.4. Continuing education 19
9.3. Participation in conferences and seminars 19
9.3.1. Participation in conferences 19
9.3.2. Invitations and participation in seminars 19
9.4. Industrial relations 19
9.5. Other dissemination activities 20
10. Bibliography 20

1. Team

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

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

Administrative assistant
Stéphanie Aubin [until March 2003]
Nelly Maloisel [since April 2003]

INRIA staff
Maxence Guesdon [Technical staff (IR), 20% Cristal, 80% Miriad]
Gérard Huet [Senior research scientist (DR)]
Michel Mauny [Senior research scientist (DR)]
François Pottier [Research scientist (CR)]
Bruno Verlyck [Technical staff (IR), 80% Cristal, 20% Miriad]
Pierre Weis [Senior research scientist (DR)]

Visiting staff
Roberto Di Cosmo [Professor, university Paris 7, until September 2003]
Basile Starynkevitch [Senior research staff, CEA, since September 2003]
Hanfei Wang [Associate professor, Wuhan University, November 2003–November 2004]

Ph.D. students
Daniel Bonniot [AMN, university Paris 7]
Nadji Gauthier [MENRT grant, university Paris 7, since October 2003]
Benjamin Grégoire [MENRT grant, university Paris 7, in common with Logical]
Tom Hirschowitz [MENRT grant, university Paris 7]
Didier Le Botlan [AMX, university Paris 7]
Vincent Simonet [ENS, university Paris 7]

Student interns
Rajeseharan Deepak [IIIT Hyderabad, May–July 2003]
Thomas Dufour [University Paris 7, May–September 2003]
Nadji Gauthier [University Paris 7, April–September 2003]
Prateek Gupta [IIT Kanpur, May–July 2003]
Julien Roussel [EPITA, October–December 2003]
Zheng Li [University Paris 7, February–September 2003]

2. 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 [56] 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 [57]. Rémy and Vouillon [8] 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 [5][6]. 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 [50].

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 [51] 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][37]. 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 [20].

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. (We are currently working on extending these mechanisms to the handling of semi-structured 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. Interoperability

Using languages such as OCaml in realistic or industrial applications often requires interaction with existing libraries or software components developed in other (more classical) programming languages. Symmetrically, one may need to program algorithmically challenging parts of some applications in OCaml, and other parts (GUI, communications, main program) in other languages. Interoperability between OCaml and other languages is therefore important. We address this need via mechanisms for direct linking with C, Fortran or Java code, and by attempts to develop OCaml bindings for standard architectures for software components (Corba, COM, .Net, SOAP).

4.6. 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 [55] 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. In particular, the relation between standard XML "types" (DTDs, Schemas) and more classical type-theoretic notions must be studied, in order to make XML transformation languages and general-purpose languages interact securely and efficiently.

4.7. 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.

4.8. 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 [project Moscova], 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/.

5.1.2. Objective Caml

Participants: Xavier Leroy, Damien Doligez [project Moscova], Jacques Garrigue [Kyoto University], Maxence Guesdon, Luc Maranget [project Moscova], Jérôme Vouillon [CNRS, université 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/.

5.1.3. Camlp4

Participants: Michel Mauny, Daniel de Rauglaudre [INRIA Futurs].

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/camlp4/.

5.1.4. Cameleon

Participant: Maxence Guesdon.

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://savannah.nongnu.org/projects/cameleon.

5.1.5. ActiveDVI

Participants: Pierre Weis, Jun Furuse [LexiFi], Didier Rémy, Roberto Di Cosmo.

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/.

5.1.6. SpamOracle

Participant: Xavier Leroy.

SpamOracle is an automatic e-mail classification tool that separates legitimate e-mail from unsolicited commercial e-mail (spam). It proceeds by Bayesian statistical analysis of word frequencies, based on a user-provided corpus of legitimate and spam messages.

Web site: http://cristal.inria.fr/~xleroy/software.html.

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://pauillac.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. Wallace

Participant: François Pottier.

Wallace is a generic library for handling subtyping constraints. It deals with constraint solving and simplification, and it is parameterized by the definition of a type algebra. Its objective is to serve as a plug-in component in the design of a constraint-based type-checker, regardless of the programming language being analyzed. Wallace was used at project Contraintes as part of a type-checker for Constraint Logic Programming.

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

5.2.2. Flow Caml

Participant: Vincent Simonet.

Flow Caml is a prototype implementation of an information flow analyzer for the Caml language. It extends Objective Caml with a type system tracing information flow. Its purpose is basically to enable realistic programs to be written and to automatically check that they obey some confidentiality or integrity policy.

Web site: http://cristal.inria.fr/~simonet/soft/flowcaml/.

5.2.3. GCaml

Participants: Jun Furuse [LexiFi], Pierre Weis.

GCaml is an experimental extension of Objective Caml with extensional polymorphism, i.e., type-indexed functions that dispatch at run-time on the types of their arguments. Based on Jun Furuse's PhD thesis [51].

Web site: http://cristal.inria.fr/~furuse/generics/.

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://cristal.inria.fr/~huet/SKT/.

Web site: http://cristal.inria.fr/~huet/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. Extending ML with second order types

Participants: Didier Le Botlan, Didier Rémy.

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 laborious task 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. Hence the user must provide all type annotations.

Didier Le Botlan and Didier Rémy studied a new type system, called MLF, that still enables type synthesis as in ML while retaining the expressiveness of System F. MLF generalizes a previous proposal by Jacques Garrigue (Kyoto University) and Didier Rémy, called Poly-ML [2], that is now part of the OCaml type system. In Poly-ML and OCaml, second-order types are embedded into first-order types by means of semi-explicit coercions. However, coercions draw a barrier between inferred types and explicit second-order types. This gap disappears in MLF because its type schemes subsume both ML type schemes with implicit instantiation, and System-F second-order types with inner polymorphism. The trick is to allow bound variable of type schemes to stand for another type scheme (providing for inner quantification) or to range over all instances of another type scheme (providing for ML-style type inference).

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. All expressions of ML can be typed in MLF without any type annotations. All expressions of System F can also be typed, but explicit type annotations are required for function parameters that are used in a polymorphic manner. The study of MLF is the topic of Didier Le Botlan's PhD dissertation, which includes unification and type inference algorithms, their correctness proofs, as well as a type soundness proof. A summary of these results has been presented at the *International Conference on Functional Programming (ICFP)* in Uppsala, Sweden in August 2003 [37]. A small prototype implementation has also been written (http://cristal.inria.fr/~lebotlan/). This prototype confirms that few

annotations are indeed necessary in practice. Moreover, inferred types remain relatively legible in simple cases, but they can also be cumbersome in certain cases.

6.1.2. Type inference with structural subtyping

Participants: François Pottier, Vincent Simonet.

Structural subtyping is a widely used form of subtyping, where two comparable types must have the same shape and may only differ in their atomic leaves (in particular, there is no lowest or greatest type). This form of subtyping naturally arises when extending a unification-based type system with atomic annotations taken from a partially ordered set in order to perform some static analysis (such as detection of uncaught exceptions, data flow or information flow analysis). Vincent Simonet designed and implemented a realistic and efficient constraint solving algorithm, for type inference in systems equipped with structural subtyping and polymorphism. He then achieved a faithful formalization of this algorithm [35], which was presented at the Asian Symposium on Programming Languages and Systems.

6.1.3. Extending HM(X) with existential and universal data-types

Participants: François Pottier, Vincent Simonet.

Vincent Simonet proposed an extension of the type system HM(X) with bounded existential and universal data-types. This work generalizes Odersky and Läufer's abstract types for ML, by allowing quantifications to be bounded by an arbitrary constraint. In Simonet's system, type inference can be reduced to solving constraints that involve restricted forms of universal quantification and implication. These constructs are not generally handled by existing constraint solvers for subtyping. Therefore, Vincent Simonet also proposed a realistic constraint solving algorithm for the case of structural subtyping, which handles these non-standard forms. This work has been presented at the International Conference on Functional Programming (ICFP 2003) [34].

6.2. Type isomorphisms

6.2.1. Type isomorphism in the presence of subtyping and recursive types

Participants: Roberto Di Cosmo, François Pottier, Didier Rémy, Julien Roussel.

Two types A and B are isomorphic if two functions f and g of types $A \to B$ and $B \to A$ exist such that both $f \circ g$ and $g \circ f$ are equivalent to the identity. Type isomorphisms capture insignificant differences in data representations or function definitions, such as the order of components and function parameters [47].

An important application of type isomorphisms is function retrieval from large libraries, using types as search keys and performing comparison modulo isomorphism, in order to remove the arbitrary aspects in the formulation of requests. Complete characterizations of type isomorphisms have long been established and used to implement retrieval tools in the case of simple types, ML, and second-order type systems. Recently, the case of recursive types has also been studied.

Our research has been focused on type isomorphisms between products in the presence of recursive types and subtyping. Taking subtyping into account is particularly interesting in the case of object-oriented languages, where a function of a given type can also be given any of its subtypes. In theory, confining our focus to products is a genuine restriction. However, most pertinent cases that can be found in practice are still covered. We developed an algorithm for type comparison in this context. We also tackled the problem of building adapters for the results of requests. A consequence of searching modulo type isomorphisms is that a returned result may not exactly match the request. For instance, parameters of the result may not appear in the same order as for the request. However, it is possible to adapt, often mechanically, the code found in the library to return a function (or a class) that matches exactly the request.

Our results are described in a draft paper [44]. They have also been used to design a prototype retrieval tool for Java classes. This prototype has been implemented in OCaml by Julien Roussel—a student intern from EPITA during the period of Oct–Dec 2003. Large scale experimentation with this prototype is yet to be carried out.

6.2.2. Type isomorphisms, normalization by evaluation, and partial evaluation

Participants: Roberto Di Cosmo, Vincent Balat [University of Genova], Marcelo Fiore [University of Cambridge], Thomas Dufour.

The study on isomorphisms of types in the presence of products, function and sum types, has led to a major result, published at LICS 2002 [49], namely that isomorphisms for such a rich type language are not finitely axiomatisable. As a fundamental tool to establish this result, we developed a precise characterisation of a canonical form of terms for the lambda calculus with sum and products. No canonical rewriting system was known for lambda calculus with strong sums, so we gave a direct, inductive definition of canonical forms, and proved that these forms were the right representative for the equivalence classes of lambda terms modulo the equational theory for strong sums. For this, we have devised a sophisticated proof, based on a complex categorical construction, with immediate applications: the standard TDPE (type directed partial evaluator) that explodes in the presence of sum types, can be modified by incorporating appropriate restrictions derived from the canonical forms, in such a way as to correctly and efficiently handle even the monstrous terms that come from the proof given in the LICS paper [49]. This result has been accepted for publication in POPL'04 [22].

During the summer, Thomas Dufour was an intern at INRIA, under the co-direction of Roberto Di Cosmo and Marcelo Fiore, and has worked on the problem of characterising isomorphisms for passive ML monomorphic data types (essentially, recursive types with sums and products, but no function space type constructor), and is now continuing his research as a PhD student under the same supervisors.

6.3. Modularity

6.3.1. Mixin modules

Participants: Tom Hirschowitz, Xavier Leroy, J. B. Wells [Heriot-Watt University].

Module systems provide support for modular programming at the level of the programming language, including abstraction constructs and static verifications. The objective is to reduce programming errors in modular programs while still allowing the construction of rich libraries of modules. The ML module system remains one of the most expressive. Nevertheless, this system is weak on at least two important points. Firstly, mutually recursive definitions cannot be split across separate modules, which hinders modularization in several cases. Secondly, once a module is defined, the language does not propose any mechanism for incrementally modifying it. Instead, one has to copy the code manually, and create a new module from scratch.

Mixin modules are a notion of modules that allow both cross-module recursion and modifiability. They have been extensively studied in the setting of call-by-name evaluation. Under a call-by-value evaluation regime, they tend to conflict with the usual static restrictions on recursive definitions. Moreover, the semantics of instantiation has to specify an order of evaluation, which involves a difficult design choice. In a technical report [39], Tom Hirschowitz, Xavier Leroy, and J. B. Wells presented a strongly-typed kernel language of call-by-value mixin modules, including improvements compared to previous proposals concerning side effects, anonymous definitions, and the practicality of the type system. The obtained language, called *MM*, is very expressive, but rather heavy, since its semantics have to perform computations over the graph of dependencies between module components. In another technical report [38], Tom Hirschowitz defined a simpler strongly-typed kernel language of call-by-value mixin modules, called Mix. Although less powerful than MM, this language appears to be more practical.

6.3.2. Recursive modules

Participant: Xavier Leroy.

Building on our work on mixin modules (section 6.3.1), Xavier Leroy designed and implemented in Objective Caml 3.07 an experimental extension of Objective Caml with mutually recursive module definitions. While not as general as a fully-fledged mixin module system, this extension provides a short-term answer to the practical need for cross-references between modules. The implementation of the new recursive module binding construct builds on our compilation scheme for extended recursion described in section 6.4.2. This work is still

preliminary, in that the detection of ill-founded recursions is performed partially at compile-time and partially at run-time. Future work includes the design and implementation of a fully static detection.

6.3.3. Modular type checking of multi-methods

Participants: Daniel Bonniot, Didier Rémy.

Multi-methods generalize conventional class-based object-oriented programming by detaching methods from objects: a multi-method is a function that performs (multiple) dynamic dispatch on the types of all its arguments.

Daniel Bonniot introduced earlier kinding constraints to enable parameterization of methods over groups of classes that have some common structure (represented by the same kind) but that are not necessarily in a subclass relationship [45]. A revised version, including an open-world view point, was published this year [13] and is the core of his PhD dissertation: *Extension and application of polymorphic constrained type systems*.

Daniel Bonniot has also proposed a type-safe design of super calls from more specific implementations of a multi-method to some of its existing more general implementations. All these extensions to type-checking multi-methods have been formalized in a unified framework based on polymorphic constrained type systems. The high-level language is then compiled in a type-safe manner into a monomorphic language akin to the Java bytecode.

These theoretical results have been transferred to the implementation of the Nice language (http://nice.sourceforge.net), which provides multi-methods and advanced high-level features while remaining compatible with Java libraries. Some recent improvements of Nice have been to enable multiple dispatch on (mono-)methods declared in Java packages and to extend the concept of dispatch to selection based on integer, boolean, string, and enumeration values.

6.3.4. Records and modules

Participant: François Pottier.

Programming languages in the ML family feature a *module* sub-language, which forms a distinct layer above the *core* language. A module is a set of value and type definitions. The module language also features functions from modules to modules (*functors*), which increase code reusability.

Despite extensive research, today's module languages remain complex and of limited flexibility. Therefore, François Pottier wishes to design a more powerful module language, by re-unifying it with the core language, in order to reduce the overall complexity of the design, and by equipping it with a more expressive type system. From this point of view, modules would become records, while functors would become functions. A type system featuring rows and higher-order polymorphism would be required to lend the language sufficient expressive power.

François Pottier continued his study on potential designs for the source language as well as on potential compilation schemes. The latter includes in particular selective monomorphization, which produces specialized copies of polymorphic functions in order to improve runtime efficiency.

6.3.5. Algebraic concrete data types

Participant: Pierre Weis.

Pierre Weis has proposed the notion of *algebraic concrete data types*, which is a middle ground between traditional ML concrete data types and algebraic abstract data type. Like concrete data types, algebraic concrete data types can be examined and destructured concretely using ML pattern matching. Like abstract data types, algebraic concrete data types cannot be constructed outside of their defining module, thus allowing the definition and enforcement of invariants at data construction time. Therefore, algebraic concrete data types combine the ability to enforce representation invariants with the convenience of pattern-matching. Moreover, they provide a solution to the irritating problem of completeness that is typical of algebraic data types, since the conceptor of an algebraic concrete type can drastically reduce the set of primitives to the strict minimum that guarantees the invariants.

This proposal for algebraic concrete data type has been added to the Objective Caml language, version 3.07, under the name *private types*.

6.4. Principled compilation

6.4.1. Polymorphic Typed Defunctionalization

Participants: Nadji Gauthier, François Pottier.

Defunctionalization is a program transformation that aims to turn a higher-order functional program into a first-order one, i.e., to eliminate the use of functions as first-class values. Its purpose is therefore identical to that of *closure conversion*, but differs however from the latter by storing a *tag*, instead of a code pointer, within every closure. Defunctionalization has been used both as a reasoning tool and as a compilation technique.

Defunctionalization is commonly defined and studied in the setting of a simply-typed λ-calculus, where it is shown that semantics and well-typedness are preserved. It has been observed that, in the setting of a polymorphic type system, such as ML or System F, defunctionalization is not type-preserving. François Pottier and Nadji Gauthier showed that extending System F with *guarded algebraic data types*, in the style of Xi, Chen, and Chen [58], allows recovering type preservation. This result, which allows adding defunctionalization to the toolbox of type-preserving compiler writers, will be presented at the conference POPL'04 [32].

6.4.2. Compilation of extended recursion

Participants: Tom Hirschowitz, Xavier Leroy, J. B. Wells [Heriot-Watt University].

In the OCaml compiler, mutually recursive definitions are implemented using the "in-place update" trick, introduced by Cousineau and Mauny in 1987. Compared to a classical method, based on the update of reference cells, it avoids one indirection at each recursive call. Compared to Appel's method for compiling mutually recursive functions used in the SML/NJ and Objective Caml compilers, it allows the definition of non-functional, mutually recursive values.

In a conference paper [26] and its companion research report [40], Tom Hirschowitz, Xavier Leroy, and J. B. Wells extend this compilation scheme to a richer set of mutually recursive definitions. Furthermore, they formalize the compilation scheme as a translation down to a low-level target language without recursive definitions. This target language is close to one of the intermediate languages used in the OCaml compiler, which accounts for its implementability. Finally, they prove the semantic correctness of the compilation scheme by a simulation argument.

6.4.3. Certified compilation

Participant: Xavier Leroy.

Formal methods are being increasingly 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 laborious and costly.

A more efficient solution to this problem is to apply formal methods to the compiler itself. We are currently exploring the feasibility of developing a *certified compiler*, i.e., a realistic compiler that comes with a machine-checked proof that the generated assembly code has the same semantics as the source code. This exploration takes place within the Coordinated Research Action (ARC) Concert, involving projects Cristal, Lemme, Mimosa, Miró, Oasis, and University of Évry.

As part of these preliminary explorations, Xavier Leroy formalized and proved correct representative parts of a compiler back-end, using the Coq proof assistant. His Coq development includes the definition and operational semantics of a register transfer language (RTL – a typical intermediate language used in compilers), the proof of a general framework for defining and solving dataflow equations, and the specification

and proof of semantic preservation for three classical optimizations based on dataflow analysis: constant propagation, register allocation by coloring of an interference graph, and elimination of unreachable or unused code. This development represents about 5000 lines of Coq and required 3 man-months.

The results of this experiment are globally encouraging. In particular, dataflow analyses are relatively easy to formalize and prove correct in the Coq proof assistant. However, the program transformations that exploit the results of these analyses are somewhat harder to prove correct.

6.5. Static analysis

6.5.1. Information Flow Analysis

Participants: François Pottier, Vincent Simonet.

François Pottier and Vincent Simonet have continued their work on information flow analysis in programming languages of the ML family. The main achievement for 2003 has been the public release of the Flow Caml system, which is an extension of the Objective Caml language with an implementation of the type system they previously designed [20]. In Flow Caml, types are annotated with security levels chosen in a user-definable lattice. Each annotation gives an approximation of the information conveyed by the expression that it describes. Since it has full type inference, the system verifies, without requiring source code annotations, that every information flow caused by the program is legal with respect to the security policy specified by the programmer. Vincent Simonet has also written comprehensive documentation for the system, including a tutorial [42].

Besides security analysis, Flow Caml is also interesting since it is one of the first realistic implementations of a programming language that features both subtyping and polymorphism, and that has a complete type inference algorithm. Hence, this work gave us the opportunity to implement an efficient library for solving subtyping constraints (see section 6.1.2), and to experiment it on real-sized programs.

6.6. The Objective Caml system, libraries and tools

6.6.1. The Objective Caml system

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

Objective Caml is our main implementation of the Caml language. It is described in section 5.1.2. This year, we released version 3.07 of the Objective Caml system, after an extensive testing period involving two "beta" pre-releases. The main novelties in this release are:

- The introduction of "private type" declarations, corresponding to the notion of algebraic concrete data types described in section 6.3.5.
- An experimental extension with recursive module definitions, as described in section 6.3.2.
- A relaxation of the so-called "value restriction", which allows more polymorphic types to be inferred for certain `let` bindings. This relaxation, designed and proved sound by Jacques Garrigue [52], allows type variables to be generalized in the type of an expansive expression, provided they occur only covariantly in the inferred type.
- A port of the native-code generator to the new AMD 64-bit architecture, known technically as x86-64 and commercially as the Opteron and Athlon64 processors.
- A mechanism to record the types inferred for each sub-expression during compilation. These types can then be browsed off-line either under the Emacs editor or by using the OCaml browser. This is useful to locate the source of type errors, and also for pedagogical purposes.
- The standard library module `Scanf` for formatted input was extended and its performances greatly improved.

6.6.2. The Caml development environment

Participants: Maxence Guesdon, Nadji Gauthier, Vincent Simonet.

Cameleon is an integrated development environment for Objective Caml. It is described in section 5.1.4. Maxence Guesdon is the main developer and maintainer of Cameleon, This year, he added a library to generate HTML pages called Toolhtml.

Nadji Gauthier contributed Sqml, a library to parse and print SQL select queries. In combination with the Dbforge generator of SQL/OCaml bindings, Sqml will allow the user to define SQL queries and verify them against the database schema at compile time.

Maxence Guesdon and Vincent Simonet developed MozCaml, a sidebar for the Mozilla web browser. This sidebar lets the user browse and search the OCaml documentation, the Caml Humps (collection of libraries), as well as news about Caml (new libraries and tools, current discussions, etc). The distribution of news items relies on the RSS format, a standard XML DTD to represent news. Maxence Guesdon developed a library called OCaml-RSS to parse and print RSS files from OCaml programs.

6.6.3. The OCaml-SOAP library

Participants: Michel Mauny, Prateek Gupta.

SOAP is a standardized framework for remote method invocation based on XML as the data representation and HTTP as the transport layer. It can be used for programming distributed applications, and also to support cross-language interoperability. OCaml-SOAP is a binding between OCaml and the SOAP framework, initially developed by Michel Mauny and Gaurav Chanda in 2002. This year, Prateek Gupta improved this initial implementation during a summer internship supervised by Michel Mauny. This second version is more complete and modular, and is now publicly released (http://caml.inria.fr/ocaml-soap/).

6.7. Extensions of Objective Caml

6.7.1. OCamlDuce: typed XML processing in OCaml

Participants: Roberto Di Cosmo, Michel Mauny, Jérôme Vouillon [CNRS, Paris 7 university].

XML is now ubiquitous as a generic format for semi-structured data, both for data exchange (on the Web and other network protocols) and for data storage (in semi-structured databases). XML has created considerable interest in the type systems community (to interpret XML DTDs and Schemas as types) and also in the functional programming community (for XML transformations and XML database queries).

Our objective is to enrich OCaml with language and typing support for the manipulation and transformation of XML data, while enforcing DTD conformance via static typing. The starting point of this work is the XDuce functional language developed at U. Penn. [53], which expresses XML transformations via extended pattern matching on XML data, and maps DTDs to regular expression types that can be checked statically.

While XDuce is very strong at expressing XML transformations, it is not a general-purpose programming language: it provides little support for conventional data structures and algorithms. The goal of the OCamlDuce effort is, therefore, to combine the strengths of XDuce and OCaml: the program parts that transform XML data are written in a dialect of XDuce, statically type-checked with high precision using XDuce's regular expression types, and translated down to OCaml; these parts can then be linked with the remainder of the program, written in standard OCaml. Two-way communication between XDuce-defined functions and OCaml-defined functions is supported.

Challenges to be overcome include designing a suitable mapping between XDuce and OCaml types, and compiling XDuce pattern-matching down to efficient and compact OCaml code.

In March 2003, together with Giuseppe Castagna's group at ENS where the CDuce extension of XDuce is being developed, we invited Haruo Hosoya (Kyoto Univ.) for two weeks, during which we had productive meetings at INRIA, PPS and ENS. We now have an early prototype implementation of OCamlDuce that combines interpreted XDuce programs with compiled OCaml modules.

6.7.2. OcamlP3L: high level parallel functional programming and code coupling

Participants: Roberto Di Cosmo, Zheng Li, Pierre Weis, François Clément [project Estime], Jérôme Jaffré [project Estime], Vincent Martin [project Estime], Susanna Pelagatti [University of Pisa], Arnaud Vodycka [project Estime].

The OcamlP3L project aims at providing an industrial-strength, high-level parallel programming library for the Ocaml language, based on the P3L skeleton language developed by the researchers of the University of Pisa.

This year, we started a productive cooperation with the Estime project on the problem of coupling legacy numerical codes. They were looking for generic tools and infrastructure to facilitate these couplings, and found all they needed in OcamlP3L, and more besides, since they also obtained parallelisation of the code essentially for free. This cooperation has also deeply influenced the evolution of the system's design, which now incorporates a clear notion of stream processor, allows for a much more flexible coordination of parallel tasks than that available in the previous version, and incorporates a notion of computational weight of the user level tasks, fine grained enough to allow the programmer complete control of how to place his heavy computation on the nodes of a computational grid. The experiment has led to a first working version of the coupling code that has been successfully tested on the INRIA Rocquencourt Cluster, and presented at Supercomputing in Nuclear Applications 2003 [24]. A more detailed article on this coupling experiment was submitted to the special issue of Journal of Functional Programming on Functional Programming and Parallel Computing.

In association with Susanna Pelagatti and Zheng Li, Roberto Di Cosmo is currently investigating formal models of data distributions and the efficient compilation of a new powerful skeleton for data parallelism in OcamlP3L; preliminary results are presented in [15].

6.7.3. MetaOCaml: multi-staged computations in OCaml

Participants: Cristiano Calcagno [Imperial College, London], Walid Taha [Rice University], Liwen Huang [Yale University], Xavier Leroy.

Multi-stage programming languages provide a small set of constructs for the construction, combination, and execution of delayed computations. Multi-staged computation therefore accounts for macro generation and run-time code generation in a principled and semantically well-founded manner.

MetaOCaml is an extension of our Objective Caml language and implementation with multi-staged constructs: quotes, escapes, and execution of dynamically-generated expressions via compilation to the Objective Caml byte-code. It is developed by Cristiano Calcagno, Walid Taha and Liwen Huang; Xavier Leroy provided technical expertise with the Objective Caml implementation. MetaOCaml is one of the first implementations of multi-staged programming within a full-featured functional programming language, and is thus a good vehicle for experimentation with actual uses of multi-staged programming. MetaOCaml is described in an article that was presented at the Generative Programming and Component Engineering conference [23].

6.7.4. Data persistence

Participant: Basile Starynkevitch.

Basile Starynkevitch, previously research engineer at Commissariat à l'Énergie Atomique, DRT/DTSI Saclay, joined the Cristal project in september 2003 for a sabbatical as *spécialiste issu de l'industrie*. He works on adding data persistence to the OCaml language. Persistence allows data to persist between runs of a program, with greater transparency and integration into the language than classical approaches based on explicit database programming. He developed a prototype persistence library called PERSIL. It provides a simple persistence mechanism, with transactions, built on top of existing mechanisms, such as segmented files or relational database systems like MySQL. Further work includes the development of more orthogonal persistence mechanisms, perhaps by extending appropriately the OCaml language and compiler.

6.8. Computational linguistics

6.8.1. Lexical, phonological and morphological tools

Participant: Gérard Huet.

Gérard Huet extracted a generic toolkit from his Sanskrit modelling platform, enabling the construction of lexicons, the computation of morphological derivatives and flexed forms, and the segmentation analysis of phonetic streams modulo euphony. This little library of finite state automata and transducers, called Zen for its simplicity, was implemented in an applicative kernel of Objective Caml, called Pidgin ML. A *literate programming* style of documentation, using Jean-Christophe Filliâtre's program annotation tool Ocamlweb, is available. The Zen toolkit is distributed as free software (under the GPL licence) in the Objective Caml Hump site. This development forms a significant symbolic manipulation software package within pure functional programming that demonstrates the feasibility of developing symbolic applications having good time and space performance, within a purely applicative methodology.

This platform is currently being used outside the Cristal team. For instance, a lexicon of French flexed forms has been implemented by Nicolas Barth and Sylvain Pogodalla, in the Calligrammes project at Loria.

The algorithmic principles of the Zen library, based on the linear contexts data structure ('zippers') and on the sharing functor (associative memory server), were presented as an invited lecture at the symposium Practical Aspects of Declarative Languages (PADL), New Orleans, Jan. 2003 [30]. An extended version was written as a chapter of the book "Thirty Five Years of Automating Mathematics", edited in honor of N. de Bruijn [17].

6.8.2. Applicative representation of finite automata and transducers

Participant: Gérard Huet.

The Zen library algorithms were abstracted as a uniform applicative representation of finite automata and transducers. This structure was presented at the Festschrift in Honor of Zohar Manna for his 64th anniversary at Taormina (Sicily), and published in its proceedings [27].

6.8.3. Sanskrit computational linguistics

Participants: Gérard Huet, Rajeseharan Deepak.

G. Huet presented an original algorithm of segmentation and tagging for the Sanskrit language at the XIIth World Sanskrit Conference, Helsinki, Finland, Aug. 2003 [28][43]. The problem concerning the correct treatment of preverbs, which gave rise to a specific prediction technique ('phantom phonemes'), was presented at the International Conference on Natural Language Processing (ICON-2003) at Mysore, India, in December 2003 [29].

The Web site http://cristal.inria.fr/~huet/SKT, which presents various Sanskrit linguistics resources interactively, has an average of 1500 monthly visitors. In September 2003, a database of 200 000 flexed forms was delivered as a free resource in the XML format (given with a specific DTD). This database is used for research experiments by Pr. Stuart Shieber's team at Harvard University.

A collaboration is under way with Pr. Peter Scharf, from the Classics Department at Brown University, for interoperable use of such resources and exchange of annotated corpuses. Rajeseharan Deepak, a student from the International Information Institute of Technology (IIIT) in Hyderabad, spent his two months of summer internship in Rocquencourt to realise a module for manipulating within Ocaml an XML database conforming to a DTD. This was used notably to adapt to Ocaml the Whitney Dictionary of Roots, which was digitised as an XML document by collaborators of P. Scharf. This work is a preliminary step in the systematic construction of the conjugated forms of Sanskrit verbs.

Another collaboration began in September with Pr. Brendan Gillon, a linguist from Mc Gill University in Montreal, on the software structuring of a tree bank that he manually constructed from an annotated corpus of 500 sentences, issued from citations chosen in the Apte course book on Sanskrit syntax. This work should lead to the first version of a formal grammar describing Sanskrit sentences.

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. Currently, four companies are members of the Caml Consortium: Artisan Components, Athys, Dassault Aviation, and LexiFi. For a complete description of this structure, refer to http://caml.inria.fr/consortium/.

8. Other Grants and Activities

8.1. National initiatives

The Cristal and Moscova projects participate in an *Action Concertée Incitative* GRID named "PL4-CARAML", also involving the PPS lab (University Paris 7), the LIFO (University of Orléans) and the LACL (University Paris 12). The theme of this action is the design of extensions to functional programming languages in order to program efficient parallel computations, based on the BSP parallel programming model. Our participation to this action focuses on the OCaml-related aspects: design of OCaml extensions and support for OCaml. This ACI is managed by the LIFO.

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.

9. Dissemination

9.1. Interaction with the scientific community

9.1.1. Learned societies

Gérard Huet was elected Member of the French Academy of Sciences in november 2002.

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

9.1.2. Collective responsibilities within INRIA

Gérard Huet was chairman of the *Section locale d'Auditions* (local hiring committee) for the INRIA Futurs CR2 hiring competition (43 candidates for 5 positions). He spent the academic year 02-03 at LaBRI in Bordeaux, in the Signes team led by Christian Retoré, to help in the creation of a project-team of the Futurs UR, devoted to the syntax/semantics interface in natural language.

Michel Mauny was chairman of the *Section locale d'Auditions* (local hiring committee) for the INRIA Rocquencourt CR2 hiring competition. He is *membre de l'équipe de direction* (member of the management team) of the INRIA Rocquencourt research unit; *vice-président du Comité des Projets* (deputy chairman of the Projects Committee) since november 2003; and *membre suppléant nommé de la Commission d'Évaluation* (appointed deputy member of the Evaluation Committee) since december 2003. He was scientific organizer of the *rencontres INRIA-Industrie* 2004 (INRIA-Industry meeting), whose theme is Software Engineering.

Finally, Michel Mauny participated in INRIA working group on the position of secretaries in INRIA research groups.

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 *Comité de Concertation*, and *membre suppléant élu* (elected deputy member) of the *Commission d'Évaluation*.

Pierre Weis and Maxence Guesdon participate in an INRIA working group whose purpose is to computerize various administrative procedures, such as purchase orders, travel authorizations (*demandes de mission*), etc. Pierre Weis also developed a Web-based submission procedure for post-doc applications, in collaboration with Faranak Grange of the INRIA Foreign Relations office.

9.1.3. Collective responsibilities outside INRIA

Roberto Di Cosmo is a member of the *Commissions de spécialistes* (hiring committees) of University Paris 7 and University of La Réunion.

Gérard Huet was president of the review committee of the PPS laboratory at University Paris 7 (november 2003). He is Member of the Board of University Paris 7.

Xavier Leroy is a member of the steering committee of the International Conference on Functional Programming (ICFP) and of the Asian Association for Foundation of Software (AAFS). He is scientific advisor for the European IST Network DART (Dynamic Assembly, Reconfiguration and Type-checking).

Michel Mauny is a member of the Executive Office of the *Réseau National de Recherche en Télécommunications* (RNRT). He is a member of the Scientific Board of the *Groupement de recherche Algorithmique, Langage et Programmation* (GDR ALP), and of the Scientific Board of the French-Moroccan cooperation program *Réseaux STIC*.

9.1.4. Editorial boards and program committees

Roberto Di Cosmo was a member of the program committees of FOSSACS'04, LICS'04, WRS'03, WRS'04, and the First International Conference on Free Software Development and Usage (LACFREE'03).

Gérard Huet is a member of the selection committee of the ESSLLI 2004 summer school that will take place in Nancy in august 2004. He is a member of the program committee of the 11th Workshop on Logic, Language, Information and Computation (WoLLIC'2004), to occur at the University of Paris 12 in july 2004.

Xavier Leroy chaired the program committee of the 31st ACM Symposium on Principles of Programming Languages (POPL 2004).

Xavier Leroy and Didier Rémy are members of the editorial board of the Journal of Functional Programming.

Michel Mauny was a member of the program committee of the JFLA'04 conference.

Didier Rémy is on the program committee of the international conference on Programming Language Design and Implementation (PLDI) that will be held in Washington DC, USA in june 2004.

9.1.5. PhD juries

Roberto Di Cosmo chaired the PhD juries of Tom Hirschowitz and Benjamin Grégoire (University Paris 7, december 2003).

Gérard Huet was jury member at the doctorate defense of Virgile Prevosto (University Paris 6, september 2003) and reviewer (*rapporteur*) of the habilitation of Guy Perrier (University Nancy 1, november 2003).

Xavier Leroy was reviewer for Benoît Sonntag's PhD thesis (University Nancy 1, november 2003). He participated in the PhD juries of Simão Melo de Sousa (University of Nice, february 2003), Hervé Grall (École des Ponts et Chaussées, december 2003), Tom Hirschowitz (University Paris 7, december 2003), and Benjamin Grégoire (University Paris 7, december 2003).

Michel Mauny acted as a reviewer in the PhD committee of Eric Moretti (december 2003, University Paris 11), and as a reviewer in the habilitation of Emmanuel Chailloux (december 2003, University Paris 7).

9.1.6. The Caml user community

Maxence Guesdon maintains the Caml Humps (http://caml.inria.fr/humps/), a comprehensive Web index of about 250 Caml libraries, tools and tutorials contributed by Caml users. This Web site contributes significantly to the visibility of the Caml language.

9.2. Teaching

9.2.1. Supervision of PhDs and internships

Roberto Di Cosmo supervised the DEA internships of Thomas Dufour (5 months) and Zheng Li (8 months), as well as the 4-month research internship of Christophe Calves (student at ENS Cachan). Thomas Dufour and Zheng Li then started a PhD, still under Roberto Di Cosmo's supervision.

Xavier Leroy supervises the PhD work of Tom Hirschowitz. Jointly with Benjamin Werner (project Logical), he co-supervises the PhD work of Benjamin Grégoire.

Michel Mauny supervised Prateek Gupta's summer internship (master's student), and is supervising Hanfei Wang (associate professor, Wuhan University, visiting scholar at INRIA sponsored by the Chinese government from November 2003 to November 2004).

Gérard Huet supervised Rajeseharan Deepak's summer internship.

François Pottier supervises Vincent Simonet's PhD studies. He supervised Nadji Gauthier, a DEA intern, over a period of six months. Nadji Gauthier is now pursuing a PhD, still under François Pottier's supervision.

Didier Rémy has been supervising Didier Le Botlan and Daniel Bonniot's PhDs since September 2000 and Alexandre Frey's PhD since 2001.

François Pottier and Didier Rémy supervised the internship of Julien Roussel (engineering student at EPITA).

9.2.2. Graduate courses

The Cristal project-team is strongly involved in the DEA *Programmation: sémantique, preuves et langages* (Programming: semantics, proofs, languages), a graduate program co-organized by Universities Paris 6, Paris 7, Paris Sud, École Normale Supérieure Paris, École Normale Supérieure de Cachan, École Polytechnique and CNAM.

Roberto Di Cosmo is the director of the DEA Programmation, and teaches the Linear Logic course in this DEA (20 hours). François Pottier teaches the Types and Programming course in this DEA (20 hours). Xavier Leroy supervises the Programming Languages track of this DEA.

Gérard Huet taught a course on the syntax/semantics interface in natural language at DEA d'Informatique of University Bordeaux 1, jointly with C. Retoré.

Xavier Leroy gave a 5-hour lecture on module systems at the *École Jeune Chercheurs en Programmation*, a summer school attended by about 25 first-year PhD students.

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

9.2.3. Undergraduate courses

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.

Michel Mauny gave two courses to engineering students, one on Functional programming at ISIA (20 hours), the other on the Unix system at ETGL (42 hours).

Pierre Weis gave a course on denotational semantics to engineering students at ENSTA (*École Nationale des Techniques Avancées*), Paris (16 hours).

9.2.4. Continuing education

Pierre Weis gave a course on "Normal forms in rings", illustrated with Caml programs, at the yearly meeting of the *professeurs de mathématiques des classes préparatoires* (math professors of the preparatory class undergraduate curriculum).

9.3. Participation in conferences and seminars

9.3.1. Participation in conferences

Xavier Leroy, Michel Mauny and François Pottier attended the symposium Principles of Programming Languages and the satellite workshops Types in Language Design and Implementation and Foundations of Object-Oriented Languages (New Orleans, USA, january 2003).

Pierre Weis attended the *Journées Francophones des Langages Applicatifs* (Chamrousse, France, january 2003).

Xavier Leroy and Didier Rémy participated in the annual meeting of IFIP Working Group 2.8 Functional Programming (Sierre, Switzerland, january 2003). Xavier Leroy gave two talks, one on certified compilation, the other on Bayesian spam filtering. Didier Rémy presented his work with Didier Le Botlan on MLF [37].

Didier Le Botlan and Vincent Simonet attended the Workshop on Applied Semantics (APPSEM 2) (Nottingham, UK, march 2003). Didier Le Botlan presented his PhD work on MLF, and Vincent Simonet presented a short overview of the Flow Caml system [36].

Xavier Leroy was invited speaker at the ETAPS federation of conferences (Warsaw, Poland, april 2003). He talked on computer security from a programming language and static analysis perspective [31].

François Pottier attended the Logic in Computer Science symposium (LICS) (Ottawa, Canada, june 2003), where he presented a paper on a constraint-based vision of rows [33].

Tom Hirschowitz, Didier Le Botlan, Didier Rémy and Vincent Simonet attended the International Conference on Functional Programming (ICFP) (Uppsala, Sweden, august 2003). Tom Hirschowitz also attended the co-located conference on Principles and Practice of Declarative Programming (PPDP). Tom Hirschowitz presented his work on the compilation of recursion [26] at PPDP. Didier Le Botlan presented his work with Didier Rémy on MLF [37] at ICFP. Vincent Simonet presented his work about existential and universal data-types in HM(X) [34] at ICFP.

Vincent Simonet participated in the Dagstuhl Seminar on Language-Based Security, organized by A. Banerjee, H. Mantel, D. Naumann and A. Sabelfeld.

Vincent Simonet attended the Asian Symposium on Programming Languages and Systems (Beijing, China, november 2003), where he presented his work about type inference with structural subtyping [35].

9.3.2. Invitations and participation in seminars

Tom Hirschowitz spent one week at the University of Genova, visiting the programming languages group. He gave a seminar lecture on his work on mixin modules. Tom Hirschowitz spent one week at Heriot-Watt University (Edinburgh), in the context of his collaboration with J. B. Wells. He gave a talk to the ULTRA group on mixin modules and on the compilation of recursion.

Didier Le Botlan gave a talk about his PhD work to the Alice group at the University of Saarbrücken, Germany in April 2003.

Michel Mauny went to Rabat (Morocco) as a member of the Scientific Board of the French-Moroccan program *Réseaux STIC*, at the end of March 2003.

9.4. Industrial relations

Xavier Leroy is consultant for the Trusted Logic start-up company, one day per week, in the context of a *convention de concours scientifique*.

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

Roberto Di Cosmo is a recognized academic expert on free software. He gave a number of seminar talks and conferences on the subject of free software, software patents, intellectual property and the digital barrier, in France, Italy, Argentina and Peru, for widely different audiences: academics, IT professionals, and the general public. Slides from some of the conferences are available from http://www.dicosmo.org/TALKS. He published an article on intellectual property in *Upgrade* [14].

Michel Mauny and Maxence Guesdon participated in the Solutions Linux exhibition (Paris, feb 4–6, 2003). At the INRIA booth, they presented Objective Caml to potential users and generally promoted the language.

Maxence Guesdon took part in the design and development of the Web site of the *Association Libre Cours*, http://www.librecours.org/. This non-profit association collects freely available courses and lecture notes in French and ensures that they are widely available to students and teachers.

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 », number 1/2, volume 155, 1999, pages 134–169.

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

[4] X. LEROY. *A modular module system.* in « Journal of Functional Programming », number 3, volume 10, 2000, pages 269–303, http://pauillac.inria.fr/~xleroy/publi/modular-modules-jfp.ps.gz.

[5] F. POTTIER. *A Versatile Constraint-Based Type Inference System.* in « Nordic Journal of Computing », number 4, volume 7, 2000, pages 312–347.

[6] F. POTTIER. *Simplifying subtyping constraints: a theory.* in « Information & Computation », number 2, volume 170, 2001, pages 153–183.

[7] F. POTTIER, V. SIMONET. *Information Flow Inference for ML.* in « Proceedings of the 29th ACM Symposium on Principles of Programming Languages (POPL'02) », pages 319–330, January, 2002.

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

[9] D. RÉMY. *Using, Understanding, and Unraveling the OCaml Language.* G. BARTHE, editor, in « Applied Semantics. Advanced Lectures », series Lecture Notes in Computer Science, volume 2395, Springer-Verlag, 2002, pages 413–537.

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

Doctoral dissertations and "Habilitation" theses

[11] B. GRÉGOIRE. *Compilation de termes de preuves.* Ph. D. Thesis, University Paris 7, December, 2003.

[12] T. HIRSCHOWITZ. *Modules mixins, modules et récursion étendue en appel par valeur.* Ph. D. Thesis, University Paris 7, December, 2003.

Articles in referred journals and book chapters

[13] D. BONNIOT. *Using kinds to type partially-polymorphic methods.* in « Electronic Notes in Theoretical Computer Science », volume 75, 2003.

[14] R. DI COSMO. *Legal Tools to Protect Software: Choosing the Right One.* in « Upgrade », number 3, volume 4, June, 2003, pages 21–23, http://www.upgrade-cepis.org/issues/2003/3/up4-3DiCosmo.pdf.

[15] R. DI COSMO, S. PELAGATTI. *A calculus for dense array distributions.* in « Parallel Processing Letters », number 13, volume 3, 2003.

[16] J.-C. FILLIÂTRE, F. POTTIER. *Producing All Ideals of a Forest, Functionally.* in « Journal of Functional Programming », number 5, volume 13, September, 2003, pages 945–956, http://pauillac.inria.fr/~fpottier/publis/filliatre-pottier.ps.gz.

[17] G. HUET. *Linear Contexts and the Sharing Functor: Techniques for Symbolic Computation.* F. KAMAREDDINE, editor, in « Thirty Five Years of Automating Mathematics », Kluwer, 2003, http://pauillac.inria.fr/~huet/PUBLIC/DB.pdf.

[18] X. LEROY. *Java bytecode verification: algorithms and formalizations.* in « Journal of Automated Reasoning », number 3–4, volume 30, 2003, pages 235–269, http://pauillac.inria.fr/~xleroy/publi/bytecode-verification-JAR.pdf.

[19] F. POTTIER, D. RÉMY. *The essence of ML type inference.* B. C. PIERCE, editor, in « Advanced topics in Types and Programming Languages », MIT Press, 2004, To appear.

[20] F. POTTIER, V. SIMONET. *Information Flow Inference for ML.* in « ACM Transactions on Programming Languages and Systems », number 1, volume 25, January, 2003, pages 117–158, http://pauillac.inria.fr/~fpottier/publis/fpottier-simonet-toplas.ps.gz.

[21] R. DI COSMO, D. KESNER, E. POLONOVSKI. *Proof Nets and Explicit Substitutions.* in « Mathematical Structures in Computer Science », number 3, volume 13, 2003, pages 409–450.

Publications in Conferences and Workshops

[22] V. BALAT, R. DI COSMO, M. FIORE. *Extensional Normalisation and Type-Directed Partial Evaluation for Typed Lambda Calculus with Sums.* in « 31st ACM symposium on Principles of Programming Languages », ACM Press, January, 2004, To appear.

[23] C. CALCAGNO, W. TAHA, L. HUANG, X. LEROY. *Implementing Multi-stage Languages Using ASTs,*

Gensym, and Reflection. in « Generative Programming and Component Engineering (GPCE'03) », 2003, http://www.cs.rice.edu/~taha/publications/conference/gpce03b.pdf.

[24] 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.

[25] J. FURUSE. *Extensional polymorphism by flow graph dispatching.* in « Programming Languages and Systems, first Asian Symposium, APLAS 2003 », series Lecture Notes in Computer Science, number 2895, Springer-Verlag, A. OHORI, editor, pages 376–393, November, 2003, http://pauillac.inria.fr/~furuse/publications/flowgraph.ps.gz.

[26] T. HIRSCHOWITZ, X. LEROY, J. B. WELLS. *Compilation of extended recursion in call-by-value functional languages.* in « International Conference on Principles and Practice of Declarative Programming », ACM Press, pages 160–171, 2003, http://pauillac.inria.fr/~xleroy/publi/compil-recursion.pdf.

[27] G. HUET. *Automata Mista.* in « Festschrift in Honor of Zohar Manna for his 64th anniversary », series Lecture Notes in Computer Science, volume 2772, Springer-Verlag, N. DERSHOWITZ, editor, 2003, http://pauillac.inria.fr/~huet/PUBLIC/zohar.pdf.

[28] G. HUET. *Lexicon-directed Segmentation and Tagging of Sanskrit.* in « XIIth World Sanskrit Conference », 2003.

[29] G. HUET. *Towards Computational Processing of Sanskrit.* in « International Conference on Natural Language Processing (ICON) », 2003.

[30] G. HUET. *Zen and the Art of Symbolic Computing: Light and Fast Applicative Algorithms for Computational Linguistics.* in « Practical Aspects of Declarative Languages (PADL) symposium », 2003, http://pauillac.inria.fr/~huet/PUBLIC/padl.pdf.

[31] X. LEROY. *Computer Security from a Programming Language and Static Analysis Perspective.* in « Programming Languages and Systems: 12th European Symposium on Programming, ESOP 2003 », series Lecture Notes in Computer Science, volume 2618, Springer-Verlag, P. DEGANO, editor, pages 1–9, 2003, Extended abstract of invited lecture.

[32] F. POTTIER, N. GAUTHIER. *Polymorphic Typed Defunctionalization.* in « 31st ACM symposium on Principles of Programming Languages », ACM Press, January, 2004, http://pauillac.inria.fr/~fpottier/publis/fpottier-gauthier-popl04.ps.gz, To appear.

[33] F. POTTIER. *A Constraint-Based Presentation and Generalization of Rows.* in « Eighteenth Annual IEEE Symposium on Logic In Computer Science (LICS'03) », pages 331–340, Ottawa, Canada, June, 2003, http://pauillac.inria.fr/~fpottier/publis/fpottier-lics03.ps.gz.

[34] V. SIMONET. *An Extension of HM(X) with Bounded Existential and Universal Data-Types.* in « Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming (ICFP 2003) », ACM Press, pages 39–50, Uppsala, Sweden, August, 2003.

[35] V. SIMONET. *Type inference with structural subtyping: A faithful formalization of an efficient constraint solver.* in « Programming Languages and Systems, first Asian Symposium, APLAS 2003 », series Lecture Notes in Computer Science, number 2895, Springer-Verlag, A. OHORI, editor, pages 283–302, November, 2003.

[36] V. SIMONET. *Flow Caml in a Nutshell.* in « Proceedings of the first APPSEM-II workshop », G. HUTTON, editor, pages 152–165, Nottingham, United Kingdom, March, 2003.

[37] 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 », pages 27–38, August, 2003, http://pauillac.inria.fr/~remy/work/mlf/icfp.pdf.

Internal Reports

[38] T. HIRSCHOWITZ. *Rigid mixin modules.* Technical report, number RR2003-46, ENS Lyon, 2003, http://www.ens-lyon.fr/LIP/Pub/Rapports/RR/RR2003/RR2003-46.ps.gz.

[39] T. HIRSCHOWITZ, X. LEROY, J. B. WELLS. *A reduction semantics for call-by-value mixin modules.* Technical report, number 4682, INRIA, 2003, http://www.inria.fr/rrrt/rr-4682.html.

[40] T. HIRSCHOWITZ, X. LEROY, J. B. WELLS. *On the implementation of recursion in call-by-value functional languages.* Technical report, number 4728, INRIA, 2003, http://www.inria.fr/rrrt/rr-4728.html.

[41] X. LEROY, D. DOLIGEZ, J. GARRIGUE, D. RÉMY, J. VOUILLON. *The Objective Caml system, documentation and user's manual – release 3.07.* INRIA, September, 2003, http://caml.inria.fr/ocaml/htmlman/.

[42] V. SIMONET. *The Flow Caml System: documentation and user's manual.* Technical Report, number 0282, INRIA, July, 2003.

Miscellaneous

[43] G. HUET. *Transducers as Lexicon Morphisms, Phonemic Segmentation by Euphony Analysis, Application to a Sanskrit Tagger.* 2003, http://pauillac.inria.fr/~huet/PUBLIC/tagger.pdf.

[44] R. DI COSMO, F. POTTIER, D. RÉMY. *Subtyping Recursive Types modulo Associative Commutative Products.* 2003, http://pauillac.inria.fr/~remy/work/dicosmo-pottier-remy-03.ps.gz, Draft paper.

Bibliography in notes

[45] D. BONNIOT. *Using kinds to type partially polymorphic multi-methods.* in « Workshop on Types in Programming (TIP'02) », 2002.

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

[47] R. DI COSMO. *Isomorphisms of Types: from Lambda Calculus to Information Retrieval and Language Design.* Birkhauser, 1995.

[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, pages 113–123, 1993, http://pauillac.inria.fr/~xleroy/publi/concurrent-gc.ps.gz.

[49] M. FIORE, R. DI COSMO, V. BALAT. *Remarks on isomorphisms in typed lambda calculi with empty and sum types.* in « Symposium on Logic in Computer Science (LICS 2002) », IEEE, 2002.

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

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

[52] J. GARRIGUE. *Relaxing the value restriction.* August, 2003, submitted for publication, available from http://wwwfun.kurims.kyoto-u.ac.jp/~garrigue/papers/morepoly.pdf.

[53] H. HOSOYA, B. C. PIERCE. *XDuce: A Statically Typed XML Processing Language.* in « ACM Transactions on Internet Technology », number 2, volume 3, May, 2003, pages 117–148.

[54] X. LEROY. *A syntactic theory of type generativity and sharing.* in « Journal of Functional Programming », number 5, volume 6, 1996, pages 667–698, http://pauillac.inria.fr/~xleroy/publi/syntactic-generativity.ps.gz.

[55] 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 », volume 28, Elsevier, pages 1365–1371, May, 1996.

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

[57] D. RÉMY. *Programming Objects with ML-ART: An extension to ML with Abstract and Record Types.* in « Theoretical Aspects of Computer Software », series Lecture Notes in Computer Science, volume 789, Springer-Verlag, M. HAGIYA, J. C. MITCHELL, editors, pages 321–346, April, 1994.

[58] H. XI, C. CHEN, G. CHEN. *Guarded Recursive Datatype Constructors.* in « Proceedings of the 30th ACM SIGPLAN Symposium on Principles of Programming Languages », ACM Press, pages 224-235, 2003.