INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Project-Team Moscova

Mobility, Security, Concurrency, Verification and Analysis

Rocquencourt

THEME COM

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, X Télécom]
Guillaume Chatelet [France Télécom, Lannion]
Pierre-Malo Deniélou [AMN, Paris 7, from 09/01/2005]
Pierre Habouzit [AMX grant, Paris 12, to 07/01/2005]
Jean Krivine [MESR grant, Paris 6]
Ma Qin [MESR grant, Paris 7, to 11/01/2005]
Gilles Peskine [AMN grant, Paris 7, to 08/31/2005]
Gilles Peskine [INRIA grant, Paris 7, from 09/01/2005]

Student interns
Prashant Kumar [IIT Kanpur, from 05/15/2005 to 07/30/2005]
Sylvain Pradalier [ENS Cachan, from 10/01/2005 to 03/31/2006]

Visiting scientist
Andrew Appel [Princeton University, from 08/01/2005 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 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 $C\#$allow downloading of programs, but do not permit migrations of active programs. On the other hand, several prototype languages (Facile [44], Obliq, Nomadic Pict [40], 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. 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 2005, Ma Qin and Guillaume Chatelet defended their PhD. Andrew Appel (Princeton University) started his sabbatical year in project MOSCOVA. Pierre-Malo Deniélou is a new PhD student. We also had Prashant Kumar (IIT Kanpur), intern with J. Krivine, and Sylvain Pradalier (ENS Cachan) who started a work on the probabilistic π-calculus with F. Zappa Nardelli for applications to biology.

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 [37]. 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 [38] 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 [39] 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. 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. 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 online in source form from http://www.cl.cam.ac.uk/users/pes20/acute/.

5.2. Caml/Jocaml

Participant: Luc Maranget.

Luc Maranget continued his long-term effort of a new version of JoCaml, the implementation of the join-calculus integrated into Ocaml. A branch from the source tree of Ocaml now does exist. The branch includes all the necessary additions to the standard Ocaml compiler and to the runtime system.

Most of the compiler modifications had been completed in 2004, as well as a specific JoCaml library on top of the Ocaml threads. In the previous version of this prototype, the was no support for distributed computing, i.e. no synchronization between remote programs. Only concurrent threads in the same program were supported. In 2005, there was progress in the development of a more avanced prototype, with extended support for distributed programming.

The most significant addition is a specific serialization (or marshalling) library. This library is required to pass values between runtimes on different computers. Those routines are specific in the sense that they have to replace communication channels by forwarders and manage channel names on a global basis. Part of the JoCaml library is now written in the JoCaml language, including a specific *channel service* library that can be used by programs to publish channels on which they accept to provide a service.

The current implementation is now mature to be released as a beta version in 2006, this was not possible in 2005 by lack of time to write the proper documentation.

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 2003-2004 reports) has been sent for journal publication in january 2005. This articles also covers pattern matching for lazy languages (Haskell) and *or-patterns* for efficient disjunctive patterns. These warning diagnostics have been implemented by Luc Maranget in the current Ocaml releases. [10].

5.4. Hevea

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

Participant: Luc Maranget.

Hevea, first released in 1997, is a fast translator from LaTeX to HTML. Hevea is written in Ocaml, 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 20000 lines of Ocaml and 5000 lines of package sources written in “almost TeX” (the working language of Hevea). Hevea has been downloaded from about 350 different sites. In 2005, version 1.08 has been released. The main improvements are the introduction of style-sheets and the replacement of ad-hoc symbol fonts by unicode character specifications, as started in 2004 with Abhishek Thakur (IIT Delhi), intern at Moscova.

In the context of hevea, style-sheets allow users more control over the final appearance of their documents, they also provide Hevea with additional expressivity.

Using Unicode characters specifications makes Hevea output more compliant with published standards. Notice that it was necessary to wait for mainstream browsers to implement those standards, before Hevea was able to follow standards.

5.5. Active DVI

Keywords: *dvi, latex, tex.*

Participant: Francesco Zappa Nardelli.

`dvix` 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), and the software should be completed and distributed in the next few months.

5.6. 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 [26]. A paper describing our design principles decisions was published [18] 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* footcitetrans-memory: 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 realised 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 analyse some complex behaviours 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).

6.3. Subtyping and abstraction safety in distributed languages

Participants: Pierre-Malo Deniélou, James Leifer.

In an effort to extend the former work presented in [4] (the core calculus which developed into Acute), we worked on the addition of subtyping in order to study the future integration of objects or version handling in a full programming language. Our aim was to construct a type safe and abstraction safe core language including simple modules, some marshalling primitives, and a type system with subtyping (with structural subtyping resulting from records). To achieve this goal we used the the "colored brackets" of HAT to achieve a type preserving semantics that syntactically tracks a term's abstraction boundaries. Our goal was to remain as abstract as possible in treating record subtyping so that the semantic techniques developed should be easily generalizable.

Our resulting language is an extended lambda-calculus with tuples and records, a simple module system (with signatures to declare some types as abstract) and primitives for distributed computation such as marshalling. From the interaction between subtyping and abstract types, we included some richer features. First is the possibility to define *partially abstract* types, which are abstract types for which a supertype is known outside the module. We also allow abstract types to be declared explicitly as subtypes of each other through the `extends` and `restricts` keywords. This feature can be useful to declare some libraries as compatible with older versions for example. In these cases, we stated the minimal requirements that were necessary to guarantee the type safety and abstraction safety of the whole system.

These addition had consequences on the semantics of the language. For example, the complexity of the subtyping relation forced us to render all subtyping explicitly in the operational semantics. The semantic interactions between this explicit subtyping annotation and the color brackets was also the cause of a semantical change of the colour brackets: in our system as opposed as in Acute, they are additive, ie they do not denote a strict frontier of knowledge but only a privacy boundary that prevents the environment from illegitimately accessing the abstract data of a subexpression.

We have proved preservation, progress, and determinism for this system and we plan to extend this work with full polymorphism. We also would like to see part of this work implemented in Acute or another distributed language.

6.4. Type-safe marshalling

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. 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 [32] 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 functionnality (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 marshalling between separate runtimes of an ML-like language.

6.5. Join-calculus with values and pattern-matching

Participants: Luc Maranget, Ma Qin.

Ma Qin's thesis (started in September 2001, supervised by Pierre-Louis Curien, team PPS, CNRS, and Luc Maranget) Curien) has been completed and defended in October 2005 [6]. The summary of her work is following (quoted from her dissertation):

> It is widely acknowledged that concurrent programs are extremely difficult to get right. This dissertation aims to improve the situation by providing better language support for concurrent programming.
>
> In our view, an ideal language for concurrent programming would be a multiparadigm one: besides high-level concurrency, it would also feature important (and well established) programming styles such as functional and object-oriented. More importantly, the ideal language should possess firm semantical foundations in mathematical models that favor formal understanding and reasoning.
>
> This dissertation explores ideas in the design, formalism, and implementation issues of such a language. Attention is particularly paid to the theoretical core of the target language — the next release of JoCaml. As a response to the two weak points which we found either absent or poorly supported in current ongoing research work, we mainly focus on two directions: the interaction between concurrency and inheritance, and the interplay between pattern matching and synchronization. More specifically, we introduce an object-oriented extension of the join calculus for the first, and we propose an extension of the join calculus with algebraic pattern matching for the second. Both extensions are implemented (prototyped) in the context of the current JoCaml system.

In 2005, a solution to the problem of visibility control during inheritance [22] has been sent for publication. We study the issue both at the object level and 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. The formal semantics of our new operation has been defined in terms of α-renaming of hidden names into fresh names, and its typing in terms of eliminating hidden names from class types. We proved the soundness property of the type system, as well as specific properties concerning hiding. Our motivation stems from language design to produce a practical programming language. As an evidence of significance, we implemented our model in a prototype system.

6.6. Reversible pi-calculus

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

We have proposed a notion of distributed backtracking [13] built on top of Milner's CCS [37]. The backtracking mechanism was constructed as a distributed monitoring system meshing well with the specifics of the host calculus CCS. Backtrack needed not to involve any additional communication structure and we obtained a syntax, termed RCCS, for *reversible* CCS, staying really close to ordinary CCS.

This reversible process algebra offers a clear-cut theoretical characterization of reversibility. In particular, given a process and a past, we showed that RCCS allows to backtrack along any *causally equivalent* past. A similar notion of computation trace equivalence exists in λ-calculus which Lévy could characterize by a suitable labelling system [29]. Thus, a pretty good summary of the theoretical status of this backtracking mechanism, is to say that RCCS is a Lévy labelling for CCS. Two reduction paths will be equivalent if and only if they lead to the same process in RCCS.

In its fully reversible setting, one cannot really use RCCS as a language for describing distributed consensus because there is no way of validating a decision. During the year 2005, we have studied and formalized the notion of transaction one obtains when integrating such irreversible action within RCCS [15]. We also illustrated such formalism with a paradigmatic problem of distributed consensus [16].

More precisely, our work was to characterize, in Milner's sense, the visible behavior of a reversible process. It amounts to exhibit a labelled transition system (LTS) that is bisimilar to the RCCS term when we only observe irreversible actions. More precisely, given a CCS process P, we defined its lift into RCCS semantics $\ell_K(P)$, where the parameter K is a set of occurrences of irreversible actions. Our main contribution was to show that $\ell_K(P)$ is bisimilar to the *K-causal LTS* of P. A K-causal LTS is obtained by considering only minimal traces that may trigger actions in K, i.e where every transition cannot commute with the final commit.

We showed that this theorem could be used for the design of transactional systems: the idea is to break down the distributed implementation of a given reference specification in two steps. First, one writes down a code which is only required to meet a weaker condition of correctness relative to the specification (i.e whose K-causal transition system is bisimilar to the specification). Second, the obtained code is lifted into RCCS and we can apply the theorem to reduce the correctness of the latter (partially) reversible code to the causal correctness of the former.

In many transactional examples, this structured programming method works well, and obtains codes which are smaller, and simpler to understand [16]. It also seems interesting from a correctness perpective, since one never has to deal with the full state space, and it is enough to consider the much smaller state space of the forward code causal compression relative to observable actions. Thus it obtains codes which are also easier to prove correct. It is only natural then to ask whether and to which extent such indirect correctness proofs can be automated.

We have developped an Ocaml library that offers primitives to construct CCS processes and build their K-causal transition system. Specifically we have implemented an algorithm, which, under certain rather mild

assumptions about the system of interest, computes its causal compression relative to a choice of observables. To this aim we used Flow Event Structures [30], which are suited to the handling of causal relationships between transitions, as an internal representation of processes. By using event structures we could cumulate the compression effects from the non interleaved representation of processes (which is a classical verification technique), and the state space compression that is allowed by relaxing the full correctness condition to the correctness of the K-causal transition system.
(see http://pauillac.inria.fr/~krivine/causal/causal.html)

6.7. Access Control for 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 a unified 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 [33]. With static analysis, one can garantee statically that such security defects will not occur [12] [34].

Slight variants of stack inspection that guarantee a clearer security property were proposed by Fournet and Gordon [33]. Abadi and Fournet also proposed another variant where a global history of executed calls replaces the stack [28].

Flow analysis provides another approach where secret data are not accessible to untrusted code [43]. The Chinese Wall policy, described below, is another security policy which is inspired by interest conflicts [31], [42]. 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.

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]. This causality history is expressive enough to code a variant of stack inspection [33], information flow or the Chinese Wall policy. More specifically, we 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 [34].

Contrary to static information flow analyses, the building of Lévy's labels is dynamic. We intend to (1) extend the labeled lambda-calculus with a dynamic test of labels and (2) provide a type system that approximates runtime labels. Such a type system could statically guarantee that a term verifies a security property.

6.8. Sharing in the weak lambda calculus

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

Although the syntactic properties of the weak lambda-calculus 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. In this variant, a new ξ'-rule is valid: it allows reductions under lambda-abstraction in subterms 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, independant of the reduction strategy. If the subterms of a term are initially labeled with distinct letters, then, after some steps of reduction, two subterms 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 [7].

6.9. Formal verification of a lockfree algorithm

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

Properties and invariants of concurrent algorithms are most often described informally, using natural language. The application of formal methods, in particular those issued by semantics and concurrency theory, to the proof of properties of concurrent algorithms is a today necessary task, yet little efforts have been carried out in this direction.

In November Andrew Appel and Francesco Zappa Nardelli started a working group common to the Moscova and Cristal project-teams. Their long-term goal is to investigate how to use a theorem prover (namely Coq) to implement and prove the correctness of a simple (but subtle) algorithm for lock-free synchronisation [36]. Several sub-tasks emerged and revealed interesting in their own. Among them, we point out the formalisation in Coq of a relaxed memory model and the use of separation logic [41] as a tool to reason about programs with pointers.

6.10. List Machine Benchmark for Mechanized Metatheory

Participant: Andrew Appel.

There are about 20 different proof assistant systems for mechanized proofs. But several are better than others for safety proofs and soundness proofs of tools such as compilers. For instance Coq, Twelf and Isabelle/HOL seem more adequate. To make these systems usable by specialists of programming languages, it is useful that an expert in programming languages try these proofs in the different systems. With X. Leroy (CRISTAL), we developed a small benchmark to compare these proof assistant systems for proof of correctness of compilers. We have two solutions based on Coq and Twelf and wrote a short article on this.

Another work by A. Appel is on the semantics of types with P.-A. Melliès (PPS), J. Vouillon (PPS) and Chris Richard (Princeton). This new semantics for types in programming languages for von Neumann machines will preserve better properties than previous works by Melliès, Vouillon and Appel et al.

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 is defended on January 20, 2006, at Ecole polytechnique.

7.2. European actions

7.2.1. Collaboration with Microsoft

In 2005, we started investigations to be member of the new laboratory between INRIA and Microsoft. With Cédric Fournet (MSR Cambridge), Gilles Barthe and Benjamin Grégoire (INRIA Sophia-Antipolis), we defined some research project about *Provable Secure Distributed Computing*. This project will start in 2006.

8. Dissemination

8.1. Animation of research

J.-J. Lévy co-organised, with K. Gopinath (IISc), a CIMPA-Unesco school on Security of Computer Systems and Networks, was held on January 25-February 5, 2005, in Bangalore (India).

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 [20]; L. Maranget edited notes for the programming projects;
- "Bases de la programmation et de l'algorithmique", 1st year course at the Ecole polytechnique, 80 students (L. Maranget is the professor "chargé de cours", in charge of this course [24];
- Compilers, 3rd year course at the Ecole polytechnique, 20 students (L. Maranget, professor "chargé de cours" [23]; F. Pottier (CRISTAL) assisted him in laboratory courses; available on the Web
- Concurrency, "Master Parisien de Recherche en informatique" (MPRI), at U. of Paris 7, 35 students, (J. Leifer, J.-J. Lévy, F. Zappa Nardelli, C. Palamidessi from project-team COMÈTE, E. Goubault from CEA).
- Strong Static Typing and Advanced Functional Programming, Bertinoro International Spring School, March 14-18 (F. Zappa Nardelli – 20 graduate students, 15 hours of lecture plus final exam).

J.-J. Lévy authored 3 computer science problems (4h+2h+2h) of the entrance examination at the Ecole polytechnique in 2005. He is the coordinator of the Computer Science examinations for Ecole polytechnique.

L. Maranget acted as problem designer and judge in the Southwestern Europe Regional ACM Collegiate Programming Contest organized by the Ecole polytechnique, on November 2005.

8.3. Partipations to conferences, Seminars, Invitations

8.3.1. Participations to conferences

- Feb, 12-16, J. Leifer and F. Zappa Nardelli visited the Computer Laboratory of the University of Cambridge for collaboration with Peter Sewell.
- Mar 2005, J. Leifer presented our work on Acute to the final PEPITO scientific meeting held in the Univ. of Cambridge, UK. (PEPITO is a 3-year project of EU, of which we were member)
- Apr 2005, J. Leifer attended ETAPS 2005 (Edinburgh) and the final PEPITO review meeting during which the project was judged successfully completed.
- Aug 22-28, J. Krivine and J.-J. Lévy attended to Concur 2005, San Francisco. J. Krivine presented a paper at Concur, J.-J. Lévy was invited at the DisCoVeri workshop.
- Sep 1-2, J.-J. Lévy attented to a meeting at MSR Cambridge on the new INRIA/MS collaboration.
- Sep 24-30, J. Leifer, J.-J. Lévy, F. Zappa Nardelli attended the ICFP conference, to the Erlang and Merlin workshops, Tallinn. A paper co-signed by Leifer and Zappa Nardelli was presented at ICFP.
- Dec 22-23, J.-J. Lévy attended to the ACI workshop on Security, in Bordeaux.

8.3.2. Other Talks and Participations

- Sep 3–6, L. Maranget visited Benjamin Pierce (UPenn). He gave a presentation of his work on warnings for ML pattern matching, and was ember of the jury of Michael Levin's Ph.D. thesis.
- Oct, A. Appel visited Wolfganf Paul (Verisoft project, Sarrebrucken).
- 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

- April 15, Juliusz Chroboczeck (PPS, U. of Paris 7) gave a talk on "Continuation passing for C".
- May 26-29, Peter Sewell visited Rocquencourt for collaboration with J. Leifer and F. Zappa Nardelli.
- Sep 5-7, J. Leifer hosted an intensive 12 hour mini-course by Robin Milner (Emeritus Professor, Univ. of Cambridge, UK) on bigraphs.
  http://pauillac.inria.fr/~leifer/milner-paris/index.html.
- September 7-11, Peter Sewell visited Rocquencourt for collaboration with J. Leifer and F. Zappa Nardelli.
- October 3-7, Massimo Merro (U. of Verona) visited Rocquencourt for collaboration with F. Zappa Nardelli. He gave a talk on the formalisation and proof of an algorithm solving consensus in presence of a particular class of failure detectors.

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, nº 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, http://www.inria.fr/rrrt/rr-4851.html.

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

Doctoral dissertations and Habilitation theses

[6] MA QIN. *Concurrent Classes and Pattern Matching in the Join Calculus*, Ph. D. Thesis, Univ. Paris 7, September 2005.

Articles in refereed journals and book chapters

[7] 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° 3838, Springer, 2005.

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

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

[10] L. MARANGET. *Warnings for Pattern Matching*, in "JFP", submitted, 2005.

[11] M. MERRO, F. ZAPPA NARDELLI. *Behavioural theory for Mobile Ambients*, in "Journal of ACM", vol. 52, n° 6, November 2005, p. 961-1023.

Publications in Conferences and Workshops

[12] 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.

[13] V. DANOS, J. KRIVINE. *Reversible communicating systems*, in "CONCUR 2004 - Concurrency Theory", LNCS, vol. 3170, Springer-Verlag, Sep 2004, p. 292-307.

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

[15] V. DANOS, J. KRIVINE. *Transactions in RCCS*, in "CONCUR 2005 - Concurrency Theory", LNCS, vol. 3653, Springer-Verlag, 2005, p. 398-412.

[16] V. DANOS, J. KRIVINE, F. TARISSAN. *Self Assembling Trees*, in "Proceedings of EA'05 - 7th International Conference on Artificial Evolution", To appear, 2005.

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

[18] 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

[19] J. J. LEIFER, R. MILNER. *Transition systems, link graphs and Petri nets*, Technical report, nº UCAM-CL-TR-598, U. of Cambridge, Computer Laboratory, August 2004.

[20] J.-J. LÉVY. *Informatique Fondamentale*, Janvier 2005, http://www.enseignement.polytechnique.fr/informatique/IF.

[21] MA QIN, L. MARANGET. *Compiling Pattern Matching in Join-Patterns*, full version of CONCUR'04 paper, Technical report, nº 5160, INRIA, April 2004, http://www.inria.fr/rrrt/rr-5160.html.

[22] MA QIN, L. MARANGET. *Information Hiding, Inheritance and Concurrency*, Technical report, nº 5631, INRIA, July 2005, http://www.inria.fr/rrrt/rr-5631.html.

[23] L. MARANGET. *Cours de compilation*, Janvier 2005, http://www.enseignement.polytechnique.fr/profs/informatique/Luc.Maranget

[24] L. MARANGET. *Les bases de la programmation et de l'algorithmique*, Aout 2005, http://www.enseignement.polytechnique.fr/profs

[25] M. MERRO, F. ZAPPA NARDELLI. *Behavioural Theory for Mobile Ambients*, Technical report, nº RR-5375, INRIA, November 2004, http://www.inria.fr/rrrt/rr-5375.html.

[26] 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, Technical report, nº RR-5329, INRIA, October 2004, http://www.inria.fr/rrrt/rr-5329.html.

[27] G. WINSKEL, F. ZAPPA NARDELLI. *New-HOPLA — a Higher-Order Process Language with Name Generation*, Technical report, nº RS-04-21, BRICS, October 2004.

Bibliography in notes

[28] 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.

[29] G. BERRY, J.-J. LÉVY. *Minimal and optimal computation of recursive programs*, in "JACM", vol. 26, 1979, p. 148-175.

[30] G. BOUDOL, I. CASTELLANI. *Flow models of distributed computations: three equivalent semantics for CCS*, in "Information and Computation", vol. 114, 1994, p. 247-314.

[31] 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.

[32] 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.

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

[34] 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.

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

[36] M. MICHAEL. *Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects*, in "IEEE Trans. Parallel Distr. Syst.", vol. 15, n° 6, 2004, p. 491-504.

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

[38] 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.

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

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

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

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

[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.