INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Project-Team Moscova

Mobility, Security, Concurrency, Verification and Analysis

Rocquencourt

THEME NUM

Activity Report

2006

Table of contents

1. Team

Head of project-team
Jean-Jacques Lévy [Senior Researcher (DR) Inria, HdR]

Vice-head of project-team
Luc Maranget [Research Associate (CR) Inria]

Administrative assistant
Sylvie Loubressac [Assistant (AI) Inria]

Research scientists
James Leifer [Research Associate (CR) Inria]
Francesco Zappa Nardelli [Research Associate (CR) Inria]

Post doctorant
Louis Mandel [postdoc INRIA]

Ph. D. students
Tomasz Blanc [INRIA grant, X Télécom, to 10/01/2006]
Pierre-Malo Deniélou [AMN, Paris 7]
Jean Krivine [MESR grant, Paris 6, to 10/01/2006]
Gilles Peskine [INRIA grant, Paris 7, to 12/31/2006]

Student intern
Sanchit Garg [IIT Bombay, from 05/15/2006 to 07/30/2006]

Visiting scientist
Andrew Appel [Princeton University, from 01/01/2005 to 07/31/2006]

2. Overall Objectives

2.1. Overall Objectives

The research in Moscova centers around the theory and practice of concurrent programming in the context of distributed and mobile systems. The ambitious long-term goal of this research is the programming of the web or, more generally, the programming of global computations on top of wide-area networks.

The scientific focus of the project-team is the design of programming languages and the analysis and conception of constructs for security. While there have been decades of work on concurrent programming, concurrent programming is still delicate. Moreover new problems arise from environments now present with the common use of the Internet, since distributed systems have become heavily extensible and reconfigurable.

The design of a good language for concurrency, distribution and mobility remains an open problem. On one hand, industrial languages such as Java and $C\#$ allow downloading of programs, but do not permit migrations of active programs. On the other hand, several prototype languages (Facile [44], Obliq, Nomadic Pict [39], Jocaml, etc) have been designed; experimental implementations have also been derived from formal models (Join-calculus, Ambients, Klaim, Acute, etc). None of these prototypes has yet the status of a real programming language.

A major obstacle to the wide deployment of these prototype languages is the security concerns raised by computing in open environments. The security research addressed in our project-team is targeted to programming languages. It is firstly concerned by type-safe marshaling for communicating data between different run-times of programming languages; it is also related to the definition of dynamic linking and rebinding in run-times; it deals with versioning of libraries in programming languages; it is finally connected to access control to libraries and the safe usage of functions in these libraries.

We are also interested by theoretical frameworks and the design of programming constructs for transaction-based systems, which are relevant in a distributed environment. A theory of reversible process calculus has been studied in that direction.

On the software side, we pursue the development of Jocaml with additional constructs for object-oriented programming. Although the development of Jocaml is rather slow, due to the departure of several implementers and to interests in other topics, Jocaml remains one of our main objective in the next years.

The Acute prototype, developed jointly at U. of Cambridge and at INRIA, demonstrates the feasibility of our ideas in type-safe marshaling, dynamic binding and versioning; it is based on Fresh Ocaml, and the integration of these ideas in/with Jocaml will be also studied in the next years. Several other prototypes also appeared in 2006: OTT – one tool for the working semanticist, Burfiks – a Bayesian web filter. We also maintain Hevea, Advix, and the pattern-matching part of Ocaml.

In 2006, Tomasz Blanc and Jean Krivine defended their PhD; Gilles Peskine is also very near to defend his PhD. Andrew Appel (Princeton University) finished his sabbatical year. Louis Mandel started a post-doctoral year in our project-team; Sanchit Garg (IIT Bombay) was intern with F. Zappa Nardelli. Tomasz Blanc is now back to the Corps des Telecoms; Jean Krivine is now in post-doc with W. Fontana (Harvard), Gilles Peskine with P. Sewell (Cambridge, UK).

In August 2006, J.-J. Lévy became director of the new Microsoft Research-INRIA Joint Centre, in Orsay. J. Leifer, P.-M. Deniélou and F. Zappa Nardelli are also now active in this centre, as members of the *Secure Distributed Computations and their Proofs*, headed by C. Fournet (Many members of the Joint Centre are former members of project-team Moscova).

Finally, we started at end of 2006 a new project PARSEC, funded by the ANR (*Agence Nationale de la Recherche*), together with MIMOSA, EVEREST, LANDE project-teams of INRIA and the team of Roberto Amadio at CNRS-PPS, U. of Paris 7. This project is coordinated by G. Boudol.

3. Scientific Foundations

3.1. Concurrency theory

Milner started the theory of concurrency in 1980 at Edinburgh. He proposed the calculus of communicating systems (CCS) as an algebra modeling interaction [36]. This theory was amongst the most important to present a compositional process language. Furthermore, it included a novel definition of operational equivalence, which has been the source of many articles, most of them quite subtle. In 1989, R. Milner, J. Parrow and D. Walker [37] introduced a new calculus, the *pi-calculus*, capable of handling reconfigurable systems. This theory has been refined by D. Sangiorgi (Edinburgh/INRIA Sophia/Bologna) and others. Many variants of the pi-calculus have been developed since 1989.

We developed a variant, called the Join-calculus [4], [5], a variant easier to implement in a distributed environment. Its purpose is to avoid the use of atomic broadcast to implement fair scheduling of processes. The Join-calculus allows concurrent and distributed programming, and simple communication between remote processes. It was designed with locations of processes and channels. It leads smoothly to the design and implementation of high-level languages which take into account low-level features such as the locations of objects.

The Join-calculus has higher-order channels as in the pi-calculus; channels names can be passed as values. However there are several restrictions: a channel name passed as argument cannot become a receiver; a receiver is permanent and has a single location, which allows one to identify channel names with their receivers. The loss of expressibility induced by these restrictions is compensated by joined receivers. A guard may wait on several receivers before triggering a new action. This is the way to achieve rendez-vous between processes. In fact, the notation of the Join-calculus is very near the natural description of distributed algorithms.

The second important feature of the Join-calculus is the concept of location. A location is a set of channels co-residing at the same place. The unit of migration is the location. Locations are structured as trees. When a location migrates, all of its sub-locations move too.

The Join-calculus, renamed Jocaml, has been fully integrated into Ocaml. Locations and channels are new features; they may be manipulated by or can handle any Ocaml values. Unfortunately the newer versions of Ocaml do not support them. We are still planning for both systems to converge.

3.2. Type systems

Types [38] are used in the theory of programming languages to guarantee (usually static) integrity of computations. Types are also used for static analysis of programs. The theory of types is used in Moscova to ensure safety properties about abstract values exchanged by two run-time environments; to define inheritance on concurrent objects in current extensions of Jocaml; to guarantee access control policies in Java- or $C\#$-like libraries.

3.3. Formal security

Formal properties for security in distributed systems started in the 90's with the BAN (Burrows, Abadi, Needham) logic paper. It became since a very active theory dealing with usual properties such as privacy, integrity, authentication, anonymity, repudiation, deniability, etc. This theory, which is not far from Concurrency theory, is relevant with the new activity of Moscova in the Microsoft Research-INRIA Joint Centre.

3.4. Labeled lambda-calculus

The theory of Church lambda-calculus is considered in our work about dynamic access control to library functions. More specifically, the theory of the confluent history-based calculus [35] defines a labeled lambda-calculus which is used both for access control in libraries and for the reversible pi-calculus.

4. Application Domains

4.1. Telecoms and Interfaces

Keywords: *distributed applications*, *security*, *telecommunications*, *verification*.

Distributed programming with mobility appears in the programming of the web and in autonomous mobile systems. It can be used for customization of user interfaces and for communications between several clients. Telecommunications is an other example application, with active networks, hot reconfigurations, and intelligent systems. For instance, France Telecom (Lannion) designs a system programmed in mobile Erlang.

5. Software

5.1. OTT: One true tool for the working semanticist

Participants: Peter Sewell [U. of Cambridge], Francesco Zappa Nardelli.

Ott is a tool for writing definitions of programming languages and calculi. It takes as input a definition of a language syntax and semantics, in a concise and readable ASCII notation that is close to what one would write in informal mathematics. It generates output:

1. a LaTeX source file that defines commands to build a typeset version of the definition;
2. a Coq version of the definition;
3. an Isabelle version of the definition; and
4. a HOL version of the definition.

Additionally, it can be run as a filter, taking a LaTeX/Coq/Isabelle/HOL source file with embedded (symbolic) terms of the defined language, parsing them and replacing them by typeset terms.

Our main goal is to support work on large programming language definitions, where the scale makes it hard to keep a definition internally consistent, and to keep a tight correspondence between a definition and implementations. We also wish to ease rapid prototyping work with smaller calculi, and to make it easier to exchange definitions and definition fragments between groups.

Most simply, the tool can be used to aid completely informal LaTeX mathematics. Here it permits the definition, and terms within proofs and exposition, to be written in a clear, editable, ASCII notation, without LaTeX noise. It generates good-quality typeset output. By parsing (and so sort-checking) this input, it quickly catches a range of simple errors, e.g. inconsistent use of judgment forms or metavariable naming conventions.

That same input can be used to generate formal definitions, for Isabelle, Coq, and HOL. It should thereby enable a smooth transition between use of informal and formal mathematics. Additionally, the tool can automatically generate definitions of functions for free variables, single and multiple substitutions, sub-grammar checks (e.g. for value sub-grammars), and binding auxiliary functions.

Our focus here is on the problem of writing and editing language definitions, not (directly) on aiding mechanized proof of meta-theory. If one is involved in hard proofs about a relatively stable small calculus then it will aid only a small part of the work (and one might choose instead to work just within a single proof assistant), but for larger languages the definition is a more substantial problem — so much so that only a handful of full-scale languages have been given complete definitions. We aim to make this more commonplace, less of a heroic task.

The current pre-alpha version of Ott is about 14000 lines of OCaml. It will be available from http://moscova.inria.fr/ zappa/software/ott.

5.2. Burfiks: Bayesian web filtering

Participants: Sanchit Garg, Francesco Zappa Nardelli.

This project studies and implement a statistical approach to improve the quality of the results returned by web search engines, and more generally, to sort the links of a web page according to their interest.

An inevitable consequence of the ambiguity of the natural language used to express search queries is that interesting results of searches on the web are often hidden among many unrelated matchings. The use of endogenous webs (among the documents that match a query, the most relevant is the one which is pointed by the greatest number of them) as done by Google turned out to be a promising solution.

However, as far as we know, an interesting and simple approach has not been considered yet: search by Bayesian filtering. Bayesian filters have been made popular by Paul Graham to identify automatically Spam messages. They rely on an elementary theorem of probability theory known as “Bayes formula”. Intuitively, by examining old messages that have been first classified by the user into “Spam” and “not-Spam”, they build a probability distribution that has good properties to predict if an incoming message will be classified by the user as Spam or not.

This project aims at applying the Bayesian approach to refine the results returned by web searches, and, more generally sorting the links found in a web page according to their interest.

We developed a software layer integrated with the Firefox browser that collects the links found in a web page and uses hints of the user to classify them, along the lines of anti-Spam Bayesian filters. Preliminary experience revealed that this tool is extremely useful to navigate quickly across pages that contains many links, as it identifies quite reliably the most interesting ones in real time.

A public release of the software is expected the next month, from http://moscova.inria.fr/ zappa/software/burfiks.

We also project to extend the software with more refined statistical models, such as indexes of co-occurrence.

In collaboration with Roberto Di Cosmo (PPS, U. Paris 7).

5.3. Acute

Participants: Pierre Habouzit, James Leifer, Francesco Zappa Nardelli [INRIA], Mair Allen-Williams, Peter Sewell, Viktor Vafeiadis, Keith Wansbrough [U. of Cambridge].

Acute is a test implementation of our current work on high-level programming languages for distributed computation. For motivation and a description of the language, see section 6.1.

The main priority for the implementation is to be rather close to the semantics, to make it easy to change as the definition changed, and easy to have reasonable confidence that the two agree, while being efficient enough to run moderate examples. An automated testing framework helps ensure the two are in sync.

The runtime is essentially an interpreter over the abstract syntax, finding redexes and performing reduction steps as in the semantics. For efficiency it uses closures and represents terms as pairs of an explicit evaluation context and the enclosed term to avoid having to re-traverse the whole term when finding redexes. Marshalled values are represented simply by a pretty-print of their abstract syntax. Numeric hashes use a hash function applied to a pretty-print of their body; it is thus important for this pretty-print to be canonical, choosing bound identifiers appropriately. Acute threads are reduced in turn, round-robin. A pool of OS threads is maintained for making blocking system calls. A genlib tool makes it easy to import (restricted versions of) Ocaml libraries, taking Ocaml .mli interface files and generating embeddings and projections between the Ocaml and internal Acute representations.

On top of Acute, we have written libraries for TCP connection management and string messaging, local and distributed channels, remote function invocation, as well as two bigger libraries that implement the mobility models of two process languages, namely Nomadic Pict and Mobile Ambients. Our experience suggests that our lightweight extension to ML suffices to enable sophisticated distributed infrastructure to be programmed as simple libraries.

Our prototype consists of about 25 000 lines of code and is written in FreshOcaml, a variant of INRIA's Ocaml with support for automatic alpha conversion. Acute is distributed on-line in source form from http://www.cl.cam.ac.uk/users/pes20/acute/.

This software activity has been suspended in 2006. A paper is accepted for publication in JFP[17].

5.4. Caml/Jocaml

Participants: Louis Mandel, Luc Maranget.

In 2005, Luc Maranget has completed his prototype version of the new JoCaml system, the implementation of the join-calculus integrated in the Objective Caml language (developed by the GALLIUM project-team). With respect to the previous join-language prototype, the most salient feature of the new prototype is a better integration into Objective Caml. We aim binary compatibility with Objective Caml, and plan releases that follow Objective Caml releases. This goal could not be met last year by Luc Maranget in 2005 by lack of time and manpower.

In 2006 (late spring), Louis Mandel has been recruited on a post-doctoral position. Louis Mandel joined our project-team in November, his task it first to release JoCaml, and afterward to study further developments of JoCaml. For the release, Louis' main task is to write a comprehensive documentation, which should normally be ready at the time of the next release of Objective Caml, scheduled early 2007.

5.5. Pattern matching in Ocaml

Keywords: *ocaml*, *pattern matching*.

Participant: Luc Maranget.

Luc Maranget wrote a paper describing his work on the analysis of pattern matching in functional languages (ML, Haskell) [16]. After almost two years of refereeing, this paper has been accepted for publication in the *Journal of Functional Programming*.

This work examines the ML pattern-matching anomalies of useless clauses and non-exhaustive matches. Providing good diagnoses for these anomalies helps enormously programmers: the novices better understand the subtleties of ML pattern-matching, the experienced programmers better control the effect of program modifications.

We state the definition of these anomalies, building upon pattern matching semantics, and propose a simple algorithm to detect them. This algorithm has been integrated in the Objective Caml compiler by Luc Maranget. The same algorithm is also usable in non-strict languages such as Haskell. Or-patterns can be considered for both strict and non-strict languages.

This paper can be seen as a by-product of Luc Maranget's expertise in the field of ML pattern-matching — Luc Maranget is the author of the pattern-matching compiler in the Objective Caml system.

5.6. Hevea

Keywords: *html*, *latex*, *tex*, *web*.

Participant: Luc Maranget.

Hevea is a translator from LaTeX to HTML, written in Objective Caml. Hevea was first released in 1997 and is still maintained and developed by Luc Maranget. A continuous (although informal) collaboration around Hevea exists, including Philip H. Viton (Ohio State University) for Windows port and Ralf Treinen (ENS Cachan) for Debian developments. For the record, Hevea consists in about 20000 lines of Caml and about 5000 lines of packages sources written in "almost TeX" (the language understood by Hevea).

This year saw the release of Hevea 1.09. Main improvements over previous version 1.08 are:

- Many more LaTeX symbols are now available. Concretely, Hevea provides implementations of many LaTeX packages for symbols, such as *latexsym* or *amssymb*. Symbols commands, such as `\Join` ($\Join$) or `\doublecap` ($\Cap$), are translated to Unicode numerical entities, such as `&#X2445;` or `&#X22D2;`. This method complies with published standards and should enable proper display on modern browsers.
- Paragraphs are now expressed with `<P>` elements. By contrast, in previous versions, paragraphs were expressed with explicit line breaks (`<BR><BR>`). The new method yields more structured HTML. This is a preliminary step to XHTML output.

Hevea is distributed as free software from its own web site http://hevea.inria.fr. The audience of Hevea is difficult to estimate, given the well established practice of binary packaging by several independent Linux distributions. Luc Maranget has contacts on a regular basis with Debian packagers, but other Linux distributions such as Mandrake also feature Hevea. Important fact: Debian switched to Hevea as their major tool for producing HTML versions of software documentations for all Debian packages.

Given its speed, Hevea is particularly suited to translation of documents of important size. Some teachers use Hevea to publish their courses on the web (see http://hevea.inria.fr/examples/). Gilles Grégoire (IRSN/DPAM–Cadarache) used Hevea to produce the HTML version of the test report PHÉBUS-PF FPT2 — PHÉBUS-PF is a set of experiments performed by IRSN and CEA. The HTML version of the report amounts to 800 files for a total size of 6 megabytes. Luc Maranget provided Gilles Grégoire with some help and a few custom developments during winter 2006.

5.7. Active DVI

Keywords: *dvi*, *latex*, *tex*.

Participant: Francesco Zappa Nardelli.

`Advix` is a new implementation of ActiveDVI, a programmable viewer for DVI files developed by the Cristal project-team at INRIA Rocquencourt. It improves the original one by (among other things) providing faster rendering of postscript specials, and by being compatible with Windows and MacOs. A pre-alpha version is being tested (about 48000 lines of OCaml and C code).

In collaboration with Didier Remy (INRIA Rocquencourt).

6. New Results

6.1. High-level programming language design for distributed computation

Participants: Pierre Habouzit, James Leifer, Francesco Zappa Nardelli [INRIA], Mair Allen-Williams, Peter Sewell, Viktor Vafeiadis, Keith Wansbrough [U. of Cambridge].

This work addresses the design of distributed languages. Our focus is on the higher-order, typed, call-by-value programming of the ML tradition: we concentrate on what must be added to ML-like languages to support typed distributed programming. We explore the design space and define and implement a programming language, Acute (see section 5.3).

This builds on previous work done by James J. Leifer, Gilles Peskine (INRIA), Peter Sewell, Keith Wansbrough (U. of Cambridge), published in ICFP 2003. The present work extends the theory to contend with the challenge of integrating with an ML like programming language. This requires a synthesis of novel and existing features.

Type-safe marshaling Type-safe marshaling demands a notion of type identity that makes sense across multiple versions of differing programs. For concrete types this is conceptually straightforward, but with abstract types more care is necessary. We generate globally-meaningful type names either by hashing module definitions, taking their dependencies into account; freshly at compile-time; or freshly at run-time. The first two enable different builds or different programs to share abstract type names, by sharing their module source code or object code respectively; the last is needed to protect the invariants of modules with effect-full initialization.

Dynamic linking and rebinding Dynamic linking and rebinding to local resources in the setting of a language with an ML-like second-class module system raises many questions: of how to specify which resources should be shipped with a marshaled value and which dynamically rebound; what evaluation strategy to use; when rebinding takes effect; and what to rebind to. For Acute we make interim choices, reasonably simple and sufficient to bring out the typing and versioning issues involved in rebinding, which here is at the granularity of module identifiers. A running Acute program consists roughly of a sequence of module definitions (of ML structures), imports of modules with specified signatures, which may or may not be linked, and marks which indicate where rebinding can take effect; together with running processes and a shared store.

Global expression names Globally-meaningful expression-level names are needed for type-safe interaction, e.g. for communication channel names or RPC handles. They can also be constructed as hashes or created fresh at compile time or run time; we show how these support several important idioms. The polytypic support and swap operations of Shinwell, Pitts and Gabbay's FreshOcaml are included to support renaming of local names occurring inside of values during communication.

Versioning In a single-program development process one ensures the executable is built from a coherent set of versions of its modules by controlling static linking often by building from a single source tree. With dynamic linking and rebinding more support is required: we add versions and version constraints to modules and imports respectively. Allowing these to refer to module names gives flexibility over whether code consumers or producers have control.

Local concurrency Local concurrency is important for distributed programming. Acute provides a minimal level of support, with threads, mutexes and condition variables. Local messaging libraries can be coded up using these, though in a production implementation they might be built-in for performance. We also provide thunkification, allowing a collection of threads (and mutexes and condition variables) to be captured as a thunk that can then be Marshalled and communicated (or stored); this enables various constructs for mobility to be coded up.

We deal with the interplay among these features and the core, in particular with the subtle interplay between versions, modules, imports, and type identity, requiring additional structure in modules and imports. We develop a semantic definition that tracks abstraction boundaries, global names, and hashes throughout compilation and execution, but which still admits an efficient implementation strategy.

The definition is too large to make proofs of the properties feasible with the available resources and tools. To increase confidence in both semantics and implementation, therefore, our implementation can optionally type-check the entire configuration after each reduction step. Our strategy has been to synchronize the development of the formal specification of the Acute language with that of its implementation [...crossreflogicielsacute..]. As a result we have been able to quickly discovered and fixed errors in the type-system and in the run-time semantics.

The specification of Acute, together with a discussion of the design rationale, is available as an INRIA Research Report [28]. A paper describing our design principles decisions was published [24] in the International Conference on Functional Programming.

6.2. Language design for distributed transactions

Participants: James Leifer, Francesco Zappa Nardellil.

Ensuring the coherency that the global state of a concurrent system is often difficult: locking mechanisms are commonly used to prevent interference, but they are no panacea and their use require a constant attention. In distributed systems this task is even more complicated: failures of machines and of network connections, the impossibility of relying on distributed locks, and the complex mechanisms to modify the local state of a remote machine require the implementation of complex protocols to update the remote states and, in some cases, to backtrack these updates.

Several researchers proposed *software transactional memory* [42]: groups of memory operations can then be performed atomically without explicitly requiring locking mechanisms. However their works are limited to the model of threads interacting through memory, and do not consider the questions of external interaction through storage systems or databases or, more generally, other machines.

Our investigation aims at designing, implementing, and proving correct, an infrastructure that offers distributed software transactional memory. The API exports operations to modify the local store, an rpc layer to interact with remote machines, and a keyword `atomic` that ensures that all the memory accesses (both on the local and on the remote stores) are realized in a all-or-none fashion. The `abort` keyword has the effect of interrupting the execution of the current atomic, undoing all the changes done both on the local and on the remote store. Atomic section can be nested, thus offering an expressive language for distributed transactions.

Our methodology consists in developing both a formal high-level operational semantics and a reference implementation. The high-level semantics is fundamental to identify and to reason about our design choices, and, since it abstracts from the implementation details, is useful to analyze some complex behaviors that may result from subtle interactions between nodes. The reference implementation (currently an Ocaml library) is used as a test-bed to ensure that the high-level semantics can be implemented safely and efficiently. The interaction between these two levels is a fruitful source of hints on how to proceed. A formal proof of the soundness of the reference implementation with respect to the high-level semantics will complete our research.

This ongoing research project is done in collaboration with Peter Sewell (U. of Cambridge). In 2006, this line of research has been suspended (but not discontinued).

6.3. Sub-typing and abstraction safety in distributed languages

Participants: Pierre-Malo Deniélou, James Leifer.

In most programming languages, type abstraction is guaranteed by syntactic scoping in a single program, but is not preserved by marshaling during distributed communication. A solution is to generate hash types at compile time that consist of a fingerprint of the source code implementing the data type. These hash types can be tupled with a marshaled value and compared efficiently at unmarshall time to guarantee abstraction safety. In this work, we extend a core calculus of ML-like modules, functions, distributed communication, and hash types, to integrate structural sub-typing, user- declared sub-typing between abstract types, and bounded existential types. Our semantics makes two contributions: (1) the explicit tracking of the interaction between abstraction boundaries and sub-typing; (2) support for user-declared module upgrades with propagation of the resulting sub-hashing relation throughout the network during communication. We prove type preservation, progress, determinacy, and erasure for our system.

This work has been presented at [22].

6.4. Type-safe marshaling

Participants: James Leifer, Gilles Peskine.

Gilles Peskine followed up on technical aspects of type safety for separately compiled programs in the presence of abstract types. The earlier work dealt with simple abstract types with obvious dependency chains. Subsequent work expanded on the core idea of using hashes to represent abstract types in a larger language incorporating common module features such as nesting, partial sealing, and both generative and applicative functors.

The calculus described in [31] has such common features. However, only a type system is provided, with no run-time semantics hence no formal notion of safety. Abstract types cause some difficulty in defining a type-preserving run-time semantics as the equality between the abstract type and its implementation is sometimes visible and sometimes hidden. Hashes (or nonces in the generative case) annotating colored brackets, introduced to model dynamic type checks in a distributed setting, also help in providing a formal way to limit the scope of type equalities. A type-preserving semantics has been defined and is being proved for a language which gives a precise account of abstract types with the varying degrees of generativity found in diverse module calculi.

The new semantics has been achieved by modifying, and often simplifying, the existing calculus in several ways. The coexistence of generative functors and applicative functors lets the programmer decide whether to create abstract types each time the functor is applied (a good thing if the functor's code manipulates an exclusive resource such as a memory location or a hardware device) or to reuse the same abstract type when given equivalent arguments (a good thing if the functor implements a parametric data structure). Where earlier systems either provide only one behavior or require a notion of static impurity whose correct treatment by reduction is unclear, Gilles Peskine's system achieves this distinction with a single, easily-understood sealing construct.

Hashes of simple modules serve as universally valid names for the abstract types that they define. In the generative case, nonces are used for the same effect. Hashes and nonces have a pendant in the type system, namely singleton signatures: the singleton signature of a module is the signature of that and computationally equivalent modules. Whereas Dreyer et al. restrict module equivalent to type components, Gilles Peskine's system includes arbitrary higher-order singleton signatures, making it possible to track all module equalities (in particular between functor arguments) throughout the reduction process.

The provided run-time semantics treats generativity as a primitive, meaning that each build of an abstract type is incompatible with any other build. This is unsatisfactory in a distributed system, where modules built on different machines must be compatible when they provide location-independent functionality (e.g., data structure modules). However, when an abstract type is only created once at each site, it is possible to create it in the same manner on each site (obtaining the same type independently), making the system suitable for distributed environments. Thus the system can model marshaling between separate run-times of an ML-like language.

The PhD of G. Peskine will be defended in early 2007.

6.5. Bi-graphical models of concurrency

Participants: James Leifer, Robin Milner [Univ. of Cambridge and École polytechnique].

A framework is defined within which reactive systems can be studied formally. The framework is based on *s-categories*, which are a new variety of categories within which reactive systems can be set up in such a way that *labeled transition systems* can be uniformly extracted. These lead in turn to behavioral preorders and equivalences, such as the failures preorder (treated elsewhere) and bisimilarity, which are guaranteed to be congruential. The theory rests on the notion of *relative pushout*, which was previously introduced by the authors.

The framework is applied to a particular graphical model, known as *link graphs*, which encompasses a variety of calculi for mobile distributed processes. The specific theory of link graphs is developed. It is then applied to an established calculus, namely *condition-event Petri nets*.

In particular, a labeled transition system is derived for condition-event nets, corresponding to a natural notion of observable actions in Petri-net theory. The transition system yields a congruential bisimilarity coinciding with one derived directly from the observable actions. This yields a calibration of the general theory of reactive systems and link graphs against known specific theories.

This work is published in [14].

6.6. Cryptographic primitives

Participants: Karthikeyan Bhargavan[Microsoft Research-INRIA] „ Ricardo Corín „ Pierre-Malo Deniélou, Cédric Fournet „ James Leifer.

Distributed applications can often be structured as roles that exchange messages according to some pre-arranged communication patterns. These sessions (or contracts, or protocols) simplify distributed programming: when coding a given role for a session, one just has to ensure that the code follows the intended message flow, under the assumption that the other roles are also compliant.

In an adversarial setting, however, remote parties may not be trusted to play their role. Hence, defensive implementations also have to monitor one another, in order to detect any deviation from the assigned roles of a session. This task involves low-level coding below session abstractions, thus giving up most of their benefits.

We explore language-based support for sessions. We extend the ML language with session types that express flows of messages between roles, such that well- typed programs always play their roles. We compile session type declarations to cryptographic communication protocols that can shield programs from any low-level attempt by coalitions of remote peers to deviate from their roles. Our main result is that, when reasoning about programs that use sessions, one can assume that all peers are well-typed, irrespective of their actual implementations.

6.7. Join-calculus with values and pattern-matching

Participants: Luc Maranget, Ma Qin.

Making concurrent programming more practical is one of our long-term goals. Ma Qin's thesis (started in September 2001, supervised by Pierre-Louis Curien, PPS-CNRS, and Luc Maranget) was part of this effort. Ma Qin completed and defended her thesis in 2005. Her dissertation explored ideas in the design, formalism, and implementation issues of concurrent languages. An important part of the dissertation proposed a class-based object oriented extension of the join-calculus. In 2006, we pursued a publication effort of Ma Qin's results, more specifically the unpublished part about information hiding [23]

The goal is to provide information hiding support in concurrent object-oriented programming languages. We study it at the object level and at the class level, in the context of an object-oriented extension of the join calculus. At the object level, we improve a privacy mechanism proposed in prior work by defining a simpler chemical semantics with privacy control during execution. At the class level, we design a new hiding operation on classes, aimed at preventing part of parent classes from being visible in client (inheriting) classes. We gave the formal semantics of this new operation in terms of α-converting hidden names to fresh names, and its typing in terms of eliminating hidden names from class types. We show the standard soundness property of the corresponding type system, as well as specific properties concerning hiding.

In addition, we prepare a journal publication of another part of Ma Qin's thesis, about mixing algebraic pattern matching and join-matching (published at CONCUR 2004). Ma Qin, presently in post-doc at Oldenburg, visited our project-team in December 2006; the journal publication will be enriched from experience with the JoCaml system, which implements both kinds of matching. In particular, we use these matchings to provide a compact coding of the (concurrent) readers–writers idiom.

6.8. Reversible process algebra and transactions

Participants: Vincent Danos [PPS-CNRS], Jean Krivine, Pawel Sobocinski [(Cambridge, UK)].

In [3], we introduced the reversible calculus of communicating systems (RCCS), which is essentially Milner's CCS [36] with the caveat that some observable actions in the standard labeled transition system (LTS) semantics are understood to be reversible. Technically, this involved first the development of an explicit syntax for keeping track of a computation history. Such a history, together with an RCCS term, forms the configuration of a given process. Secondly, a new LTS semantics, given by providing appropriate structural operational semantics (SOS) rules, allowed the reversible components of a given state's history to be undone. In [19], it was argued that a calculus such as RCCS makes it much easier to model *transactions* – ie computations where several agents interact in order to agree on a common irreversible action. Indeed, it seems that guaranteeing the soundness of such transactions is easy enough since policies are normally specified by requiring the local states of the participants to satisfy certain criteria. On the other hand, completeness seems to be more difficult, since the existence of a possible computation leading to all of the agents having the required state does not guarantee that such a state will be reached – for instance, the agents may deadlock while racing to obtain the necessary shared resources. If we stipulate that the actions leading to transactions are reversible and enrich the participants with a history then such intermediate actions can be undone, meaning that the reversible computations are "essentially" the transactions. In terms of the underlying LTSs, it was shown in [19] that the LTS where the labels are taken to be the transactions and the LTS of processes with histories and reversible actions, where the reversible actions are equated with τs, are weakly bisimilar. It results in a easier way of programming transactions which we call *Declarative Concurrent Programming*. See [21] for a fully worked-out example which presents an implementation of a paradigmatic example of distributed consensus. This work was presented at SOS'2006.

We started a collaboration with Pawel Sobocinski (Computer Laboratory, Cambridge University, UK). The idea was to study to what extent the principle of reversing a process algebra could be extended to other kind of concurrent formalisms. Indeed we have shown that the design of a calculus such as RCCS involves an underlying abstract construction of the "history" category from a category of computations. The fact that the computations agree essentially with the "causal" computations in the original category is captured by an equivalence of categories. This work was presented at EXPRESS'06 [20] and in Krivine's dissertation [12].

6.9. Concurrency and bio-modelling

Participant: Jean Krivine.

In July 2006 Jean Krivine was invited at Harvard Medical School (Cambridge, USA), in Walter Fontana's laboratory, to collaborate on the implementation of a platform that would allow biologists to design models of biomolecular systems. The idea is to use a concurrent formalism, based on a graph rewriting system, close to the κ-calculus designed by Vincent Danos and Cosimo Laneve [30]. This joint work will bring the expertise on reversible process algebras and programming into this long term project that should proceed until 2008.

6.10. Labeled Lambda-Calculus and Variants

Participants: Tomasz Blanc, Jean-Jacques Lévy, Luc Maranget.

We introduced a new property of the labeled lambda-calculus: *context irreversibility*. We have $C[M] \to C[M']$ if and only if $M \to M'$. This property shows that when a (labeled) context disappears at one point of a reduction, this disappearance is irreversible: the context cannot be rebuilt in the reduction that follows. In the (unlabeled) lambda-calculus, this property is false: we have $(\lambda x.xy)y \to yy$ and $\lambda x.xy\neg \to y$. Lévy had put in light in his thesis that syntactic coincidences might occur in the lambda-calculus [35]. These coincidences are responsible for the loss of context irreversibility in the unlabeled calculus. Context irreversibility is similar to time irreversibility: labeled contexts can be used to time-stamp a reduction.

We examined two variants of the lambda-calculus. In the lambda-calculus by value, only redexes whose right part is a value (i.e. an abstraction) are contracted. The labels express the property of stability in the lambda-calculus but not in the lambda-calculus by value. We adapted the labels to recover this property. The lambda-calculus by value and the labeled lambda-calculus by value are confluent and verify the theorems of finite developments and standardization (although the definition of a standard reduction is adapted). To prove finite developments, we used an elegant, intuitive technique based on a notion of *future redex imbrication*.

We also examined the weak lambda-calculus. Although its syntactic properties did not receive great attention in the past decades, this theory is more relevant for the implementation of programming languages than the usual theory of the strong lambda-calculus. Contrary to the latter and similarly to lazy functional languages, the weak lambda-calculus does not validate the ξ-rule i.e. the reduction under a lambda-abstraction. Without this rule, the weak lambda-calculus is not confluent. We based our labeled language on a variant of the weak lambda-calculus by the Çağman and Hindley for Combinatory Logic [45]. In this variant, a new ξ'-rule is valid: it allows reductions under lambda-abstraction in sub-terms that do not contain a bound variable. This variant enjoys confluence. We proposed a labeling of this language that expresses a confluent theory of reductions with sharing, independent of the reduction strategy. If the sub-terms of a term are initially labeled with distinct letters, then, after some steps of reduction, two sub-terms that have the same label are equal. This formal setting corresponds to the shared evaluation strategy by Wadsworth defined with dags in 1971. This work was published on the occasion of Jan Willem Klop's 60th birthday [13]. A slightly simplified version of this work, context irreversibility and the labeled lambda-calculus by value are presented in Tomasz Blanc's thesis [11].

6.11. Security Properties in the Lambda-Calculus

Participants: Tomasz Blanc, Jean-Jacques Lévy.

In extensible virtual machines such as the JVM or the CLR, software components from various origins (applets, local libraries, ...) share the same local resources (CPU, files, ...) but not necessarily the same level of trust. We model access control in such semi-trusted environment in a minimal language based on the lambda-calculus. We aim at expressing in this framework a variety of known access control mechanisms.

The stack inspection is a dynamic access control mechanism that is used in the JVM and the CLR. Before accessing a sensitive resource, the call stack is inspected to check that every caller is allowed to access the resource. This mechanism is difficult to handle since it depends on the runtime stack, which is not statically known. Moreover, Fournet and Gordon showed that it is surprisingly hard to express what security property is guaranteed by stack inspection [32]. With static analysis, one can guarantee statically that such security defects will not occur [1] [33].

Flow analysis is another approach with stronger theoretical foundations. This analysis consists in tracking secret data along the execution of a program: these data are not accessible to untrusted code [43]. This approach has two advantages: (1) it is particularly adapted to static analysis and (2) it formally guarantees a security property: non-interference.

Nevertheless a static flow analysis is insufficient to express inherently dynamic security policies such as the Chinese Wall policy which is inspired by interest conflicts [29], [41]. This policy is defined as follows: initially, Alice may choose to communicate with Bob or Charlie; but once she communicated with Bob (resp. Charlie), she is not allowed anymore to communicate with Charlie (resp. Bob).

All these security mechanisms rely crucially on history of past events. To trace this history in the lambda-calculus, we use Lévy's labels: the labels of redexes keep track of the past interactions that created it [35]. We introduced a variant of the labeled lambda-calculus where the contraction of a redex is conditioned by the name of the redex and by its context. We proved that an instance of this language corresponds to a reasonable variant of the lambda-calculus with stack inspection that was proposed by Fournet and Gordon in [32].

We examined the property of non-interference in the lambda-calculus and in the lambda-calculus by value. A sub-term of an initial term M interferes if it influences the result of M's reduction. We showed that this property is strongly related with the stability property. In a lambda-calculus with references, in addition to the "functional" interference that already exists in the pure lambda-calculus, there is a "memory" interference due to the fact that the use of a reference may influence the result of a reduction. More precisely, the content of a reference during a time interval may have an influence. We adapted the labels of the lambda-calculus to capture this phenomenon: we took advantage of context irreversibility property to time-stamp reductions. In this labeled calculus, we formally expressed (non-)interference.

We introduced principals in the lambda-calculus. In this calculus, we defined a new security property: independence. A reduction $\mathcal{R}$ involving two principals A and B is independent from the interaction between A and B if this reduction may be decomposed into two reductions $\mathcal{R}_A$ and $\mathcal{R}_B$ where $\mathcal{R}_A$ (respectively $\mathcal{R}_B$) ignores the principal A (resp. B). To express the Chinese Wall policy, we took advantage of the labels of the lambda-calculus. We proved that a reduction that respects the Chinese Wall policy for Alice and Bob is independent from the interaction between Alice and Bob. This work is presented in Tomasz Blanc's thesis [11].

6.12. Concurrent C Minor

Participants: Andrew Appel, Francesco Zappa Nardelli, Cristal and Moscova.

Concurrent C Minor is an international research project to connect machine-verified source programs in sequential and concurrent programming languages to machine-verified optimizing compilers. It takes inspiration from Xavier Leroy's design of the (sequential) C Minor language and verified compiler and has goals complementary to Leroy's CompCert project [34].

In some application domains it is not enough to build reliable software systems, one wants proved-correct software. This is the case for safety-critical systems (where software bugs can cause injury or death) and for security-critical applications (where an attacker is deliberately searching for, and exploiting, software bugs). Since proofs are large and complex, the proof-checking must be mechanized. Machine-checked proofs of real software systems are difficult, but now should be possible, given the recent advances in the theory and engineering of mechanized proof systems applied to software verification. But there are several challenges:

- Real software systems are usually built from components in different programming languages.
- Some parts of the program need full correctness proofs, which must be constructed with great effort; other parts need only safety proofs, which can be constructed automatically.
- One reasons about correctness at the source-code level, but one runs a machine-code program translated by a compiler; the compiler must be proved correct.
- These proofs about different properties, with respect to different programming languages, must be integrated together end-to-end in a way that is also proved correct and machine-checked.

We address these challenges by defining a high-level intermediate language, Concurrent C Minor (CCm), with a small-step operational semantics representable in machine-checked proof assistants (we use Coq). We define a Concurrent Separation Logic [40] for reasoning directly about source programs written in CCm. In addition, CCm is suitable as a target language for compilers from Java, C, C#, ML, and other languages. This will allow safe and well-specified interaction of program modules written in different programming languages. We do not assume a sequentially consistent processor, and we formalize and reason about relaxed memory models. Finally, CCm is a concurrent language with a threads-and-locks model similar to Cthreads.

Web page of the project: http://moscova.inria.fr/~zappa/projects/cminor.

In collaboration with Sandrine Blazy (ENSIIE and INRIA Rocquencourt), Aquinas Hobor (Princeton University) and Adriana Compagnoni (Stevens Tech.).

6.13. Typed intermediate languages and typed assembly languages

Participant: Andrew Appel.

Andrew Appel, while a visiting scientist at INRIA Rocquencourt, collaborated with Paul-André Melliès and Jérôme Vouillon (both of CNRS-PPS, U. of Paris 7) on semantic models for type systems. They constructed a model of recursive and impredicatively quantified types with mutable references. In this model one can interpret all of the type constructors needed for typed intermediate languages and typed assembly languages used for object-oriented and functional languages. Then one establishes in this purely semantic fashion a soundness proof of the typing systems underlying these TILs and TALs ensuring that every well-typed program is safe. The technique is generic, and applies to any small-step semantics including lambda-calculus, labeled transition systems, and von Neumann machines. It is also simple, and reduces mainly to defining a Kripke semantics of the G¨odel-L¨ob logic of provability. A mechanical verification in Coq shows the soundness of the type system as applied to a von Neumann machine.

The paper is presented at POPL'07 (ACM Symposium of Principles of Programming Languages); this conference was very competitive in 2007 accepting (as full-length papers) only 17% of those submitted. [18].

7. Other Grants and Activities

7.1. National actions

7.1.1. France télécom

J.-J. Lévy was co-supervisor, with Pierre Crégut (France Télécom, Lannion), of the doctoral work of Guillaume Chatelet, who is working on extensions of Erlang with primitives for mobility. This PhD was defended on January 20, 2006, at the École polytechnique.

7.1.2. PARSEC

We started at end of 2006 a new project PARSEC, funded by the ANR (*Agence Nationale de la Recherche*), together with MIMOSA, EVEREST, LANDE project-teams of INRIA and the team of Roberto Amadio at CNRS-PPS, U. of Paris 7. This project is coordinated by G. Boudol. This project is about the design of programming languages for distributed applications and their security properties.

7.2. European actions

7.2.1. Collaboration with Microsoft

In 2006, we started to work at the Microsoft Research-INRIA Joint Centre in a common project with Cédric Fournet (MSR Cambridge), Gilles Barthe, Benjamin Grégoire and Santiago Zanella (INRIA Sophia-Antipolis). The project is named *Provable Secure Distributed Computing* and deals with security, programming languages theory and formal proofs.

8. Dissemination

8.1. Animation of research

J.-J. Lévy co-organized, with K. Gopinath (IISc), M. Crochemore (CNRS), R. Hariharan (IISc) et P. Népomiastchy (INRIA), a workshop at the Indian Institute of Science (IISc), Bangalore, on February 8-9, 2006.

Luc Maranget is elected member of *Comité technique paritaire* of Inria, 1 meeting per month about the general politics of Inria.

J.-J. Lévy is director of the Microsoft Research-INRIA Joint Centre, see http://www.msr-inria.inria.fr/. He participated to the start of the 3 research teams of so-called track A (Formal Methods); and to the new track B (Computational Sciences). He co-organises the official opening of the Centre on Jan 11, 2007.

J.-J. Lévy co-organizes, with Y. Bertot, G. Huet, G. Plotkin, the Gilles Kahn Colloquium, Jan 11, 2007, in memory of our ex INRIA president.

J. Leifer is member of the post-doctoral hiring committee.

J.-J. Lévy chaired the committee for hiring new researchers (CR2) at Rocquencourt; this tiresome and time-consuming job lasted over 4 months: Feb-Mar-Apr-May 2006.

J.-J. Lévy participated to the jury of following PhDs: Louis Mandel, Paris 6, May 22 (reviewer); François-Régis Sinot, École polytechnique, Sep 19; Samuel Hym, Paris 7, Dec 1.

8.2. Teaching

Our project-team participates to the following courses:

- “Informatique fondamentale”, 2nd year course at the Ecole polytechnique, 213 students (L. Maranget wrote the examination text and corrected the examination; J.-J. Lévy, professor on sabbatical years, is the author of the lecture notes[27], 271 pp.);
- “Bases de la programmation et de l’algorithmique”, 1st year course at the Ecole polytechnique, 100 students, 9×1.5 hours, (L. Maranget is the professor “chargé de cours”, in charge of this course; in 2006, L. Maranget and P. Baptiste, LIX, wrote new lecture notes [25], 219 pp.);
- “Concurrency”, Master Parisien de Recherche en informatique (MPRI), 2005-2006, at U. of Paris 7, 35 students, (J. Leifer taught the pi-calculus semantics: 12 hours[26]);
- “Concurrency”, Master Parisien de Recherche en informatique (MPRI), 2006-2007, at U. of Paris 7, 35 students, (F. Zappa Nardelli was in charge of 1/4 of this graduate course: 15 hours + the final exam);
- “Informatique Fondamentale 1”, 1st year course at U. of Paris 7, 2 groups (15 + 27 students) among 350 students (P.-M. Deniélou did the lab classes on Java programming).

We also had following activities related to teaching:

- J.-J. Lévy co-authored 3 computer science problems (4h+2h+2h) of the entrance examination at the Ecole polytechnique in 2006. He is the coordinator of the Computer Science examinations at École polytechnique.
- J.-J. Lévy wrote the first chapter of a french encyclopedia of Computer Science, entitled *Encyclopédie de l’informatique et des systèmes d’information*, published by Vuibert in early 2007 [15];
- J.-J. Lévy also co-authored with Gilles Dowek the book *Introduction à la théorie des langages de programmation*, which gathers lecture notes of a course about principles of programming languages at École polytechnique [10].

8.3. Partipations to conferences, Seminars, Invitations

8.3.1. Participations to conferences

- Jan 30-Feb 3, J. Krivine was invited at the Geometry of Computation 2006 (GEOCAL’06) workshop at the Institut of Mathématiques, Luminy-Marseille. He presented a talk on concurrent formalisms for biology.

- July 11-21, J. Leifer participated to the Summer School on Language-Based Techniques for Concurrent and Distributed Software in Eugene, Oregon. This summer school covered advanced research topics in applications of type theory and static analysis to problems of concurrent and distributed programming. The Acute programming language, of which Leifer is a co-author, was presented by Peter Sewell (Univ. of Cambridge).
- Sep 16-23, P.-M. Deniélou and J. Leifer participated to the International Conference on Functional Programming (ICFP) in Portland, Oregon, USA. P.-M. Deniélou presented his paper "Abstraction preservation and sub-typing in distributed languages", written jointly with J. Leifer.
- Dec 6-8, L. Maranget presented a paper at ASIAN'06, Tokyo.

8.3.2. Other Talks and Participations

- Jan 12, J. Krivine gave a talk at the LSL group, CEA, Saclay.
- Feb 1, J. Krivine gave a talk at the "Logique and Programmation" group at the Institut of Mathématiques, Luminy-Marseille.
- Mar 9, F. Zappa Nardelli gave a talk on "Acute: exemples, typage et sémantique opérationelle" at the "Groupe de travail Programmation", PPS, U. Paris 7.
- Mar 24, J. Krivine gave a talk at the LIX/Comete seminar at Ecole polytechnique, Palaiseau.
- Mar 24-28, F. Zappa Nardelli visited the Computer Laboratory of the University of Cambridge for collaboration with Peter Sewell.
- Apr 5-7, P.-M. Deniélou and J. Leifer participated to a meeting at Microsoft Research, Cambridge, for scientific collaboration with Cédric Fournet on the design of authentication primitives in programming languages
- May 9-12, P.-M. Deniélou and J. Leifer participated to a meeting at INRIA Sophia with Gilles Barthe and Cédric Fournet to write the technical proposal for the creation of the new Microsoft Research-INRIA Joint Centre.
- Jun 7, J.-J. Lévy gave a talk about Theoretical Computer Science to the winners of the *Kangourou* of Mathematics in the *classes de 4ème* in the *lycées* at U. of Versailles-Saint-Quentin.
- Jun 22-26, F. Zappa Nardelli visited the Computer Science Department, University of Pisa, he presented a talk on "Acute: high-level programming language design for distributed computation".
- Jun 25-27, P.-M. Deniélou and J. Leifer participated to a meeting at Microsoft Research, Cambridge, with Gilles Barthe and Cédric Fournet, to refine our research agenda for the new Microsoft Research-INRIA Joint Centre.
- Aug 30-31, J.-J. Lévy visited Microsoft Research in Cambridge as new director of the Microsoft Research-INRIA Joint Centre.
- Sep 4, T. Blanc gave a talk on the security properties in the lambda-calculus at Vrije Universiteit, Amsterdam, Netherlands.
- Oct 2, F. Zappa Nardelli gave a talk on "Ott: Tool support for the working semanticist" at the "Groupe de travail Moscova/Gallium".
- Nov 14, L. Mandel gave a talk on "Design, Semantics and Implementation of ReactiveML" at the *ALCHEMY Seminar*, INRIA Futurs Saclay.
- Nov 22-24, J.-J. Lévy and L. Mandel attended to PaRI-STIC in Nancy. Posters on the ACI ALIDECS and PARSEC were presented.
- Dec 9-12, P.-M. Deniélou and J. Leifer participated to a meeting at Microsoft Research, Cambridge, for scientific collaboration with Cédric Fournet on the the semantics and implementation of secure session primitives in ML-like languages.

8.3.3. Visits

- May 28-Jun 2, Matthew Parkinson visited Rocquencourt for collaboration with A. Appel and F. Zappa Nardelli. He gave a talk on “Concurrent separation logics”.
- May-Jun-Jul, Sanchit Garg (Indian Institute of Technology, Bombay) worked in the Moscova research-team on Burfiks under the direction of F. Zappa Nardelli. He was supported by the program *Internship at INRIA for international students*.
- Sep 11- Oct 2, Luca Padovani, Univ. of Bologna, visited our project-team.

9. Bibliography

Major publications by the team in recent years

[1] F. Besson, T. Blanc, C. Fournet, A. D. Gordon. *From Stack Inspction to Access Control: A Security Analysis for Libraries*, in "17th IEEE Computer Security Foundations Workshop", June 2004, p. 61–75.

[2] T. Blanc, J.-J. Lévy, L. Maranget. *Sharing in the weak lambda-calculus*, in "Terms and Cycles: Steps on the Road to Infinity. Essays dedicated to Jan Willem Klop", 2005.

[3] V. Danos, J. Krivine. *Reversible communicating systems*, in "Proceedings of CONCUR’04: 15th International Conference on Concurrency Theory", LNCS, vol. 3170, Springer-Verlag, Sep 2004, p. 292-307.

[4] C. Fournet, G. Gonthier. *The Reflexive Chemical Abstract Machine and the Join-Calculus*, in "Proceedings of the 23rd Annual Symposium on Principles of Programming Languages (POPL) (St. Petersburg Beach, Florida)", ACM, January 1996, p. 372–385.

[5] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, D. Rémy. *A Calculus of Mobile Agents*, in "CONCUR ’96: Concurrency Theory (7th International Conference, Pisa, Italy, August 1996)", U. Montanari, V. Sassone (editors). , LNCS, vol. 1119, Springer, 1996, p. 406–421.

[6] C. Fournet, C. Laneve, L. Maranget, D. Rémy. *Inheritance in the join calculus*, in "Journal of Logics and Algebraic Programming", vol. 57, n^{o} 1–2, September 2003, p. 23–29.

[7] J. J. Leifer, G. Peskine, P. Sewell, K. Wansbrough. *Global abstraction-safe marshalling with hash types*, in "Proc. 8th ICFP", Extended Abstract of INRIA Research Report 4851, 2003, 2003, http://pauillac.inria.fr/~leifer/research.html.

[8] J.-J. Lévy. *Réductions correctes et optimales dans le lambda-calcul*, Ph. D. Thesis, Université Paris 7, 1978.

[9] M. Qin, L. Maranget. *Expressive Synchronization Types for Inheritance in the Join Calculus*, in "Proceedings of APLAS’03, Beijing China", LNCS, Springer, November 2003.

Year Publications

Books and Monographs

[10] G. Dowek, J.-J. Lévy. *Introduction à la théorie des langages de programmation*, ISBN : 2-7302-1333-3, Les Éditions de l’École polytechnique, 2006.

Doctoral dissertations and Habilitation theses

[11] T. BLANC. *Propriétés de sécurité dans le lambda-calcul*, Ph. D. Thesis, Ecole Polytechnique, 2006.

[12] J. KRIVINE. *Algèbres de Processus Réversibles*, Ph. D. Thesis, Université Paris 6 & INRIA-Rocquencourt, 2006, http://moscova.inria.fr/~krivine.

Articles in refereed journals and book chapters

[13] T. BLANC, J.-J. LÉVY, L. MARANGET. *Processes, Terms and Cycles: Steps on the Road to Infinity. Essays dedicated to Jan Willem Klop*, LNCS, chap. Sharing in the weak lambda-calculus, n^{O} 3838, Springer, December 2005.

[14] J. J. LEIFER, R. MILNER. *Transition systems, link graphs and Petri nets*, in "Mathematical Structures in Computer Science", vol. 16, n^{O} 6, 2006, http://pauillac.inria.fr/~leifer/.

[15] J.-J. LÉVY. *Encyclopédie de l'Informatique et des Systèmes d'Information*, ISBN:2-7117-4846-4, chap. Algorithmique et programmation – Structures de Données, ed. Jean-Eric Pin, Vuibert, 2007.

[16] L. MARANGET. *Warnings for pattern matching*, in "JFP", 2007.

[17] P. SEWELL, J. LEIFER, K. WANSBROUGH, F. Z. NARDELLI, M. ALLEN-WILLIAMS, P. HABOUZIT, V. VAFEIADIS. *High-level programming language design for distributed computation*, in "JFP", 2007.

Publications in Conferences and Workshops

[18] A. W. APPEL, P.-A. MELLIES, C. D. RICHARDS, J. VOUILLON. *A Very Modal Model of a Modern, Major, General Type System*, in "34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Nice", January 2007.

[19] V. DANOS, J. KRIVINE. *Transactions in RCCS*, in "Proceedings of CONCUR'05: 16th International Conference on Concurrency Theory", LNCS, vol. 3653, Springer-Verlag, August 2005.

[20] V. DANOS, J. KRIVINE, P. SOBOCIŃSKI. *General reversibility*, in "Proceedings of EXPRESS'06", ENTCS, To appear, 2006.

[21] V. DANOS, J. KRIVINE, F. TARISSAN. *Self Assembling Trees*, in "Proceedings SOS'06", ENTCS, To appear, 2006.

[22] P.-M. DENIÉLOU, J. J. LEIFER. *Abstraction preservation and subtyping in distributed languages*, in "Proc. 11th ICFP", 2006, http://pauillac.inria.fr/~leifer/.

[23] MA QIN, L. MARANGET. *Information Hiding in the Join-Calculus*, in "11th Annual Asian Computing Science Conference (ASIAN'06), Tokyo", LNCS, Springer, December 2006.

[24] P. SEWELL, J. J. LEIFER, K. WANSBROUGH, F. ZAPPA NARDELLI, M. ALLEN-WILLIAMS, P. HABOUZIT, V. VAFEIADIS. *Acute: High-level programming language design for distributed computation*, in "Proc. ICFP 2005", ACM Press, September 2005, p. 15-26.

Internal Reports

[25] P. BAPTISTE, L. MARANGET. , École polytechnique, Aout 2006, http://www.enseignement.polytechnique.fr/profs/informatique/Lu

[26] J. J. LEIFER. , MPRI, January 2006, http://pauillac.inria.fr/~leifer/teaching/mpri-concurrency-2005/.

[27] J.-J. LÉVY. , Ecole polytechnique, Janvier 2005, http://www.enseignement.polytechnique.fr/informatique/IF.

[28] P. SEWELL, J. J. LEIFER, K. WANSBROUGH, M. ALLEN-WILLIAMS, F. ZAPPA NARDELLI, P. HABOUZIT, V. VAFEIADIS. *Acute: High-level programming language design for distributed computation. Design rationale and language definition*, Also published as UCAM-CL-TR-605, U. of Cambridge. 193 pp., Technical report, n^{o} RR-5329, INRIA, October 2004, http://hal.inria.fr/inria-00070671.

References in notes

[29] D. F. C. BREWER, M. J. NASH. *The Chinese Wall Security Policy*, in "Proceedings of the 1989 IEEE Symposium on Security and Privacy", 1989, p. 206–214.

[30] V. DANOS, C. LANEVE. *Graphs for Formal Molecular Biology*, in "Proceedings of the First International Workshop on Computational Methods in Systems Biology (CMSB'03, Rovereto, Italy)", LNCS, vol. 2602, Springer-Verlag, 2003, p. 34–46.

[31] D. DREYER, K. CRARY, R. HARPER. *A Type System for Higher-Order Modules*, in "Proc. 30th POPL, New Orleans", 2003, p. 236–249, http://www-2.cs.cmu.edu/~rwh/papers.htm.

[32] C. FOURNET, A. D. GORDON. *Stack Inspection: Theory and Variants*, Technical report, n^{o} MSR–TR–2001–103, Microsoft Research, 2001.

[33] T. JENSEN, D. L. MÉTAYER, T. THORN. *Verification of control flow based security propertie*, in "Proceedings of the 1999 IEEE Symposium on Security and Privacy", IEEE Computer Society Press, 1999, p. 89–103.

[34] X. LEROY. *Formal certification of a compiler back-end, or: programming a compiler with a proof assistant*, in "33rd symposium Principles of Programming Languages", ACM Press, 2006, p. 42–54.

[35] J.-J. LÉVY. *Réductions correctes et optimales dans le lambda-calcul*, Ph. D. Thesis, Université Paris 7, 1978.

[36] R. MILNER. *Communication and Concurrency*, International Series on Computer Science, Prentice Hall, 1989.

[37] R. MILNER, J. PARROW, D. WALKER. *A Calculus of Mobile Processes, Parts I and II*, in "Journal of Information and Computation", vol. 100, September 1992, p. 1–77.

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

[39] B. C. PIERCE, D. N. TURNER. *Pict: User Manual*, Available electronically, 1997.

[40] J. REYNOLDS. *Separation logic: a logic for shared mutable data structures*, in "Invited paper, LICS'02", 2002.

[41] F. B. SCHNEIDER. *Enforceable security policies*, in "ACM Transactions on Information and System Security", vol. 3, n^o 1, February 2000, p. 30–50.

[42] N. SHAVIT, D. TOUITOU. *Software transactional memory*, in "Proc. ACM symposium on Principles of Distributed Computing", 1995.

[43] V. SIMONET. *Inférence de flots d'information pour ML: formalisation et implantation*, Ph. D. Thesis, Université Paris 7 - Denis Diderot, 2004.

[44] B. THOMSEN, L. LETH, T.-M. KUO. *A Facile Tutorial*, in "CONCUR '96: Concurrency Theory (7th International Conference, Pisa, Italy, August 1996)", U. MONTANARI, V. SASSONE (editors). , LNCS, vol. 1119, Springer, 1996, p. 278–298.

[45] N. ÇAĞMAN, J. R. HINDLEY. *Combinatory Weak Reduction in Lambda Calculus*, in "Theoretical Computer Science", vol. 198, 1998, p. pp. 239-249.