HAL open science

Introducing modularity and sharing in a typed Lambda-calculus

Alain Costes

To cite this version:

Alain Costes. Introducing modularity and sharing in a typed Lambda-calculus. [Research Report] RR-1903, INRIA. 1993. inria-00077191

HAL Id: inria-00077191

https://inria.hal.science/inria-00077191

Submitted on 29 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Introducing modularity and sharing in a typed Lambda-calculus

Alain COSTE

N° 1903
Mai 1993

PROGRAMME 1

Architectures parallèles,
Bases de données,
Réseaux et Systèmes distribués

Rapport de recherche

1993

Introducing Modularity and Sharing in a Typed Lambda-Calculus[1]

Introduction des concepts de modularité et de partage dans un lambda-calcul typé

Alain Coste
INRIA
IRIT. Université Paul Sabatier
118, route de Narbonne. 31062-Toulouse
email: coste@irit.fr
fax: 61 55 62 58

Abstract

Thanks to the Curry-Howard isomorphism, typed lambda-calculi provide a convenient logical framework to formalize the concept of proof. The large size of proofs dictates the introduction of concepts of structuration and modularity. This in turn raises the problem of information sharing between modules. This paper presents the results of such a study for a language derived from Nederpelt's system Λ. We first discuss the reasons why sharing is not correctly supported, and then a new calculus is proposed, which allows the definition of modules. This calculus provides a mechanism of access between modules that let them share information. It is defined in terms of an operational semantics, and a comparison with the system Λ is provided.

Résumé

Grâce à l'isomorphisme de Curry-Howard, les lambda-calculs typés fournissent un cadre logique permettant de formaliser le concept de preuve. La taille importante des preuves impose d'introduire des concepts de structuration et de modularité, qui créent à leur tour des problèmes de partage de l'information entre modules. Cet article présente les résultats d'une telle étude pour un langage dérivé du système Λ de Nederpelt. Nous analysons d'abord les raisons pour lesquelles le partage n'est pas réalisé correctement, et nous proposons ensuite un nouveau calcul qui autorise la définition de modules. Ce calcul possède un mécanisme d'accès entre modules qui permet le partage de l'information. Le calcul est défini par une sémantique opérationnelle, puis comparé avec le système Λ.

---

1. This work is partially supported by the EUREKA program (ESF project).

Introduction

The will to formalize the notion of proof has led to the definition of formal systems in which mathematical objects (variables, functions,...) and logical objects (axioms, inference rules, theorems, proofs) are represented at the same time. The Curry-Howard isomorphism [How80], by establishing a correspondence between formal systems of logic on one hand and typed lambda-calculi on the other hand, makes the latter interesting candidates for such logical frameworks.

The large size of proofs dictates the introduction of concepts of structuration and modularity. Structuration is aimed at making proofs more readable and liable, while modularity allows to reuse previous proofs.

The basic lambda-calculus offers the notion of scope of a declaration as a structuration tool, but it does not contain the notion of modularity. Extensions have been defined, which formalize this notion, but a problem is frequently raised when a proof reuses modules: how will these modules share information? This problem is well-known in programming languages [Mac84], but we want to study it in the context of a typed lambda calculus used as a proof system.

This paper is aimed at proposing a calculus in which modules may at will share the data they import from other modules. This calculus, ΛS, is presented as an extension of the ΛC-calculus [Gro91], which is itself an extension of the system Λ of Nederpelt [Ned73], which belongs to the family of Automath languages [Deb80].

In order to introduce our formalism, we first give a partial definition of ΛC. We only present the elements which are useful for our purpose. We then emphasize the deficiencies of the importation mechanism of ΛC, and we propose a new mechanism of access between contexts, which solves difficulties.

1 The ΛC Calculus.

1.1 Informal presentation.

A system aimed at the formalization of reasoning must allow the representation of different concepts:

- Terms and types for mathematical objects.
- Propositions and proofs for logical objects.

The Curry-Howard isomorphism unifies the concepts of proposition and type on one side, of proof and term on the other side. A proposition is seen as the type of its proof (seen as a term). The formal system is simplified, the only concepts needed being terms and types.

The system Λ leads the simplification one step further, by identifying the notions of term and type. This identification is based on the following remark: abstraction $\lambda x{:}\alpha.\tau$ (in the language of terms) and dependent product $\Pi(x{:}\alpha)\alpha$ (in the language of types) play the same syntactical

role of binding operators, and can be identified. In the same way, application operations are identified in the two languages. This leads to a quite simple syntactic system. On the other hand, the role of an expression is not obvious from its syntax, but must be determined from the context of its use.

1.2 Syntax.

There are two main syntactical categories: texts and contexts. Texts are used to represent axioms, deduction rules, theorems and proofs. These elements are grouped into theories, represented by contexts. The language uses a unique constant **primal**, and a countable set of variables **X**.

expression:	**e**	::=	**t** | **c**
binding:	**b**	::=	**x:t** | **x:=t** | **x==c** | **import x**
seq of bindings	**bb**	::=	**b** | **b; bb**
text:	**t**	::=	**primal** | **x** | **[b ⊢ t]** | **t(t)** | **(t)**
seq of texts:	**tt**	::=	**t** | **tt,t**
context:	**c**	::=	**x** | **[|bb|]** | **c(t)** | **(c)**
variable:	**x**	∈	**X**

In a binding **b**, a variable **x** is bound either to a text **t** in a <u>declaration</u> **x:t** or in a <u>text definition</u> **x:=t**, or to a context **c** in a <u>context definition</u> **x==c**. Definitions allow the enrichment of the language by new constants.

The pseudo-binding **import x** is intended to be replaced by the bindings of the context bound to **x**.

A variable **x** appearing in any position but the left hand side of a binding is a <u>reference</u> to the variable **x**.

The abstraction $[x{:}t_1 \vdash t_2]$ may represent a function (with formal parameter **x** of type t_1), an inference rule, or the deduction of the proposition t_2 under the assumption t_1. Notice that **x** is bound in t_2, not in t_1.

The abstraction $[x{:=}t_1 \vdash t_2]$ is used to introduce an abbreviation of the expression t_1 by the variable **x** in the expression t_2.

The application $t_1(t_2)$ may represent the application of the function t_1 to the argument t_2, or the application of the inference rule t_1 to the proposition t_2.

The application **c(t)** represents the instantiation of the generic context **c** by the parameter **t**. Parenthesis **(t)**, **(c)** are used to eliminate syntactic ambiguities. Multi-ary functions are represented in a curryfied way, and the following abbreviations are used:

$[b_1 \vdash [b_2 \vdash t]]$	≡	$[b_1; b_2 \vdash t]$	
$[x_1{:}t_1; x_2{:}t_1 \vdash t_3]$	≡	$[x_1,x_2{:}t_1 \vdash t_3]$	
$[x{:}t_1 \vdash t_2]$	≡	$[t_1 \vdash t_2]$	if **x** not free in t_2
$(t_1(t_2))(t_3)$	≡	$t_1(t_2,t_3)$	
$(c(t_2))(t_3)$	≡	$c(t_2,t_3)$	

1.3 Semantics.

First of all, a remark must be made concerning the names of variables. In lambda-calculus, names of variables are irrelevant. This fact is expressed in the α-conversion rule $\lambda \mathbf{x.e} = \lambda \mathbf{y.[y/x]e}$. Moreover, when applying substitutions, renaming may be necessary to prevent variable captures. Here, a convention is made that an implicit renaming mechanism prevents name clashes, and expressions are said syntactically identical if it is true modulo α-conversion. This convention is justified by the existence of name-free notations, e.g. De Bruijn indices [Deb72], [Gro91].

The semantics of the language is expressed by the validity check (or type checking) of an expression. To check an expression, it is necessary to check its subexpressions. Even if the expression is closed, its subexpressions may not be so. Therefore, validity check must be carried out under an environment, which memorises the bindings visible from the checked expression. An environment is a sequence of declarations, and of definitions of texts and contexts:

1.3.1 Environments.

$$\mathbf{K} ::= \emptyset \mid \mathbf{K[x{:}t]} \mid \mathbf{K[x{:=}t]} \mid \mathbf{K[x{==}c]}$$

The search of a binding in an environment is defined by the sequent:

$\mathbf{K} \vdash_{sel} \mathbf{x} \rightarrow [\mathbf{x}\ \lambda \mathbf{e}]$
In the environment **K**, the variable **x** is bound to the expression **e**.

$$\mathbf{K[x\ \lambda\ e]} \vdash_{sel} \mathbf{x} \rightarrow [\mathbf{x}\ \lambda\ \mathbf{e}] \qquad \frac{\mathbf{K} \vdash_{sel} \mathbf{x} \rightarrow [\mathbf{x}\ \lambda \mathbf{e}]}{\mathbf{K[x'\ \lambda' e']} \vdash_{sel} \mathbf{x} \rightarrow [\mathbf{x}\ \lambda \mathbf{e}]} \quad \mathbf{x} \neq \mathbf{x'}$$

Here λ stands for :, :=, or == ; **e** stands for **t** or **c**. Notice that the <u>last</u> binding stored in the environment (thus the last binding of the expression) gains always the priority in the case of multiple bindings using the same variable name.

1.3.2 Typing expressions.

The type checking uses the type of an expression, defined by the sequent:

$\mathbf{K} \vdash_{typ} \mathbf{t} \rightarrow \mathbf{t'}$
In the environmemnt **K**, the type of **t** is **t'**.

$$\frac{\mathbf{K} \vdash_{sel} \mathbf{x} \rightarrow [\mathbf{x{:}t}]}{\mathbf{K} \vdash_{typ} \mathbf{x} \rightarrow \mathbf{t}} \qquad \frac{\mathbf{K} \vdash_{sel} \mathbf{x} \rightarrow [\mathbf{x{:=}t}] \qquad \mathbf{K} \vdash_{typ} \mathbf{t} \rightarrow \mathbf{t'}}{\mathbf{K} \vdash_{typ} \mathbf{x} \rightarrow \mathbf{t'}}$$

$$\frac{\mathbf{K[x\lambda e]} \vdash_{typ} \mathbf{t} \rightarrow \mathbf{t'}}{\mathbf{K} \vdash_{typ} [\mathbf{x\lambda e} \vdash \mathbf{t}] \rightarrow [\mathbf{x\lambda e} \vdash \mathbf{t'}]} \qquad \frac{\mathbf{K} \vdash_{typ} \mathbf{t}_1 \rightarrow \mathbf{t'}_1}{\mathbf{K} \vdash_{typ} \mathbf{t}_1(\mathbf{t}_2) \rightarrow \mathbf{t'}_1(\mathbf{t}_2)}$$

$$\frac{\mathbf{K} \vdash_{sel} \mathbf{x} \rightarrow [\mathbf{x} == [|\mathbf{bb}|]] \qquad \mathbf{K\ bb} \vdash_{typ} \mathbf{t} \rightarrow \mathbf{t'}}{\mathbf{K} \vdash_{typ} [\mathbf{import\ x} \vdash \mathbf{t}] \rightarrow [\mathbf{import\ x} \vdash \mathbf{t'}]}$$

Notice that only texts have a type, and that **primal** has no type. The typing of **[import x ⊢ t]** shows how context variables are expanded before being pushed onto the environment. This explains why environments contain only bindings, not pseudo-bindings.

1.3.3 Reductions.

The equality of two expressions is defined modulo some reductions, defined by the sequent:

$\mathbf{K} \vdash_{red} \mathbf{e} >>_{\mathbf{r}} \mathbf{e'}$ $\qquad$ $r ::= \beta \mid \delta \mid \sigma \mid d \mid s$
In the environment **K**, **e** **r**-reduces to **e'**

$$\mathbf{K} \vdash_{red} [\mathbf{x}:\mathbf{t}_1 \vdash \mathbf{t}_2](\mathbf{t}_3) >>_{\beta} [\mathbf{x}:=\mathbf{t}_3 \vdash \mathbf{t}_2]$$

$$\mathbf{K} \vdash_{red} [|\mathbf{x}:\mathbf{t}_1 ; \mathbf{bb}|] (\mathbf{t}_3) >>_{\beta} [|\mathbf{x}:=\mathbf{t}_3 ; \mathbf{bb}|]$$

$$\frac{\mathbf{K} \vdash_{sel} \mathbf{x} \rightarrow [\mathbf{x}:=\mathbf{t}]}{\mathbf{K} \vdash_{red} \mathbf{x} >>_{\delta} \mathbf{t}} \qquad \frac{\mathbf{K} \vdash_{sel} \mathbf{x} \rightarrow [\mathbf{x} == [|\mathbf{bb}|]]}{\mathbf{K} \vdash_{red} \mathbf{import\ x} >>_{\sigma} \mathbf{bb}}$$

$$\mathbf{K} \vdash_{red} [\mathbf{x}:=\mathbf{t}_1 \vdash \mathbf{t}_2] >>_{d} \mathbf{t}_2 \qquad \text{if } \mathbf{x} \text{ not free in } \mathbf{t}_2$$

$$\mathbf{K} \vdash_{red} [\mathbf{x}==\mathbf{c} \vdash \mathbf{t}] >>_{s} \mathbf{t} \qquad \text{if } \mathbf{x} \text{ not free in } \mathbf{t}$$

The β-reduction is defined in a lazy way: instead of substituting all the occurrences of **x** in $\mathbf{t_2}$ by $\mathbf{t_1}$, the operation is registered in a definition $\mathbf{x}:=\mathbf{t_1}$. The β-reduction is defined for text and context applications.

The δ-reduction expands the text definitions, while the d-reduction allows the elimination of definitions once they have been totally expanded.

The σ-reduction expands in line the importations of contexts, while the s-reduction allows the elimination of a context definition once all the importations of this context have been expanded.

The relation of reduction is the union of the relations of β, δ, σ, d, s-reductions.

1.3.4 Contractions.

An expression can <u>contract</u> into another expression via the application of several (possibly none) steps of reductions (reflexive-transitive closure of the relation of reduction). At each step the reduction is applied either to the whole expression, or to a subexpression (congruential closure). The reflexive, transitive, congruential closure of the relation of reduction is called contraction, and denoted by the sequent: $\mathbf{K} \vdash_{cont} \mathbf{e} >> \mathbf{e'}$

1.3.5 Equality.

As it was pointed out in paragraph 1.3, the use of a name free notation for variable names avoids α-conversion. Syntactical identity (denoted $\equiv$) is equivalent here to the "syntactical identity modulo α-conversion" of classical presentations of lambda-calculus.

The congruence induced on the set of expressions by the reduction relation (i.e. its reflexive, symmetric, transitive and congruential closure) is called equality, and denoted by the sequent $\mathbf{K} \vdash_{equ} \mathbf{e} = \mathbf{e'}$. Notice that the sign "=" alone (i.e. without the sequent), used in various places in this paper, does not belong to the formalisation.

The reduction relation being Church-Rosser and every valid expression being strongly normalizable [Gro91], the equality is decidable, and a straightforward decision algorithm is to compare the (unique) normal forms of the expressions:

$$\frac{\mathbf{K} \vdash_{cont} \mathbf{e} >> \mathbf{e}_1 \qquad \mathbf{K} \vdash_{cont} \mathbf{e'} >> \mathbf{e'}_1 \qquad \mathbf{e}_1 \equiv \mathbf{e'}_1}{\mathbf{K} \vdash_{equ} \mathbf{e} = \mathbf{e'}}$$

1.3.6 Validity.

The complete system of rules can be found in [Gro91]. Only the rule for text applications is given here:

$$\frac{\mathbf{K} \vdash_{typ} \mathbf{t} \rightarrow \mathbf{t}_1 \qquad \dfrac{\mathbf{K} \vdash_{val} \mathbf{t} \qquad \mathbf{K} \vdash_{val} \mathbf{t'}}{\mathbf{K} \vdash_{typ} \mathbf{t'} \rightarrow \mathbf{t'}_1} \qquad \mathbf{K} \vdash_{equ} \mathbf{t}_1 = [\mathbf{x}:\mathbf{t'}_1 \vdash \mathbf{t}_2]}{\mathbf{K} \vdash_{val} \mathbf{t}(\mathbf{t'})} \qquad (\text{t_app})$$

This rule expresses the fact that, for the application $\mathbf{t}(\mathbf{t'})$ to be valid, $\mathbf{t}$ must have a functional type $[\mathbf{x}:\mathbf{t'}_1 \vdash \mathbf{t}_2]$, and the type of the formal parameter must be equal to the type $\mathbf{t'}_1$ of the argument.

1.4 The importation mechanism.

As it was pointed out in paragraph 1.2, the pseudo-binding **import c1** allows to expand in a context **c2** the set of bindings of a context **c1**.

c1 == [| **x:primal** |]
c2 == [| **import c1; y:x** |] is equivalent to **c2** == [| **x:primal; y:x** |]

This mechanism permits the definition of theories, and their reuse in the development of new theories. Moreover, it is possible to define generic (parametric) contexts, which are instantiable at will. For instance, we can define the theory of generic equality:

sort : primal [1]

equality == [|s:**sort**; =:[s; s ⊢ **prop**]; **refl**:[x:s ⊢ x=x]; ..... |]
N : sort
0, 1 : N
equN == **equality**(**N**)
import equN -- equivalent to =:[N; N ⊢ **prop**]; **refl**:[x:N ⊢ x= x]; .....
refl(1) .....

It should be noticed that declarations play a double role in contexts. For instance, in the example above, the intended meanings of **s** and = are as follows:

- **s:sort** declares the formal parameter **s**; its semantics is exactly the one of abstraction in lambda-calculus; **s** is aimed at being instantiated by an argument (β-reduction).
- =:[s; s ⊢ **prop**] declares the operator = of the theory **equality**, which becomes visible in all contexts importing **equality**; = is not a formal parameter.

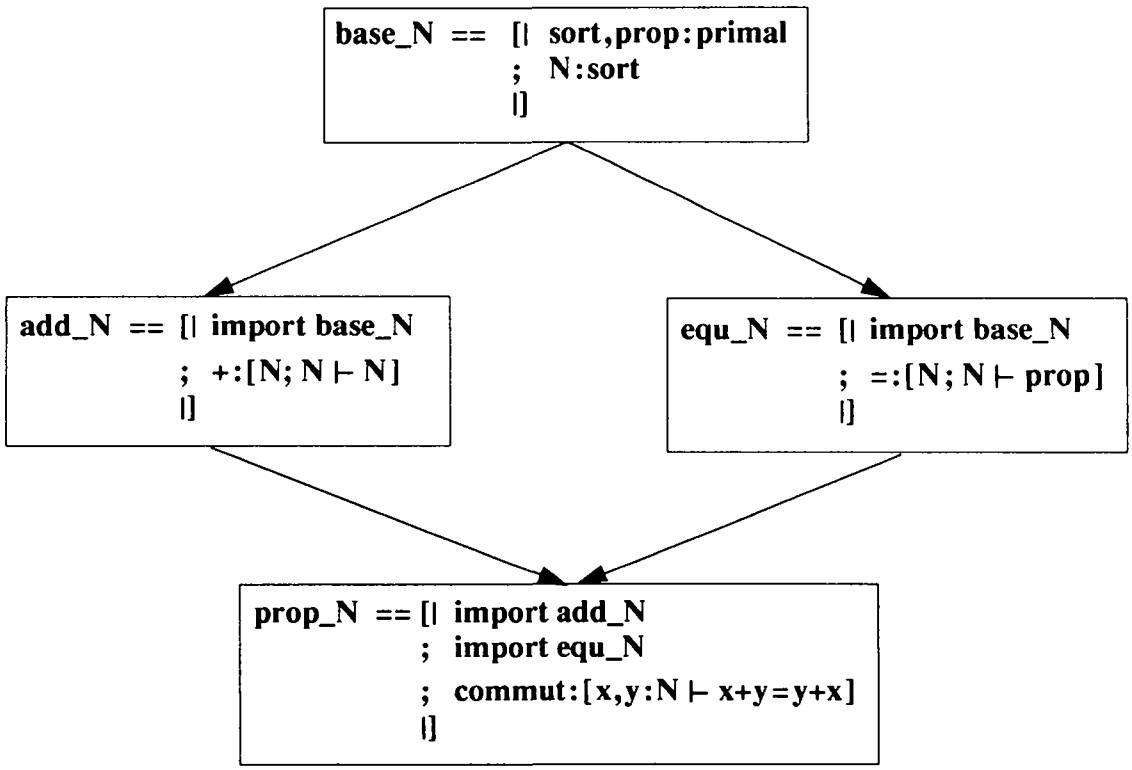
In the following, the two types of declarations are distinguished: the former is called parametric declaration, and the latter data declaration. Declarations inside texts are always parametric, as a text cannot be imported. The distinction, which is for the moment immaterial, will be made explicit in AS.

2 Sharing information between contexts.

2.1 Example.

When theories import other theories, the problem illustrated in the example below is very common:

1. In order to ease the reading of examples, we take some freedom with the syntax, neglecting square brackets and semicolons at the external level. Binary operators + and = are written in infix notation, + having priority over =. ..... is intended for the irrelevant continuation of the expression. Comments are preceded with --.



Basic properties of naturals are stated in the context **base_N**. Addition and equality theories in **N** are then independently stated in contexts **add_N** and **equ_N**, which naturally import **base_N**. Then, in order to specify properties of addition in **N**, the context **prop_N** imports **add_N** and **equ_N**. It is thus equivalent to:

prop_N ==
[| $\mathbf{sort_{(1)}, prop_{(1)}}$**:primal**
; $\mathbf{N_{(1)}}$**:**$\mathbf{sort_{(1)}}$
; **+:[**$\mathbf{N_{(1)}}$**;** $\mathbf{N_{(1)}}$ **⊢** $\mathbf{N_{(1)}}$**]**
; $\mathbf{sort_{(2)}, prop_{(2)}}$**:primal**
; $\mathbf{N_{(2)}}$**:**$\mathbf{sort_{(2)}}$
; **=:[**$\mathbf{N_{(2)}}$**;** $\mathbf{N_{(2)}}$ **⊢** $\mathbf{prop_{(2)}}$**]**
; **commut:[x,y:**$\mathbf{N_{(2)}}$ **⊢ x+y=y+x]**
|]

The declarations of **sort**, **prop**, **N** are made two times. Meta-indices **(1)** and **(2)** have been added to disambiguate the expressions. They obey to the rules of search of a binding in an environment (cf paragraph 1.3.1).

The application **x+y** is ill-typed, as the type $\mathbf{N_{(1)}}$ of the formal parameters of + is not equal to the type $\mathbf{N_{(2)}}$ of the arguments of +. Though this example may seem artificial (why decompose such a simple theory in three sub-theories?), it illustrates a rather common situation: several theories $\mathbf{th_1}$, $\mathbf{th_2}$,..., $\mathbf{th_n}$, developed in an independent way (possibly by different programmers) may use the same theory $\mathbf{th_0}$. Every time a theory $\mathbf{th_{n+p}}$ imports at least two theories among $\mathbf{th_1}$, $\mathbf{th_2}$,..., $\mathbf{th_n}$, the problem may arise.

2.2 Analysis of the problem.

The problem shown above arises from the following causes:

- AC uses the standard scoping rule of lambda-calculus: when the same variable is declared several times, the last declaration always gains priority. For instance, in

  [x:t; x:t ⊢ x], the third occurrence of **x** references the second declaration **x:t**. This rule corresponds to the following intuitive meaning: the redeclaration of a variable points out the intent not to reuse the previous one. Notice that here the redeclaration is explicitly made by the programmer.
- The importation of a context yields a textual copy of this context in the importing context, thus a redeclaration of the bindings of the imported context. This may seem well tuned to intuition, as several importations of the same context in a given context point out an explicit will of redeclaration. Unhappily, the previous example shows that the transitivity inherent to the copy mechanism induces unwished redeclarations.

Declarations explicitly made by the programmer are now called primary declarations, while those which result of the copy of a context are called secondary declarations. The above analysis shows that the problem is caused by secondary declarations. One primary declaration may induce several secondary declarations by copy. The latter are considered as different declarations by the formalism, whereas they are unique in the programmer's mind.

2.3 A first approach to the solution.

The most radical solution is to eliminate secondary declarations. For this purpose, textual copy associated with importation is replaced by access authorization to the information enclosed in a context. Then a unique version of a context exists (the one declared by the programmer), to which other contexts have access. The access authorization uses the key-word **see**, in order to underline the semantic difference with the importation mechanism. Moreover, **see** may be followed by any context, as opposed to **import** which could only be followed by a context variable.,

base_N == **[| sort,prop:primal; N:sort |]**
add_N == **[| see base_N; +:[N; N ⊢ N] |]**
equ_N == **[| see base_N; =:[N; N ⊢ prop] |]**
prop_N == **[| see add_N; see equ_N; commut:[x,y:N ⊢ x+y=y+x] |]**

There is a unique declaration of **N**, made in the context **base_N**, and made visible in the other contexts by **see** instructions. The declarations of **+**, **x** and **y** reference the same variable **N**, and the application **x+y** in the context **prop_N** is well typed.

This simple mechanism however fails in the case of parametric contexts:

ctx1 == [| s:sort [1]**; +:[s; s ⊢ s] |]**
ctx2 == [| N:sort; see ctx1(N); ..... |]
ctx3 == [| R:sort; see ctx1(R); ..... |]

1. We make a free use of previously declared identifiers like sort, prop, neglecting to declare them in each example.

The context visible from **ctx2** is **[|+:[N; N ⊢ N|]**, whereas the one visible from **ctx3** is **[|+:[R; R ⊢ R]|]**. These two contexts being different, copies of **ctx1** must be created, in order to be instantiated by **N** and **R**.

2.4 The proposed solution.

As copies of contexts are unavoidable, it is impossible to eliminate secondary declarations. So we must define criterions to decide when two declarations are equal (i.e. are the declaration of the <u>same</u> variable).

- Two primary declarations are always different, as they point out the will of the programmer to declare two different variables.
- Two secondary declarations (or a primary and a secondary one) are equal if the following conditions stand:
  - They must be copied from the same primary declaration (a primary declaration is considered as copied from itself); as a matter of fact, in this case, the programmer has made only one declaration.
  - The declared variables have the same type. This means that the possible instantiations of parameters have not modified (or have modified in the same way) their declared type.

The effect of these rules is shown in the following example:

```
M,N,P:sort
ctx1==[|s:sort; x:s; y:P|]
ctx2==[|t:sort; z:t|]
see ctx1(M)          -- equivalent to     see [|x:M; y:P|]
x1:=x
y1:=y
see ctx1(M)          -- equivalent to     see [|x:M; y:P|]
x2:=x
see ctx1(N)          -- equivalent to     see [|x:N; y:P|]
x3:=x
y3:=y
see ctx2(M)          -- equivalent to     see [|z:M|]
z4:=z
```

The following relations stand:

x1 = x2	same origin, same type
y1 = y3	same origin, same type
x1 ≠ x3	same origin, different types
x1 ≠ z4	different origins, same type

Therefore a secondary declaration must keep a trace of the primary declaration from which it is copied, in order to apply the above rules. Before defining the corresponding formalism, we show that, as opposed to what happens with ΛC, the names of declared variables can no longer be neglected, and that name-free notations (as De Bruijn indices) are no longer usable.

- Parametric declarations (in a text or a context): the name of parameters is irrelevant; the only pertinent information is their relative position. De Bruijn indices are well adapted to this situation, as they forget the name and remember the position. In the following example, **a1** = **a2**, though the parameter names **t** and **u** are different.

  a1:=[t:sort ⊢ t]

  a2:=[u:sort ⊢ u]

- Data declarations (necessarily in a context): here, the relevant information is the existence of the declaration, not its position. As a matter of fact, every primary declaration creates a new variable, but the precise position where it is created is irrelevant. In the following example, the two variables **t** are considered as different, as they appear in two distinct declarations. Thus **b1** ≠ **b2**, although they are syntactically identical. But **c1** = **c2**, as they reference the same variable **u**, declared in the context **ctx**. The use of De Bruijn indices does not allow to distinguish **b1** and **b2**, as the references to the variable **t** yields the same index.

  b1:=[see [|t:sort|] ⊢ t]

  b2:=[see [|t:sort|] ⊢ t]
  ctx==[u:sort|]

  c1:=[see ctx ⊢ u]

  c2:=[see ctx ⊢ u]

The representation of variables must allow a finer discrimination than the one possible in ΛC. It is necessary to distinguish between parametric and data declarations on one hand, and primary and secondary declarations on the other hand. A formalism allowing a uniform representation of all types of declarations is chosen. Every variable is identified by a unique name, and two distinct variables have different names. This representation has practical consequences:

- As two primary declarations always correspond to different variables, two declarations of variables with the same name must not exist in a program.
- When a secondary declaration is created (by copy), a new name of variable must be used. However, the old name must be remembered in the secondary declaration (trace of origin).
- The α-conversion rule must be explicitly used to establish the equality of two expressions containing parametric declarations.

3 The ΛS Calculus.

3.1 Syntax.

Modifications concern bindings **b** and contexts **c**.

binding:	**b**	::=	**x:t**	|	**x{x}:t**	|	**x:=t**	|	**x==c**	|	**see c**			
context:	**c**	::=	**x**	|	**[|bb|]**	|	**[|bb ⊢ bb|]**	|	**c(t)**	|	**(c)**			

In the declaration $x_1\{x_2\}:t$, x_1 is the name of the declared variable, x_2 is its origin.

- Primary data declaration: it is its own origin; thus $x_2 = x_1$.
- Secondary data declaration: $x_1\{x_2\}:t$ results of the copy of a primary declaration $x_2\{x_2\}:t'$.

In the declaration **x:t**, **x** is the name of the variable; this form is used when the origin is not relevant (e.g. for parametric declarations, and for many semantic rules).

For text and context definitions, it is not necessary to keep track of the origin, as two defined variables may always be compared by expanding them (δ or σ-reduction). The pseudo-binding **see c**, which makes the bindings enclosed in **c** visible outside, is named an access.

The structure of contexts is enriched by the construction [| **bb** ⊢ **bb** |]. We have seen the necessity to distinguish parametric and data declarations. In ΛC, this is made at the metalanguage level, as shown by the example in paragraph 1.4. When importing a context, nothing prevents a declaration from being considered at one time as a parametric declaration, at the other time as a data declaration. In ΛS, such a confusion leads to incoherences, and the distinction parametric-data must be dealt with at the language level. The construction [| **bb** |] now corresponds to non-parametric contexts.

3.2 Semantics

3.2.1 Naming of variables.

We have explained in paragraph 2.4 that there should not exist two variable declarations using the same name in a program. This constraint can obviously not be imposed to the programmer. It could not be obeyed when reusing already existing modules. Thus, the following convention will be adopted: in a program, variables are designated by identifiers which are free of constraint. The validity check is divided into two phases:

- Naming of variables: for each variable bound in the program, a name is created, which is substituted to all the occurrences of this variable identifier. Names respect the unicity constraint. During this phase, context variables are expanded if necessary, in order to insure a correct naming.
- Validity check itself: it corresponds to the one of ΛC.

The naming phase is not formalized here, as it involves rather boring technical details, but is illustrated on an example.

a==[| s:primal |]	---------->	a_1==[| $s_1\{s_1\}$:**primal** |]
see a	naming	**see** a_1
b==[| x:s ⊢ y:x |]		b_1==[| $x_1:s_1 \vdash y_1\{y_1\}:x_1$ |]
N:s		$N_1\{N_1\}:s_1$
see b(N)		see [| $x_2:s_1 \vdash y_2\{y_1\}:x_2$ |](N_1)
z:y		$z_1\{z_1\}:y_2$
z:z		$z_2\{z_2\}:z_1$

In this example, identifiers are represented by a letter, and names by an indexed letter. It can be seen that context variables are expanded only when necessary (i.e. when they are involved in an application). When copied, data declarations yield new names, but keep track of their origin. Two distinct variables get different names, even when they have the same identifier.

It should also be noticed that context expansion does not eliminate the key-word **see**, whereas in ΛC it eliminates **import**. This conforms to the announced semantics: **see c** is an access authorization to the bindings enclosed in the context **c**, but these bindings do not become bindings written at the level of the accessing context, even in case of a local copy. Particularly, **see** cannot create parameters in a text or a context.

The validity check is formalized in the following lines, with particular insistence upon the peculiarities of ΛS.

3.2.2 Environments.

$$\mathbf{K} ::= \emptyset \mid \mathbf{K}[\mathbf{x}:\mathbf{t}] \mid \mathbf{K}[\mathbf{x}\{\mathbf{x}\}:\mathbf{t}] \mid \mathbf{K}[\mathbf{x}:=\mathbf{t}] \mid \mathbf{K}[\mathbf{x}==\mathbf{c}] \mid \mathbf{K}[\mathbf{see}\ \mathbf{c}]$$

In contrast to ΛC, accesses in the form of **see c** are pushed onto the environment. As **see** is not eliminated by context expansion, it is interesting to keep inside environments the tree structuration that contexts induce on the program. This allows more sophisticated searching methods to be implemented, compared to ΛC which uses stack-like environments. For instance, the interpreter based on the ideas presented in this paper makes a search guided by incomplete pathes (represented by identifiers in dot notation) in the tree of contexts. Here, for the sake of simplicity, we define a depth-first search.

$$\mathbf{K}[\mathbf{x}\,\lambda\,\mathbf{e}] \vdash_{sel} \mathbf{x} \rightarrow [\mathbf{x}\,\lambda\,\mathbf{e}] \qquad \frac{\mathbf{K} \vdash_{sel} \mathbf{x} \rightarrow [\mathbf{x}\,\lambda\,\mathbf{e}]}{\mathbf{K}[\mathbf{x}'\,\lambda'\,\mathbf{e}'] \vdash_{sel} \mathbf{x} \rightarrow [\mathbf{x}\,\lambda\,\mathbf{e}]} \quad \mathbf{x} \neq \mathbf{x}'$$

$$\frac{\mathbf{K} \vdash_{expand} \mathbf{c} \rightarrow \mathbf{bb} \qquad \mathbf{K}\,\mathbf{bb} \vdash_{sel} \mathbf{x} \rightarrow [\mathbf{x}\,\lambda\,\mathbf{e}]}{\mathbf{K}[\mathbf{see}\ \mathbf{c}] \vdash_{sel} \mathbf{x} \rightarrow [\mathbf{x}\,\lambda\,\mathbf{e}]} \qquad \lambda \text{ stands for } :,\ :=,\ ==$$

$$\mathbf{K} \vdash_{expand} [|\,\mathbf{bb}\,|] \rightarrow \mathbf{bb} \qquad \frac{\mathbf{K} \vdash_{sel} \mathbf{x} \rightarrow [\mathbf{x}==\mathbf{c}] \qquad \mathbf{K} \vdash_{expand} \mathbf{c} \rightarrow \mathbf{bb}}{\mathbf{K} \vdash_{expand} \mathbf{x} \rightarrow \mathbf{bb}}$$

Because of the context processing made by the validity check, only two forms of pseudo-bindings can be pushed onto the environment, namely **see x** (with **x** defining a non-parametric context) and **see** [| **bb** |]. Thus the sequent $\vdash_{expand}$ takes only these two cases into account.

3.2.3 Typing expressions.

$$\frac{\mathbf{K} \vdash_{sel} \mathbf{x} \rightarrow [\mathbf{x}:\mathbf{t}]}{\mathbf{K} \vdash_{typ} \mathbf{x} \rightarrow \mathbf{t}} \qquad \frac{\mathbf{K} \vdash_{sel} \mathbf{x} \rightarrow [\mathbf{x}:=\mathbf{t}] \qquad \mathbf{K} \vdash_{typ} \mathbf{t} \rightarrow \mathbf{t}'}{\mathbf{K} \vdash_{typ} \mathbf{x} \rightarrow \mathbf{t}'}$$

$$\frac{\mathbf{K}\,\mathbf{b} \vdash_{typ} \mathbf{t} \rightarrow \mathbf{t}'}{\mathbf{K} \vdash_{typ} [\mathbf{b} \vdash \mathbf{t}] \rightarrow [\mathbf{b} \vdash \mathbf{t}']} \qquad \frac{\mathbf{K} \vdash_{typ} \mathbf{t}_1 \rightarrow \mathbf{t}'_1}{\mathbf{K} \vdash_{typ} \mathbf{t}_1(\mathbf{t}_2) \rightarrow \mathbf{t}'_1(\mathbf{t}_2)}$$

The fact that all the bindings (including accesses) are processed in an uniform way simplifies the presentation of semantics, by eliminating the special treatment of the **see** case. As in ΛC, only texts have a type, and **primal** has no type.

3.2.4 Reductions and Contractions

$$\mathbf{K} \vdash_{red} [x{:}t_1 \vdash t_2](t_3) \gg_\beta [x{:=}t_3 \vdash t_2]$$

$$\mathbf{K} \vdash_{red} [|x{:}t_1 \vdash bb|](t_3) \gg_\beta [|x{:=}t_3 \vdash bb|]$$

$$\mathbf{K} \vdash_{red} [|x{:}t_1; bb_1 \vdash bb_2|](t_3) \gg_\beta [|x{:=}t_3; bb_1 \vdash bb_2|]$$

$$\frac{\mathbf{K} \vdash_{sel} x \rightarrow [x{:=}t]}{\mathbf{K} \vdash_{red} x \gg_\delta \mathbf{copy}\ t}$$

$$\mathbf{K} \vdash_{red} [x{:=}t_1 \vdash t_2] \gg_d t_2 \qquad \text{if } x \text{ not free in } t_2$$

The β-reduction is the same as in ΛC, except for contexts, where it is only applied to parametric declarations, and not to data declarations.

The δ-reduction uses the **copy** operation, the role of which is on one hand to create new names for all the variables bound inside the copied expression, on the other hand to keep in each copied declaration the track of the primary declaration it comes from. The process is shown by an example:

copy [|x{x}:t; y{y}:x|] = [|x'{x}:t; y'{y}:x'|]
copy [|x'{x}:t; y'{y}:x'|] = [|x''{x}:t; y''{y}:x''|]

The d-reduction is exactly the same as in ΛC. There is no σ-reduction, as context variables are expanded at the time of naming. There is no reduction to eliminate context definitions and accesses, as this operation should not preserve the closure of expressions in the environment. This problem is treated at the level of conversions, where the eliminated context can be pushed onto the environment.

The contractions are defined in exactly the same way as in ΛC.

3.2.5 Conversions

The simple syntactic identity check of contracted expressions used in ΛC is no longer sufficient. The name-dependent notation imposes to use explicit α-conversion to compare parameters, and rules for comparison of data declarations have been given in paragraph 2.4. These rules define what we call <u>γ-conversion</u>. The elimination of context definitions and accesses is also realised here (s-conversion).

$\mathbf{K} \vdash_{conv} e = e'$
In the environment **K**, the expressions **e** and **e'** are convertible.

$$\mathbf{K} \vdash_{conv} \mathbf{primal} = \mathbf{primal} \qquad \mathbf{K} \vdash_{conv} \mathbf{x} = \mathbf{x} \qquad \text{(id)}$$

$$\frac{\mathbf{K} \vdash_{conv} \mathbf{t}_1 = \mathbf{t'}_1 \qquad \vdash_{name} \mathbf{x''} \qquad \mathbf{K}\,[\mathbf{x}\{\mathbf{x''}\}:\mathbf{t}_1]\,[\mathbf{x'}\{\mathbf{x''}\}:\mathbf{t'}_1] \vdash_{conv} \mathbf{t}_2 = \mathbf{t'}_2}{\mathbf{K} \vdash_{conv} [\mathbf{x}:\mathbf{t}_1 \vdash \mathbf{t}_2] = [\mathbf{x'}:\mathbf{t'}_1 \vdash \mathbf{t'}_2]} \qquad (\alpha)$$

$$\frac{\mathbf{K} \vdash_{sel} \mathbf{x} \rightarrow [\mathbf{x}\{\mathbf{x''}\}:\mathbf{t}] \qquad \mathbf{K} \vdash_{sel} \mathbf{x'} \rightarrow [\mathbf{x'}\{\mathbf{x''}\}:\mathbf{t'}] \qquad \mathbf{K} \vdash_{equ} \mathbf{t} = \mathbf{t'}}{\mathbf{K} \vdash_{conv} \mathbf{x} = \mathbf{x'}} \qquad (\alpha\text{-}\gamma)$$

$$\frac{\mathbf{K}\,[\mathbf{x}==\mathbf{c}] \vdash_{conv} \mathbf{t} = \mathbf{t'}}{\mathbf{K} \vdash_{conv} [\mathbf{x}==\mathbf{c} \vdash \mathbf{t}] = \mathbf{t'}} \qquad \frac{\mathbf{K}\,[\mathbf{x'}==\mathbf{c'}] \vdash_{conv} \mathbf{t} = \mathbf{t'}}{\mathbf{K} \vdash_{conv} \mathbf{t} = [\mathbf{x'}==\mathbf{c'} \vdash \mathbf{t'}]} \qquad (s_{==})$$

$$\frac{\mathbf{K}\,[\mathbf{see}\ \mathbf{c}] \vdash_{conv} \mathbf{t} = \mathbf{t'}}{\mathbf{K} \vdash_{conv} [\mathbf{see}\ \mathbf{c} \vdash \mathbf{t}] = \mathbf{t'}} \qquad \frac{\mathbf{K}\,[\mathbf{see}\ \mathbf{c'}] \vdash_{conv} \mathbf{t} = \mathbf{t'}}{\mathbf{K} \vdash_{conv} \mathbf{t} = [\mathbf{see}\ \mathbf{c'} \vdash \mathbf{t'}]} \qquad (s_{see})$$

$$\frac{\mathbf{K} \vdash_{conv} \mathbf{t}_1 = \mathbf{t'}_1 \qquad \mathbf{K} \vdash_{conv} \mathbf{t}_2 = \mathbf{t'}_2}{\mathbf{K} \vdash_{conv} \mathbf{t}_1\,(\mathbf{t}_2) = \mathbf{t'}_1\,(\mathbf{t'}_2)} \qquad \text{(app)}$$

Notice that α-conversion does not rely on the usual operation of substitution, but uses the environment: the rule (α) memorizes in the environment the fact that the two parameters are α-converted, thanks to the sequent $\vdash_{name}$ which builds new variable names; this information is used later by the rule (α–γ) to check the α-convertibility of references to these parameters. The same rule (α–γ) checks also the γ-convertibility of references to data declarations. For s-conversions, notice how context definitions and accesses are stored in the environment once eliminated from expressions.

Only rules for texts have been given. Analogous rules stand for contexts.

3.2.6 Equality

The reduction has the same properties as in ΛC. The algorithm for equality checking is similar, but relies on convertibility instead of syntactic identity.

$$\frac{\mathbf{K} \vdash_{cont} \mathbf{e} >> \mathbf{e}_1 \qquad \mathbf{K} \vdash_{cont} \mathbf{e'} >> \mathbf{e'}_1 \qquad \mathbf{K} \vdash_{conv} \mathbf{e}_1 = \mathbf{e'}_1}{\mathbf{K} \vdash_{equ} \mathbf{e} = \mathbf{e'}}$$

3.2.7 Validity.

$\mathbf{K} \vdash_{valid} \mathbf{e}$	The expression **e** is valid in the environment **K**.

λ stands for :, :=, ==

$$\mathbf{K} \vdash_{val} \mathbf{primal} \qquad \text{(prim)} \qquad\qquad \frac{\mathbf{K} \vdash_{sel} \mathbf{x} \rightarrow [\mathbf{x}\,\lambda\,\mathbf{e}]}{\mathbf{K} \vdash_{val} \mathbf{x}} \qquad \text{(var)}$$

$$\frac{K \vdash_{val} t}{K \vdash_{val} x{:}t} \qquad \frac{K \vdash_{val} t}{K \vdash_{val} x{:=}t} \qquad \text{(t_bind)}$$

$$\frac{K \vdash_{val} c}{K \vdash_{val} x{==}c} \qquad \frac{K \vdash_{val} c \qquad K \vdash_{equ} c = [|\,bb\,|]}{K \vdash_{val} \textbf{see}\ c} \qquad \text{(c_bind)}$$

$$\frac{K \vdash_{val} b \qquad K\,b \vdash_{val} bb}{K \vdash_{val} b;bb} \quad \text{(seq_bind)} \qquad \frac{K \vdash_{val} bb}{K \vdash_{val} [|\,bb\,|]} \quad \text{(ctx)}$$

$$\frac{K \vdash_{val} b \qquad K\,b \vdash_{val} t}{K \vdash_{val} [b \vdash t]} \quad \text{(t_abs)} \qquad \frac{K \vdash_{val} bb_1 \qquad K\,bb_1 \vdash_{val} bb_2}{K \vdash_{val} [|\,bb_1 \vdash bb_2\,|]} \quad \text{(c_abs)}$$

$$\frac{\begin{array}{c} K \vdash_{val} t_1 \qquad K \vdash_{val} t_2 \\ K \vdash_{typ} t_1 \rightarrow t'_1 \qquad K \vdash_{typ} t_2 \rightarrow t'_2 \\ K \vdash_{equ} t'_1 = [x{:}t'_2 \vdash t'_3] \end{array}}{K \vdash_{val} t_1(t_2)} \qquad \text{(t_app)}$$

$$\frac{\begin{array}{c} K \vdash_{val} c \qquad K \vdash_{val} t \qquad K \vdash_{typ} t \rightarrow t' \\ K \vdash_{equ} c = [|\,x{:}t'\ bbb \vdash bb\,|] \end{array}}{K \vdash_{val} c(t)} \qquad bbb ::= \emptyset \ \mid\ ;bb' \qquad \text{(c_app)}$$

Concerning abstractions and applications, similar rules stand for texts and contexts. The rule (c_app) performs another operation, which is not expressed in the present formalism, for the sake of simplicity: after its validity has been checked, the application **c(t)** is β-reduced. This is necessary for data declarations to be seen with their real type (i.e. possibly instantiated by **t**) when accessed by **see c(t)**. The rule (c_bind) shows that only non-parametric contexts can be accessed via **see**. Thus, a parametric context must be totally instantiated before being accessed, which is coherent with the distinction we introduced between parametric and data declarations.

So, after validity checking, **see** can be followed neither by a context application nor by a parametric context. As every **see** pushed onto the environment has previously been checked, only **see x** and **see [| bb |]** can be pushed onto the environment, as stated in paragraph 3.2.2.

Conclusion

This paper is aimed at introducing in a typed lambda-calculus two well-known concepts in programming languages: structuration and modularity. A fundamental operation of lambda-calculus is abstraction, which introduces a natural structuration, bound to the scope of declarations. On the other hand, modularity needs an extension of the calculus. The simplest extension, which is to internalise the notion of context (seen as a sequence of abstractions) leads to difficulties

when several contexts must share the same declarations. We show how the introduction of a duality between parametric and data declarations solves these difficulties. This leads us to define a new conversion: the γ-conversion (for data), dual of the α-conversion (for parameters).

The ideas presented in this paper have been used to extend the Deva language [Sin&al89], itself based on the ΛC calculus. Some questions had to be considered, which were neglected in the definition of the calculus. Among those are the access to distinct variables with the same identifier (solved by a dot notation), the possibility to share or not data between contexts, and the persistence of modules which, contrary to contexts, must be accessible to other programs than the one where they are defined. We think that the solutions presented here for modularity and sharing constitute a partial answer to the challenge set by the size of completely formalized proofs and program developments.

References

[Deb72]: N. G. de Bruijn.
Lambda Calculus notations with nameless dummies, a tool for automatic formula manipulation, with an application to the Church-Rosser theorem.
Indigationes Mathematicae 34 (1972) 381-392.

[Deb80]: N. G. de Bruijn.
A survey of the project Automath.
In J. P. Seldin and J. R. Hindley, editors, to H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, pages 579-606.
Academic Press, 1980.

[Gro91]: Ph. de Groote.
Définition et propriétés d'un métacalcul de représentation de théories.
Thèse Université de Louvain. Février 1991.

[How80]: W. A. Howard.
The formulae-as-types notion of construction.
In J. P. Seldin and J. R. Hindley, editors, to H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, pages 479-490.
Academic Press, 1980.

[Mac84]: D. B. MacQueen.
Modules for standard ML.
In Proc Symposium on Lisp and Functional Programming, Austin. Texas. August 6-8 1984, pp 198-207. ACM New York.

[Ned73]: R. P. Nederpelt.
Strong Normalization in a typed lambda calculus with lambda structured types.
PhD thesis, Technische hogeschool Eindhoven, 1973.

[Sin&al89]: M. Sintzoff, M. Weber, Ph de Groote, J. Cazin
Definition 1.1 of the generic development language Deva
ToolUse.TD.deva1.1.89. Internal Ref.:RR 89-xx . CERT GMD UCL.
November 1989

Université Paul Sabatier
118, route de Narbonne - 31062 Toulouse

Unité de Recherche INRIA Lorraine Technopôle de Nancy-Brabois - Campus Scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 Villers Les Nancy Cedex (France)
Antenne de Metz Tchnopôle de Metz 2000 - Cescom - 4, rue Marconi - 57070 Metz (France)
Unité de Recherche INRIA Rennes IRISA, Campus Universitaire de Beaulieu 35042 Rennes Cedex (France)
Unité de Recherche INRIA Rhône-Alpes 46, avenue Félix Viallet - 38031 Grenoble Cedex (France)
Unité de Recherche INRIA Rocquencourt Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)
Unité de Recherche INRIA Sophia Antipolis 2004, route des Lucioles - B.P. 93 - 06902 Sophia Antipolis Cedex (France)

ÉDITEUR
INRIA - Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)

ISSN 0249 - 6399

RR.1903