INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Project-Team Moscova

Mobility, Security, Concurrency, Verification and Analysis

Rocquencourt

THEME COM

Activity Report

2004

Table of contents

1. Team

Head of project-team
Jean-Jacques Lévy [DR INRIA]

Vice-head of project-team
Luc Maranget [CR INRIA]

Administrative assistant
Sylvie Loubressac [AI INRIA]

Research scientists
James Leifer [CR INRIA]
Francesco Zappa Nardelli [CR INRIA, since 10/01/2004]

Ph. D. students
Tomasz Blanc [INRIA grant, Corps des Télécom]
Guillaume Chatelet [France Télécom, Lannion]
Jean Krivine [MESR grant, Paris 6]
Pierre Habouzit [AMX grant, Paris 12, since 09/01/2004]
Ma Qin [MESR grant, Paris 7]
Gilles Peskine [AMN grant, Paris 7]

Student interns
Celia Franco [Epita, from 04/01/2004 to 07/30/2004]
Pierre Habouzit [DEA Paris 7, from 04/01/2004 to 07/30/2004]
Abhishek Thakur [IIT Delhi, from 05/15/2004 to 07/30/2004]

Visiting scientist
Andrew Appel [Princeton University, from 06/01/2004 to 07/14/2004]

2. 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 extendible 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 $C\sharp$ allow downloading of programs, but do not permit migrations of active programs. On the other hand, several prototype languages (Facile [31], Obliq, Nomadic Pict [28], 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 adressed in our project-team is targeted to programming languages. It is firstly concerned by type-safe marshalling for communicating data between different runtimes of programming languages; it is also related to the definition of dynamic linking and rebinding in runtimes; 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 and, especially in 2004, the usage of classical ML-like pattern-matching for synchronisation. 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, developped jointly at U. of Cambridge and at INRIA, demonstrates the feasibility of our ideas in type-safe marshalling, 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.

In 2004, Francesco Zappa Nardelli joined the Moscova project-team. Damien Doligez (and his activity on program specifications and verification) is no longer member of Moscova.

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 [25]. 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 [26] 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 [1][2], 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 expressivity 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 sublocations 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 [27] 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 $C\sharp$-like libraries.

3.3. 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 [5] 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. Acute, a prototype implementation of a high-level programming language for distributed computation

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. It is expected to be released at the end of December 2004.

5.2. Caml/Jocaml

Participants: Luc Maranget, Gilles Peskine.

Gilles Peskine wrote a lightweight ($\sim$1500 lines of Ocaml) implementation of the local fragment of Jocaml as a preprocessor targetting Ocaml. This implementation was not released. It fulfilled its purpose in giving some insight as to how each feature of Jocaml (in particular channels, synchronisation, compilation of join definitions, exception propagation) interacts with the rest of Ocaml.

Luc Maranget continued his long-term effort on a new version of Jocaml, the implementation of the Join-calculus integrated in Ocaml). Our aim is to fully integrate Jocaml into the Ocaml source, in order to easily follow up the successive releases of Ocaml. At the moment, there is only a branch from the source tree of Ocaml; this branch includes all the necessary additions to the standard Ocaml compilers and some additions to the runtime system.

More specifically, the modified compiler contains a specific parser, generating a small number of new abstract syntax tree nodes, specific to Jocaml. The typechecker is also slightly modified, to integrate the new typing rules for Jocaml constructs. The production of lambda-code is also modified in a similar manner. By contrast with abstract syntax trees, the definition of lambda-code remains standard, Jocaml constructs being compiled to standard lambda code plus calls to specific library functions. The work on the compiler performed this year consists of extending and debugging the first prototype, which was completed two years ago. One significant addition is the prototype implementation of the work of Ma Qin and Luc Maranget on extending Join-calculus with algebraic pattern matching [11].

In 2004, the main work consisted in writing the specific Jocaml library on top of Ocaml threads. The new, join library performs join matching and thread management (invisible to users). Luc Maranget took inspiration in the similar, but less ambitious, code written by Gilles Peskine (see above). All added code (about 650 lines of Ocaml) makes no asumption on the nature of threads. As a result, we now have a working, experimental version of Jocaml working on top of the three different kind of threads available in Ocaml (byte-code with custom threads and system threads, native code with system threads).

The existing system allows concurrent programming with Jocaml concurrent constructs, but does not features distributed communication. A specific serialization (or marshalling) library has yet to be written. New serialization routines are required for distributed communication, *i.e.* communication between different programs executing on different computers. Those routines are specific in the sense that they should replace communication channels by forwarders and manage channel names on a global basis. Luc Maranget should perform these extension in 2005.

5.3. Pattern matching in Ocaml

Keywords: *ocaml*, *pattern matching*.

Participant: Luc Maranget.

The work on warning messages for pattern matching in the Ocaml compiler (see the 2003 report) has been formally described and submitted for journal publication in 2004. This work demonstrates how to implement fast and complete tests for discovering non-exhaustive or useless (possibly disjunctive) patterns. These warning diagnostics have also been implemented by Luc Maranget in the Ocaml releases since 3.06.

5.4. Hevea

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

Participants: Luc Maranget, Abhishek Thakur.

Hevea is a fast translator from LaTeX to HTML, written in Ocaml. Hevea, first released in 1997, is maintained and developed by Luc Maranget. A continuous (although informal) collaboration around Hevea exists, including Philip H. Viton (Ohio State University) for the Windows port and Ralf Treinen (ENS Cachan) for Debian developments. Hevea consists of about 20000 lines of Caml and 5000 lines of package sources written in “almost TeX” (the working language of Hevea). In November 2004, Hevea has been downloaded from about 300 different sites. During this year, some important improvements were achieved by Abhishek Thakur (IIT Delhi) during his summer internship in our team under the supervision of Luc Maranget. The following new features have been added to Hevea:

- Mathematical symbols in Hevea output are now expressed as (numerical) Unicode entities. This makes more correct display of mathematics by modern browsers supporting Unicode; it no longer needs explicit changes of fonts.
- Abhishek Thakur designed and implemented a simple version of the “Babel” package for non-English and multilingual documents. This important addition to Hevea improves the processing of multilingual documents. The previous technique of using specific command-line flags for French or English is now superseeded. Benefits are better compatibility with LaTeX and availability of more languages.
- Several packages for the typesetting of inference rules are now present. These packages are important for the translation of scientific papers in areas close to mathematical logic.

Due to some weaknesses in the documentation, those additions were not released in 2004, but will be present in the next releases of Hevea, scheduled for 2005.

5.5. Tool support for semantics

Keywords: *interpreter*, *semantics*.

Participant: Gilles Peskine.

When describing new programming languages or calculi, it is common to advance on two fronts in parallel. A written description of the language is always needed, both to be reasoned upon and for presentation. An implementation of the theory is useful for experimentation as well as proof of feasability. Maintaining consistency between the two approaches can be painful. Gilles Peskine is working on a tool that can produce both a readable description of operational semantics and a working interpreter (in Objective Caml). This is work in progress.

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 5.1).

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 marshalling Type-safe marshalling 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 initialisation.

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 marshalled 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 occuring 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 synchronise 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 avaliable as an INRIA Research Report [15].

6.2. Type-safe marshalling for distributed environments with nested modules and parametricity

Participant: 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. The 2004 work expands on the core idea of using hashes to represent abstract types, this time in a state-of-the-art calculus for ML-style modules [22].

Challenging features include nested modules, the coexistence of generative and applicative functors, and that of generative and applicative sealing. Nested modules let the programmer encapsulate functionality as needed. Functors (functions on modules) are used where multiple components need to be combined to provide the desired behavior. Sealing lets one hide the implementation of a type so that it can only be used through its published interface: this is how ML-like languages support abstract types. Generativity means that each build of an abstract type is incompatible with any other build. Applicativity means that modules or types built in similar ways are identified. The tension between the two becomes apparent in a distributed system, where modules built on different machines must be compatible if they provide location-independent functionnality (e.g., a data structure module) but not if they are specific to a particular component of the system (e.g., module that accesses a particular file system).

A calculus combining the previous features with link-time typechecking that allows independently compiled programs to communicate is being designed and proved (no proof of type safety exists for the calculus described in [22]). This study is also a necessary step before support for type-safe marshalling with useful treatment of abstract types can be added to Ocaml.

6.3. Join-calculus

Participants: Luc Maranget, Ma Qin.

Join-calculus with values and pattern-matching

This year, the work on the Join-calculus consisted in combining ML pattern matching for data types and the Join-calculus. This feature is important for concurrent programming; it also is the heart of Ma Qin's thesis (started in 2001) under the supervision of Pierre-Louis Curien (PPS-CNRS) and Luc Maranget.

The semantical foundation of this work is a new extension of the Join-calculus: *the applied Join-calculus*. It is a value passing calculus, in the sense that channels now carry actual values, whereas ordinary join channels only carry (channel) names. The values that we consider are terms; Join-definitions can be defined by pattern maching on terms (as in ML), and not only on names.

Ma Qin and Luc Maranget propose a simple semantics for pattern matching in the new calculus. The new semantics is intuitive since both ordinary join-matching and algebraic pattern matching rely on just substitutions over expressions. This pattern matching can be compiled into the constructs of ordinary join-matching and classical ML pattern matching. Moreover, in practice, this approach yields a compilation scheme for pattern matching in Jocaml, which we implemented.

The value passing calculus implies a more realistic treatment of equivalence in a process calculus, with respect to encoding of values in a name-passing calculus. More specifically, ordinary bisimulation operates on closed processes only, yielding a behavioral equivalence of programs at runtime. The program equivalence that we consider operates on all processes without the ground restriction; it is defined from the previous runtime equivalence by universal quantification on ground substitutions. With this new "open" bisimulation, Ma Qin and Luc Maranget have proved that their compilation scheme is correct.

This addition to the Join-calculus is part (with previous work on object-oriented extensions) of Ma Qin's thesis, of which defense is scheduled for next year. It has been published at CONCUR'04 [11].

6.4. Reversible pi-calculus

Participants: Vincent Danos [PPS-CNRS], Jean Krivine.

In [10], we proposed a notion of distributed backtracking built on top of Milner's CCS. The backtracking mechanism is constructed as a distributed monitoring system meshing well with the specific features of CCS. Backtracking does not involve any additional communication structure and we obtained a new calculus, *Reversible CCS* (RCCS), that stays close to ordinary CCS.

With this reversible process algebra, we have clear-cut theoretical characterization of reversibility. Given a process and a past, one can show that the calculus allows consistent backtracking along any *causally equivalent* past. Computation traces originated from a given process are said to be causally equivalent when one can transform one trace into the other by commuting successive independent concurrent actions, or cancelling successive inverse actions.

A similar notion of computation trace equivalence exists in the λ-calculus which Lévy characterized with a suitable labelling system [5]. Thus, a good summary of this backtracking mechanism, is to say that RCCS is a Lévy labelling for CCS. Two reduction paths are equivalent if and only if they lead to the same process in RCCS.

To summarize, in [10], we proposed a syntax for reversible communicating systems, together with a characterization, in terms of causally equivalent traces, of the exact amount of flexibility enabled for backtracking.

A second step in our work is to study a more accurate version of RCCS in which several actions are declared *irreversible*. These actions are also called “commit actions”. As a direct effect, irreversible actions prevent the committed process from backtracking in its own local history. As a side effect, it implicitly commits other causally related processes. With this property, we obtain an elegant and new way to encode distributed consensus problems, which does not require a global coordinator (as in the two-phase commits protocols).

In these distributed consensus problems, a key point is to handle deadlocks arising during a round of computation. Typically, it happens when two processes independently take opposite decisions. In many cases, the hard point of the implementation is to define an exit by backtracking from these deadlocks. We are working on a generic method to handle this automatically. We prove that if a CCS encoding has deadlocks but satisfies good properties on the locations of committed actions, its sibling code in RCCS will be correct with respect to the specification (we can construct a bisimulation relation). This results in a generic way of encoding distributed consensus problems which are small codes, much easier to prove correct.

6.5. Security Analysis for Java- or $C\sharp$-like libraries

Participants: Frédéric Besson [Microsoft Research], Tomasz Blanc, Cédric Fournet [Microsoft Research], Andrew Gordon [Microsoft Research].

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. The stack inspection mechanism provides dynamic access control in such semi-trusted environments. Before accessing a sensitive resource, the call stack is inspected to check that every caller is allowed to access the resource. Stack inspection is a useful mechanism but it is also a source of complication: its behaviour depends on the runtime stack, which is not statically known. As expected, it entails difficulties in validation of libraries by testing and code review.

We developed a new formal model of the essentials of access control in the CLR (types, classes and inheritance, access modifiers, permissions, and stack inspection). In this model, we described a static analysis to help identify security defects. We stated and proved its correctness. We also built a tool based on this static analysis for the full CLR. Our tool inputs a set of libraries plus a description of the permissions granted to unknown, potentially hostile code. It constructs a permission-sensitive call graph, which can be queried to identify potential defects. The implementation allows to analyze large pre-existing libraries for the CLR.

This work was presented at the Computer Security Foundations Workshop [9].

6.6. Access Control for the Lambda-Calculus

Participants: Tomasz Blanc, Jean-Jacques Lévy.

Stack inspection is not the only mechanism that provides access control in a semi-trusted envoronment. Other security policies were proposed according to the context in which the question occured. Fournet and Gordon reviewed slight variants of stack inspection [23]. Abadi and Fournet proposed an other variant where a global history of executed calls replaces the stack [20]. Flow analysis provides another approach where secret data are not accessible to untrusted code [30]. The Chinese Wall policy, described below, is another security policy which is inspired by interest conflicts [21][29]. Chinese Wall: 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). This policy is inherently dynamic.

We currently work on a dynamic security mechanism in a minimal language based on the lambda-calculus. We intend to use Lévy's labels to trace history: labels of redexes keep track of the past interactions that created this redex [5]: this minimal security mechanism is dynamic and expressive enough to code a variant of stack inspection [23] or the Chinese Wall policy.

Since causality is the central concept in our security mechanism, we intend to formalise security properties in terms of causality, in a modal logic similar to the logic that Jensen, Le Métayer and Thorne used to statically analyse stack inspection by model-checking [24].

7. Other Grants and Activities

7.1. National actions

7.1.1. France télécom

J.-J. Lévy is 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 will be defended in 2005 at Ecole polytechnique.

7.2. European actions

7.2.1. PEPITO project

2004 is the third year of the european project PEPITO (*Peer to Peer, Implementation and Theory*), part of the Global Computing action inside Esprit. This project is a consortium with KTH (Seif Haridi, coordinator), U. of Cambridge (Peter Sewell), EPFL (Uwe Nestmann), SICS (Per Brand), UCL (Peter van Roy). It is concerned with theoretical and practical aspects of distributed and symmetric applications, with mobility. It federates efforts already started with Mozart (Mobile Oz), Jocaml and Scala.

We had a review meeting in Rovereto (February 2004). Our contribution was mainly around calculi for resource controls (Dynamic Join and M-calculus by A. Schmitt) and is now mostly around Type Safe Marshalling, a common effort with U. of Cambridge, which can be viewed as an other way of using safe resources. Celia Franco implemented, during her internship, a fully distributed post-it application on top of the P2P library in Mozart in collaboration with the Mozart group at U. of Louvain.

8. Dissemination

8.1. Animation of research

J.-J. Lévy chaired the 3rd IFIP International Theoretical Computer Science conference (Track 1 and 2), August 23-26, Toulouse. This conference was part of the 2004 IFIP World Computer Congress. J.-J. Lévy was also editor of the IFIP-TCS proceedings (600 pages).

J.-J. Lévy organised a CIMPA-Unesco school on Security of Computer Systems and Networks, which will be held on January 25-February 5, 2005, in Bangalore (India).

L. Maranget served in the program committee of the 2nd *Asian Symposium on Programming Language And Systems* (APLAS'04, http://www.comp.nus.edu.sg/~aplas/).

J.-J. Lévy has participated to the PhD juries of David Teller (ENS-Lyon) and Olivier Tardieu (Ecole des Mines).

L. Maranget acted as external reviewer for Michael Levin's Phd, at U. of Pennsylvania.

8.2. Teaching

Our project-team participates to the following courses:

- "Informatique fondamentale", 2nd year course at the Ecole polytechnique, 200 students (J.-J. Lévy is professor in charge of this course; L. Maranget, professor "chargé de cours") was chairing the programming projects; J.-J. Lévy revised the lecture notes (270 pages) [17]; L. Maranget also edited notes for programming projects [19].

- Compilers, 3rd year course at the Ecole polytechnique, 20 students (L. Maranget, professor “chargé de cours”; J. Leifer, “vacataire”, assisted him in laboratory courses). This course is available on the Web[18].
- Recursive Data Structures, 3rd year course at the Ecole polytechnique, 100 students (L. Maranget is in charge of this course) [19].
- Concurrency, “DEA Sémantique, preuves et programmation”, now renamed MPRI, at U. of Paris 7, 40 students, (J. Leifer, J.-J. Lévy, C. Palamidessi from project-team COMÈTE, P.-L. Curien from PPS-CNRS;, E. Goubault from CEA).

J.-J. Lévy co-authored with Gilles Dowek a book on the “Introduction à la théorie des langages de programmation”, which will be published in 2005.

J.-J. Lévy co-authored 3 computer science problems (4h+2h+2h) of the entrance examination at the Ecole polytechnique in 2004. He contributed to the reform of this examination for Computer Science; this may have impact on the teaching of computer science in the “classes préparatoires”.

L. Maranget acted as chief judge in the Southwestern Europe Regional ACM Collegiate Programming Contest organized by Ecole polytechnique. He supervised the design and testing of all problems, of which he authored three (in nine). He also took part to the judging process (Nov 20-21) and presented the solutions of all problems at the closing session of this contest.

8.3. Partipations to conferences, Seminars, Invitations

8.3.1. Participations to conferences

- January 11-16, J.-J. Lévy, J. Leifer, J. Krivine, T. Blanc, J. Peskine attended POPL 2004 and the associated workshops, in Venice, Italy.
- March 8-12, J. Leifer and J.-J. Lévy went to Rovereto (Italy) for the annual review of the European project PEPITO.
- April 18-May 1, and September 3-13, J. Leifer went to Cambridge (UK) for scientific collaboration with P. Sewell, K. Wansbrough, and F. Zappa Nardelli.
- May 15-20, J. Leifer and J.-J. Lévy gave an invited talk at the IWFMS in Nanjing (China). L. Maranget and Ma Qin also presented a paper at IWFMS.
- Jun 28-30, T. Blanc participated to the 17th IEEE Computer Security Foundations Workshop (CSFW’04), Pacific Grove, USA.
- Aug 21-17, J.-J. Lévy attended the IFIP-TCS conference, affiliated to the IFIP World Computer Congress, Toulouse.
- Aug 30-Sep 4, J. Krivine, J.-J. Lévy, L. Maranget, Ma Qin participated to the CONCUR’04 conference, London.
- Nov 15-17, J.-J. Lévy attended to the ACI workshop on Security, in Toulouse.
- August 22-27, J.-J. Lévy attended the IFIP TCS conference in Toulouse.
- November 8-9, F. Zappa Nardelli participated to the Research Seminar on Programming Languages and Models for Mobile Communicating Systems, at U. of Copenhagen, Denmark. He gave two talks, one on the behavioural theory of the calculus of Mobile Ambients, one on the semantics of the Acute programming language.
- Dec 16-17, J. Krivine attended the symposium on Converging Sciences, Trento, Italy.

8.3.2. Other Talks and Participations

- Dec 9, F. Zappa Nardelli gave a talk on the design of the Acute programming language at ENS-Lyon.
- May 24-28 and July 14-30, C. Franco went to Louvain la Neuve (Belgium) for scientific collaboration with P. Van Roy.
- Aug 14-21, G. Peskine participated to the AFP 2004 summer school on Advanced Functional Programming, Tartu, Estonia.
- Sep 23-24, J. Krivine visited I. Castellani, project-team MIMOSA, at INRIA Sophia-Antipolis.

8.3.3. Visits

- January, B. Blanchet gave a talk on automatic proof of strong secrecy for distributed protocols.
- February, L. Wischik (U. of Bologna) gave a talk on a distributed implementation of the Fusion calculus.
- April - May, A. Appel (Princeton University) visited for four weeks, during which he gave two talks on proof carrying code.
- September, B. Charron-Bost gave a talk on the distinction between two fundamental distributed algorithms, consensus and atomic commitment.
- October, P. Sewell visited Rocquencourt for collaboration with J. Leifer and F. Zappa Nardelli. He gave a talk on the design of the Acute programming language.
- October, T. Cothia gave a talk on type based distributed acces control.

9. Bibliography

Major publications by the team in recent years

[1] 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.

[2] 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.

[3] 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.

[4] 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://www.inria.fr/rrrt/rr-4851.html.

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

[6] M. QIN, L. MARANGET. *Expressive Synchronization Types for Inheritance in the Join Calculus*, in "Proceedings of APLAS'03, Beijing China", LNCS, Springer, November 2003.

Articles in referred journals and book chapters

[7] G. CASTAGNA, J. VITEK, F. ZAPPA NARDELLI. *The Seal Calculus*, in "Information and Computation", 2004.

[8] L. MARANGET. *On using hevea, a fast LaTeX to HTML translator*, in "Eutupon, 11/12", Democritus University of Thrace, 2004.

Publications in Conferences and Workshops

[9] 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.

[10] V. DANOS, J. KRIVINE. *Reversible Communicating Systems*, in "Proc. CONCUR 2004, London", LNCS, n° 3170, Springer, Sep 2004, p. 292-307.

[11] MA QIN, L. MARANGET. *Compiling Pattern Matching in Join-Patterns*, in "Proc. CONCUR 2004, London", LNCS, n° 3170, Springer, Sep 2004, p. 417-431.

Internal Reports

[12] J. J. LEIFER, R. MILNER. *Transition systems, link graphs and Petri nets*, 64 pp., Technical report, n° UCAM-CL-TR-598, U. of Cambridge, Computer Laboratory, August 2004.

[13] MA QIN, L. MARANGET. *Compiling Pattern Matching in Join-Patterns*, full version of CONCUR'04 paper, Technical report, n° 5160, INRIA, April 2004, http://www.inria.fr/rrrt/rr-5160.html.

[14] M. MERRO, F. ZAPPA NARDELLI. *Behavioural Theory for Mobile Ambients*, 61 pp., Technical report, n° RR-5375, INRIA, November 2004, http://www.inria.fr/rrrt/rr-5375.html.

[15] 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° RR-5329, INRIA, October 2004, http://www.inria.fr/rrrt/rr-5329.html.

[16] G. WINSKEL, F. ZAPPA NARDELLI. *New-HOPLA — a Higher-Order Process Language with Name Generation*, 38 pp., Technical report, n° RS-04-21, BRICS, October 2004.

Miscellaneous

[17] J.-J. LÉVY. *Informatique Fondamentale*, Ecole polytechnique, Janvier 2004, http://www.enseignement.polytechnique.fr/informatic

[18] L. MARANGET. *Cours de compilation*, Ecole polytechnique, Janvier 2004, http://www.enseignement.polytechnique.fr/profs/inform

[19] L. MARANGET. *Les bases de la programmation et de l'algorithmique*, Ecole polytechnique, Août 2004, http://www.enseignement.polytechnique.fr/profs/informatique/Luc.Maranget/421.

Bibliography in notes

[20] M. ABADI, C. FOURNET. *Access Control based on Execution History*, in "Network and Distributed System Symposium (NDSS'03)", Internet Society, February 2003, p. 107–121.

[21] 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.

[22] 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.

[23] C. FOURNET, A. D. GORDON. *Stack Inspection: Theory and Variants*, Technical report, n° MSR–TR–2001–103, Microsoft Research, 2001.

[24] 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.

[25] R. MILNER. *Communication and Concurrency*, Prentice Hall, 1989.

[26] 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.

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

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

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

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

[31] 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.