

Sort Abstraction for Static Analyses of Mobile Processes

Frédéric Prost

Leibniz-IMAG, 46 Av. Félix Viallet,
38 000 Grenoble, France
Frederic.Prost@imag.fr

1 INTRODUCTION

The non-interference type based analysis and its closely related analyses (dead code elimination, program slicing, confidentiality, strictness, etc.) have been thoroughly studied in many programming paradigms. The work on this area has been initiated with functional programming, see e.g. [HR98, Pro00, ABHR99], as well as imperative programming, e.g. [SV98]. Following this idea, similar approaches have been studied for process algebras, see e.g. [Aba97, HR00, YH00, HVY00, Kob03]).

Nevertheless, works on process algebras are hampered by the fact that there is no canonical notion of types. Moreover, in most typing systems of the literature one has the following kind of rule:

$$\frac{\Gamma \vdash P : \tau \quad \Gamma \vdash Q : \tau}{\Gamma \vdash P \parallel Q : \tau}$$

that is to say: if both processes P, Q are of the same type τ under a common context Γ , then they can be put in parallel obtaining a new term of type τ . The problem about this kind of typing rule is that the type of the resulting term is the same than the one of its components. Compare with functional programming where each syntactic structure has an effect on types: λ -abstractions build arrow types, and applications eliminate arrow types. This lack of precision hampers accurate type-based analyses. Indeed, there is no way to represent, using types, information flows between P and Q . Hence the need of a more precise typing discipline for process algebras.

This work is an attempt to provide such a typing system. We consider a calculus including functional features (λ -terms) as well as communications viewed as addressed resources. We apply to this system known techniques

from the functional programming community to use types to perform non-interference analysis. We consider the adaptation of [Pro00], in which type based non-interference analysis is canonically defined for pure type systems, in a concurrent context. This non-interference analysis can then be used for various analyses including secure information flow analysis.

2 A λ -CALCULUS WITH ADRESSED RESOURCES

λ_{ar} is a modified version of Boudol’s blue-calculus [Bou97]. The first modification is about reductions: resource fetching, i.e. replacement of an address by its content, can occur anywhere in a term unlike in the blue-calculus where it can only occur in a head variable. We consider more or less resource fetching as a kind of β -reduction where there is no binder and where there is concurrency with respect to the argument. The second modification deals with Parallel operators: We consider two different parallel operators. One aims at the parallelism between two processes, whereas the second is used to represent parallelism between data and processes.

There are two different classes of terms in λ_{ar} , terms and addressed resources. They are inductively defined as follows:

t	$::=$	x, a	variables, addresses
		$(t \ t)$	application
		$\lambda x.t$	abstraction
		$t \parallel t$	process parallel
		$\nu a(t)$	new addresses
		$t \mid s$	process and address resource1
		$s \mid t$	process and address resource2
s	$::=$	$\langle a \Leftarrow t \rangle$	single resource
		$\langle a = t \rangle$	multiple resource
		$s \mid s$	resource parallel

An addressed resource is a term defined at a given address. In $\langle a \Leftarrow t \rangle$, a is the *address* and t the *resource*. The resource can be single or multiple. A single resource is discarded when used, whereas a multiple resource can be used as enough as necessary. One may think at the linear implication [Gir87] vs classical implication. Addressed resources may be packed together with “|” operator. Addressed may be viewed as aliases or more intuitively as hyperlinks and addressed resources as web pages.

The only λ_{ar} variable binders are λ and ν . In $\langle a \Leftarrow t \rangle$, for instance, name a is not bound. *Free names* are defined accordingly to these binders.

A context is a term written using the same syntax as for λ_{ar} -terms, plus a special constant \square , the hole. We use capital letters A, B, C, \dots to denote contexts. A context $C[\]$ can be filled with a λ_{ar} -term t . $C[t]$ is $C[\]$ where \square has been replaced with t . Notice that t free variables may become bounded in $C[t]$.

We define a structural equivalence \equiv between λ_{ar} -terms. This equivalence allows us to see terms as a “chemical solution” à la [BB92], where processes and addressed resources may move around and interact together. Structural equivalence rules are given in Figure 1.

$t \parallel u \equiv u \parallel t$ $t \mid s \equiv s \mid t$	$s_1 \mid s_2 \equiv s_2 \mid s_1$	commutativity
$(t \parallel u) \parallel r \equiv t \parallel (u \parallel r)$ $(s_1 \mid s_2) \mid t \equiv s_1 \mid (s_2 \mid t)$	$(t \parallel u) \mid s \equiv t \parallel (u \mid s)$	associativity
$\langle a = t \rangle \equiv \langle a = t \rangle \mid \langle a \Leftarrow t \rangle$		duplication
$va(t) \parallel u \equiv va(t \parallel u)$ $(a \notin \text{fn}(u) \cup \text{fn}(s))$	$va(t) \mid s \equiv va(t \mid s)$	scope migration
$t \equiv u \implies C[t] \equiv C[u]$		

FIGURE 1. structural equivalence

Commutativity, associativity and scope migration structural rules allow us to write any λ_{ar} -term in a canonical way. We call *agents* the terms defined by the following grammar:

$$\text{Ag} ::= a \mid x \mid \lambda x.t \mid (t \ t)$$

where t is any λ_{ar} -term. For instance term $(\lambda x.x \ (t \parallel u))$ is an agent, while term $t \parallel (f \ \lambda x.(x \ x))$ is not. The reader can notice that agents have the shape of λ -terms but are not λ -terms.

Fact 1 (Canonical Form) *Any term t of λ_{ar} is structurally equivalent with*

a term of the following canonical form:

$$\begin{array}{ll}
\mathbf{v}a_1, \dots, a_n & \\
(t_1 \parallel \dots \parallel t_p) & \text{agents} \\
| \langle b_1 \Leftarrow v_1 \rangle | \dots | \langle b_q \Leftarrow v_q \rangle & \text{single resources} \\
| \langle b_1 = v_1 \rangle | \dots | \langle b_q = v_q \rangle & \text{multiple resources}
\end{array}$$

There are two kinds of reduction in λ_{ar} : β -reductions and substitution of addresses by a resource defined at this address, also called, following [Bou97], *resource fetching*.

Definition 1 (Reduction rules)

$$\begin{array}{ll}
(\lambda x.t \ u) & \rightarrow_{\beta} \ u\{x := t\} \\
t \mid \langle a \Leftarrow u \rangle & \rightarrow_{\rho} \ t\{a := u\}
\end{array}$$

for \rightarrow_{ρ} we require $a \in \text{fn}(t)$ and t an agent. We write \rightarrow for $\rightarrow_{\beta} \cup \rightarrow_{\rho}$. We contextually close \rightarrow by a context rule, and extend it to a congruence:

$$\begin{array}{ll}
t \rightarrow t' & \Longrightarrow \ C[t] \rightarrow C[t'] \\
t \rightarrow t' \text{ and } t \equiv u & \Longrightarrow \ u \rightarrow t'
\end{array}$$

Resource fetching, as well as parallel operators (which are commutative) are the main reasons of the more expressive power of λ_{ar} w.r.t. λ -calculus.

The first noticeable property of β and ρ -reductions is that none of them normalizes. It is well-known (see for instance [Bar84]) that the untyped λ -calculus is not strongly normalizing. On the other hand consider the term:

$$\Omega = u \mid \langle u = u \rangle$$

Ω infinitely ρ -reduces to itself, hence ρ -reduction is not normalizing.

Another property is the possibility of the non-deterministic choice encoding:

$$c \mid \langle c \Leftarrow t \rangle \mid \langle c \Leftarrow u \rangle \tag{1}$$

with $c \notin \text{fn}(t) \cup \text{fn}(u)$, this term can be reduced either to $t \mid \langle c \Leftarrow u \rangle$ or $u \mid \langle c \Leftarrow t \rangle$. It is easy to derive an encoding of a non-deterministic choice operator as:

$$\oplus \equiv \lambda x, y. \mathbf{v}c (c \mid \langle c \Leftarrow x \rangle \mid \langle c \Leftarrow y \rangle)$$

which is not possible to define in λ -calculus. At this point one may notice, that there is another way to have non-determinism. Consider the following term:

$$t \equiv t_1 \parallel t_2 \parallel \dots \parallel t_n \mid \langle a \Leftarrow v \rangle \tag{2}$$

where $a \in \text{fn}(t_i)$ for $i \in \{1, \dots, n\}$. And let t'_i denotes the term:

$$t'_i \equiv t_1 \parallel t_2 \parallel \dots \parallel t_i[a := v] \parallel \dots \parallel t_n$$

for all $i \in \{1, \dots, n\}$, $t \rightarrow t'_i$.

Those two forms of non-determinism, the one coming from the commutativity of \parallel and the other coming from the commutativity of $|$ are different. One is based on the multiplicity of producers and the other one on the multiplicity of consumers. The non-determinism exhibited in (2) cope well with the intuition of inter-leaving. n processes run in parallel, thus we have no way to know which one will be the first to have access to resource v located at address a . On the other hand, non-determinism of (1) is of another nature: there are two concurrent resources located at a same *address* and it is up to the process to choose which definition will be used. This non-determinism is not inherent to concurrent systems.

In λ_{arc} it is possible to model communications in a π -calculus style. Consider the following term:

$$\langle a \Leftarrow \lambda x.t \rangle | (a \ v) \rightarrow_{\rho} (\lambda x.t \ v) \rightarrow_{\beta} t\{x := v\}$$

It mimics the π -calculus communication:

$$a?(x).t \parallel a!\langle v \rangle \rightarrow t\{x := v\}$$

where $a?(x).t$ is the process that receives a value v on channel a and then behaves like $t\{x := v\}$, and where $a!\langle v \rangle$ is the process emitting value v on channel a . Nevertheless, in the context of the π -calculus only channel names can be emitted, in spite of arbitrary terms in λ_{arc} .

These simple examples, although being minimal, illustrates interesting λ_{arc} features. The first one is a justification of the two different parallel operators. We have seen that it directly leads to different forms of non-determinism. It suggests that there is more than just a syntactic distinction between parallel operators. The second one is that λ_{arc} is somewhat finer than π -calculus in the sense that a π -calculus single step reduction is interpreted by a ρ and a β -reduction. The ρ -reduction represents the process of downloading distant code while β -reduction represents parameter passing. Those two steps, communication and binding of a variable to a value in the body of a process, are mixed together in π -calculus whereas they are clearly identified in λ_{arc} .

3 TYPING SYSTEM

λ_{at} is a very rich calculus. In this section we propose a typing discipline that limits the number of well formed terms. It gives some control over acceptable programs. Types may also be seen as skeletons over which it is possible to perform static analyses by annotating them.

λ_{at} types are standard λ -calculus types (ground types and arrow types), with the addition of a parallel type: $\text{Pa}(\sigma, \tau)$.

Definition 2 (Types) *Types are inductively defined by the following grammar:*

$$\begin{array}{lcl} \tau & ::= & b \quad \textit{Base type} \\ & | & \circ \quad \textit{Store type} \\ & | & \tau \rightarrow \tau \quad \textit{Arrow type} \\ & | & \text{Pa}(\tau, \tau) \quad \textit{Parallel type} \end{array}$$

Because of structural equivalence on terms, we define a related equivalence over types. The relation $\equiv_{\mathbb{T}}$ between two types is the smallest reflexive, symmetric and transitive relation containing the relation R defined by: let τ, σ, γ be in \mathbb{T} , then $R(\text{Pa}(\tau, \sigma), \text{Pa}(\sigma, \tau))$ and, $R(\text{Pa}(\tau, \text{Pa}(\sigma, \gamma)), \text{Pa}(\text{Pa}(\tau, \sigma), \gamma))$. In the rest of this paper we work modulo $\equiv_{\mathbb{T}}$ equivalence. That is, when we write $\tau = \sigma$ we intend $\tau \equiv_{\mathbb{T}} \sigma$.

We extend the syntax of λ_{at} in order to have type annotations in terms. We develop a typing system à la Church: we add type annotations in binders. More formally:

$$t ::= \dots \mid \nu a:\tau(t) \mid \lambda x:\tau.t \mid \dots$$

other terms remain unchanged.

Type contexts are lists of pairs assigning a type to a variable:

$$\Gamma ::= \langle \rangle \mid \Gamma, x : \tau$$

where $\langle \rangle$ is the empty list. A variable can only occur once in a context.

The type system deals with judgments of the form $\Gamma \vdash t : \tau$. Deduction rules are given in figure 2.

A peculiarity of this type system is that types follow the shape of terms in way close to the one of usual functional programming. To our knowledge it is a first time that a typing has been done for parallel constructs. In most process algebras of typing systems one has the rule:

$$\frac{\Gamma \vdash t : \tau \quad \Gamma \vdash u : \tau}{\Gamma \vdash t \mid u : \tau}$$

$$\begin{array}{c}
[HYP] \frac{}{\Gamma \vdash x : \tau} (x : \tau \in \Gamma) \qquad [NEW] \frac{\Gamma, a : \sigma \vdash t : \tau}{\Gamma \vdash \nu a : \sigma(t) : \tau} \\
[APP] \frac{\Gamma \vdash t : \sigma \rightarrow \tau \quad \Gamma \vdash u : \sigma}{\Gamma \vdash (t \ u) : \tau} \qquad [ABS] \frac{\Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash \lambda x : \sigma. t : \sigma \rightarrow \tau} \\
[STO1] \frac{\Gamma \vdash t : \tau}{\Gamma \vdash \langle a \Leftarrow t \rangle : \circ} (a : \tau \in \Gamma) \qquad [STO2] \frac{\Gamma \vdash t : \tau}{\Gamma \vdash \langle a = t \rangle : \circ} (a : \tau \in \Gamma) \\
[PPAR] \frac{\Gamma \vdash t : \tau \quad \Gamma \vdash u : \sigma}{\Gamma \vdash t \parallel u : \text{Pa}(\tau, \sigma)} (\tau, \sigma \neq \circ) \\
[PD1] \frac{\Gamma \vdash s : \circ \quad \Gamma \vdash t : \tau}{\Gamma \vdash s \mid t : \tau} \\
[PD2] \frac{\Gamma \vdash t : \tau \quad \Gamma \vdash s : \circ}{\Gamma \vdash t \mid s : \tau} \\
[DPAR] \frac{\Gamma \vdash s_1 : \circ \quad \Gamma \vdash s_2 : \circ}{\Gamma \vdash s_1 \mid s_2 : \circ}
\end{array}$$

FIGURE 2. λ_{ar} typing rules

therefore in such system it is difficult to analyse interaction between t and u : they are of the same type. In π -calculus it is not possible to define such types. Indeed the number of parallel processes may change at any communication step: the shape of terms evolves during computation. If we combine to this the inherent non-determinism of process algebras there is no way to enforce subject reduction. In λ_{ar} , we have made clear the distinction between several concepts merged together in π -calculus.

Standard type properties are valid in typed λ_{ar} . The first one is the unicity of types. For a given well formed term there is only one type w.r.t. a given environment.

Lemma 1 (Type unicity) *If $\Gamma \vdash t : \tau$ and $\Gamma \vdash t : \tau'$ then $\tau = \tau'$.*

Theorem 1 (Structural equivalence) *$\Gamma \vdash t : \tau$ and $t \equiv u$ implies $\Gamma \vdash u : \tau$.*

Lemma 2 (Context weakening) *$\Gamma \vdash t : \tau$ and $x \notin \Gamma$ imply $\Gamma, x : \sigma \vdash t : \tau$.*

Lemma 3 (Substitution Lemma) $\Gamma \vdash x : \tau$, $\Gamma \vdash u : \tau$ and $\Gamma \vdash t : \sigma$ imply $\Gamma \vdash t\{x := u\} : \sigma$.

Theorem 2 (Subject reduction) $\Gamma \vdash t : \tau$ and $t \rightarrow t'$ imply $\Gamma \vdash t' : \tau$.

The proof of subject reduction is very simple for λ_{at} w.r.t. other concurrent typed systems. It comes from the fact that in λ_{at} communications are seen like non-deterministic redexes and as in λ calculus we insure the fact that variables are replaced by terms of same type (rules *STO1*, *STO2* and *ABS*).

We now give some simple examples illustrating the control offered by typing discipline over λ_{at} terms. Typing limits the number of well formed terms.

The first thing to notice is that typed λ_{at} -terms do not normalize. Indeed, it is possible to type term Ω defined above. For any type σ , $u : \sigma \vdash \langle u = u \rangle : \circ$, and $u : \sigma \vdash u : \sigma$, thus by rule *PD2* we find $u : \sigma \vdash \Omega : \sigma$.

Consider the following λ_{at} term $t = u \mid \langle u = u \parallel u \rangle$. This term reduces for ever in this way:

$$\begin{aligned} t &\rightarrow u \parallel u \mid \langle u = u \parallel u \rangle \\ &\rightarrow u \parallel u \parallel u \mid \langle u = u \parallel u \rangle \\ &\rightarrow \dots \end{aligned}$$

Term t cannot possibly be well typed in our typing system. Indeed, let τ be the type of u . Then $u \parallel u$ is of type $\text{Pa}(\tau, \tau)$ (because of rule *PPAR*) which is different from τ . Hence $\langle u = u \parallel u \rangle$ cannot be well typed: rule *STO2* forces the fact that the address and the resource must have the same type. This simple example shows how types remain stable through reductions.

4 NON-INTERFERENCE ANALYSIS

In this section we deepen aims and motivations behind λ_{at} and precisely explain in which direction further research based on λ_{at} are going. We give a first result of non-interference for λ_{at} .

Non-interference is a key property for program analysis [ABHR99]. It addresses the problem of program dependency: how portions of programs are related to each other. This problem is central for settings such as secrecy, partial evaluation, program slicing, dead code analysis etc.

In [Pro00] we have presented a generic mechanism to modify standard functional type systems in order to perform type-based analyses. The idea is that analysing programs by the use of types amounts to be able to abstract

terms over sorts. Sorts are, roughly speaking, types of types. For instance $\text{Int} : *$ means that type of integers Int is of sort $*$. In [Pro00] we consider two sorts \top and \perp . Therefore there are two different types for integers: $\text{Int}^\top : \top$ and $\text{Int}^\perp : \perp$. Moreover a notion of variable sort is introduced, $\text{Int} : \alpha$ meaning that Int is of variable sort α . That is α can be either \top or \perp . Abstraction and application relatively to this kind of variable are defined. It is possible to prove that parts typed with types of sort \top and parts typed with types of sorts \perp are independent.

Following this line, we define an extension of λ_{arr} , $\lambda_{\text{arr}}\mathcal{E}$, where it is possible to abstract over sorts. Sorts are roughly the type of types, and are defined by $\mathfrak{s} ::= \top \mid \perp \mid \alpha$, where α represents sort variables.

We modify the typed version of λ_{arr} : sorts may occur in typed terms. In the line of [Pro00], we introduce the possibility to explicitly deal with sorts. Extended types are defined by:

$$\tau := b^{\mathfrak{s}} \mid \circ \mid \tau \rightarrow \tau \mid \text{Pa}(\tau, \tau) \mid \forall \alpha. \tau$$

Beside the definition of extended types (polymorphism over sorts), we have added annotations on base types. Those annotations will denote the sort of the corresponding base type. The most intuitive way to see it is to imagine that there are two ways to classify data: private and public. This approach is the one advocated in seminal paper [Aba97].

Associated with these extended types are extended contexts, contexts are list of either pairs of term variable, types or sort variables.

$$\Gamma \vdash \langle \rangle \mid \Gamma, x : \tau \mid \Gamma, \alpha$$

The reader has to be aware that now the order of declarations is significant unlike contexts for simple type system. Indeed now types may depend on sort variables consider the following context: $\Gamma = \alpha, \tau : \alpha, x : \tau$. Therefore in addition to term typing rules we add rules to ensure well formed contexts and types. Statement $\Gamma \vdash$ means that Γ is a valid context and $\Gamma \vdash \tau$ means that τ is a valid type under environment Γ . Statement $\Gamma \vdash \mathfrak{s}$ means that \mathfrak{s} is a valid sort under environment Γ , i.e. \mathfrak{s} occurs in Γ or $\mathfrak{s} = \top$ or \perp . Finally, we extended terms typing rules $\Gamma \vdash t : \tau$ for the new syntactical constructs. These statements are mutually inductively defined in fig. 3. We only give the two new rules for terms, abstraction of sorts relatively to terms and application of a term to a sort, other rules remain unchanged from the ones given in fig. 2.

Properties stated in 2 for λ_{arr} still hold in $\lambda_{\text{arr}}\mathcal{E}$.

Roughly, the introduction of \perp and \top sorts have produced two versions of λ_{arr} . In one version everything is annotated with \top , and on the other one

• Contexts:	$[CtxTyp] \frac{\Gamma \vdash \Gamma \vdash \tau (x \notin \Gamma)}{\Gamma, x : \tau \vdash}$	$[CtxSor] \frac{\Gamma \vdash}{\Gamma, \alpha \vdash} (\alpha \notin \Gamma)$	
• Sorts:	$[Sor\top] \frac{\Gamma \vdash}{\Gamma \vdash \top}$	$[Sor\perp] \frac{\Gamma \vdash}{\Gamma \vdash \perp}$	$[SorVar] \frac{\Gamma \vdash}{\Gamma \vdash \alpha} (\alpha \in \Gamma)$
• Types:	$[TpBase] \frac{\Gamma \vdash \mathfrak{s}}{\Gamma \vdash b^{\mathfrak{s}}}$	$[Tp\rightarrow] \frac{\Gamma \vdash \tau \quad \Gamma \vdash \sigma}{\Gamma \vdash \tau \rightarrow \sigma}$	$[TpPar] \frac{\Gamma \vdash \tau \quad \Gamma \vdash \sigma}{\Gamma \vdash Pa(\tau, \sigma)}$
	$[Tp\forall] \frac{\Gamma, \alpha \vdash \tau}{\Gamma \vdash \forall \alpha. \tau}$		
• Terms:	$[ABSSor] \frac{\Gamma, \alpha \vdash t : \tau}{\Gamma \vdash (\lambda \alpha. t) : (\forall \alpha. \tau)}$	$[APPSor] \frac{\Gamma \vdash t : \forall \alpha. \tau \quad \Gamma \vdash \mathfrak{s}}{\Gamma \vdash (t \ \mathfrak{s}) : (\tau\{\alpha := \mathfrak{s}\})}$	

FIGURE 3. $\lambda_{\text{atv}}\mathcal{E}$ typing rules

\perp is the only annotation. In facts both versions are mixed together since it is possible to build terms with sub-terms having types of sort \top and of sort \perp . Consider under context $f : \text{Int}^{\perp} \rightarrow \text{Int}^{\top}, x : \text{Int}^{\perp}$ the term:

$$(f \ x) \mid \langle x \Leftarrow 5^{\perp} \rangle \mid \langle f \Leftarrow \lambda y : \text{Int}^{\perp}. 4^{\top} \rangle$$

We define Γ/\mathfrak{s} -saturated types, terms and resources as follows:

Type τ is a Γ/\mathfrak{s} -saturated type if either: $\tau = b^{\mathfrak{s}}$; or $\tau = \tau_1 \rightarrow \tau_2$ or $Pa(\tau_1, \tau_2)$ and τ_1, τ_2 are Γ/\mathfrak{s} -saturated types; or $\tau = \forall \alpha. \tau_1$ and $\tau_1\{\alpha := \mathfrak{s}\}$ is a Γ/\mathfrak{s} -saturated type.

t is Γ/\mathfrak{s} -saturated term if either:

$\Gamma \vdash t : b^{\mathfrak{s}}$; or $t = x$ and $\Gamma \vdash x : \tau$ and τ is a Γ/\mathfrak{s} -saturated type; or $t = (t_1 \ t_2)$ or $t_1 \parallel t_2$ and t_1, t_2 are Γ/\mathfrak{s} -saturated terms; or $t = \lambda x : \tau. t_1$ or $\nu x : \tau(t_1)$ and τ is a Γ/\mathfrak{s} -saturated type and t_1 is a $\Gamma, x : \tau/\mathfrak{s}$ -saturated term; or $t = t \mid \mathfrak{s}$ or $t = \mathfrak{s} \mid t$, and t is a Γ/\mathfrak{s} -saturated type and \mathfrak{s} a Γ/\mathfrak{s} -saturated resource.

\mathfrak{s} is Γ/\mathfrak{s} -saturated resource if $\mathfrak{s} = \langle a = t \rangle$ or $\mathfrak{s} = \langle a \Leftarrow t \rangle$ and t is a Γ/\mathfrak{s} -saturated term .

We write \mathfrak{s} -saturated for short instead of Γ/\mathfrak{s} -saturated when Γ is understood or unimportant.

We take may testing as behavioral equivalence. A test is a process of type $\Theta^{\mathfrak{s}}$, the type of tests of sort \mathfrak{s} (i.e. \mathfrak{s} is either \top or \perp). There is a special inhabitant of this type: $\theta^{\mathfrak{s}}$, the success. Thus we have the following typing rule:

$$\frac{\Gamma \vdash}{\Gamma \vdash \theta^{\mathfrak{s}} : \Theta^{\mathfrak{s}}}$$

We use \mathfrak{T}^s to range over tests of sort s . Test \mathfrak{T} placed in parallel with a term t may interact and regarding t 's behavior, \mathfrak{T} may produce θ as a result of its observation. We say that a term t s -passes test \mathfrak{T} if there exists a reduction $\mathfrak{T} \parallel t \rightarrow^* t'$ such that t' is of the form $\theta^s \parallel u$. If t s -passes test \mathfrak{T}^s , we write $t \Downarrow_s^{\mathfrak{T}}$.

Definition 3 (May observational equivalence) *Let $\Gamma \vdash t, u : \tau$, with τ Γ/s -saturated, we have $t \cong_s u$ if for every test \mathfrak{T} such that $\Gamma \vdash \mathfrak{T} : \Theta^s$:*

$$t \Downarrow_s^{\mathfrak{T}} \iff u \Downarrow_s^{\mathfrak{T}}$$

We can now state the non-interference result for $\lambda_{\text{at}}\mathcal{E}$.

Theorem 3 (Non-interference) *If $\Gamma, x : \sigma \vdash t : \tau$, σ is \perp -saturated and τ is \top -saturated, and $\Gamma \vdash u, u' : \sigma$, then $t[x := u] \cong_{\top} t[x := u']$.*

The proof is done in a similar way as the proof of non-interference of [Pro00]. The idea is to introduce a dummy constant as a canonical term of some \perp -type. Then, an order relation between terms is defined relatively to those constants. A term t is less than another one u , $t \leq u$, if one can obtain t by substituting some u sub-terms with dummy constants. \leq is monotonic w.r.t. \rightarrow . Finally, we show that $\Gamma \vdash t, u : \tau$ and $t \leq u$ implies that $t \cong_{\top} u$, from which it is easy to infer the non-interference theorem.

Definition 4 (dummy terms) *Let $\Gamma \vdash \tau$ with τ Γ/s -saturated, define constant d_{τ} , such that $\Gamma \vdash d_{\tau} : \tau$.*

Definition 5 (pruning relation) *We inductively define the pruning relation \leq_{\perp} between well formed terms and addressed resources as follows:*

Terms:

- $d_{\tau} \leq_{\perp} t$ if $\Gamma \vdash t : \tau$, and τ is Γ/\perp -saturated.
- $x \leq_{\perp} x$.
- $(t \ u) \leq_{\perp} (t' \ u')$, and $(t \parallel u) \leq_{\perp} (t' \parallel u')$ if $t \leq_{\perp} t'$ and $u \leq_{\perp} u'$.
- $\lambda x : \tau. t \leq_{\perp} \lambda x : \tau. t'$, $\lambda \alpha. t \leq_{\perp} \lambda \alpha. t'$ and $\nu x : \tau(t) \leq_{\perp} \nu x : \tau(t')$ if $t \leq_{\perp} t'$.
- $(t \mid s) \leq_{\perp} (t' \mid s')$ if $t \leq_{\perp} t'$ and $s \mid s'$.

Addressed resources:

- $\langle a \Leftarrow t \rangle \leq_{\perp} \langle a \Leftarrow t' \rangle$ and $\langle a = t \rangle \leq_{\perp} \langle a = t' \rangle$ if $t \leq_{\perp} t'$.

- $(s_1 \mid s_2) \leq_{\perp} (s'_1 \mid s'_2)$ if $s_i \leq_{\perp} s'_i$, $i \in \{1, 2\}$.

Fact 2 (Dummy equality) *If $t \leq_{\perp} u$ and there are no dummy constant d_{τ} with $\tau \perp$ -saturated then $t = u$.*

Lemma 4 (maximality) *Let $\Gamma \vdash t, t' : \Theta^{\top}$ and $t \leq_{\perp} \theta^{\top} \leq_{\perp} t'$. Then $t = \theta^{\top} = t'$.*

Proof : By definition of \leq_{\perp} , we have that t is a constant of type Θ^{\top} : there are two such constants $d_{\Theta^{\top}}$, and θ^{\top} , now by definition of \leq_{\perp} , t must be either $d_{\Theta^{\top}}$ (notice it is the dummy type of sort \perp) or θ^{\top} . Hence the result. ■

Lemma 5 (Monotonicity) *Let $t \leq_{\perp} u$. Then,*

1. $u \rightarrow u' \implies \exists t' s.t. t \rightarrow^* t'$ and $t' \leq_{\perp} u'$.
2. $t \rightarrow t' \implies \exists u' s.t. u \rightarrow u'$ and $t' \leq_{\perp} u'$.

Proof : We have the following property (*) if $t_1 \leq_{\perp} t_2$ and $u_1 \leq_{\perp} u_2$ then $t_1\{x := u_1\} \leq_{\perp} t_2\{x := u_2\}$. It is proved by induction on the form of terms t_1, t_2 .

Another straightforward property (***) is that if $t \leq_{\perp} u$ then for any contexts $C[] \leq_{\perp} C'[[]]$, $C[t] \leq_{\perp} C'[u]$.

Proof of 1: Suppose that $C[]$ is a context such that $u = C[(\lambda x.t_x \ t_u)]$ and $u' = C[t_x\{x := t_u\}]$. We have $t \leq_{\perp} u$, thus two possibilities.

(1) $t = C[v]$ and $v \leq_{\perp} (\lambda x.t_x \ t_u)$. Again we have two subcases: either $v = d$, and in this case $t = t' = C[d] \leq_{\perp} C[t_x\{x := t_u\}] = u'$, or $v = (\lambda x.t'_x \ t'_u)$ with $t'_x \leq_{\perp} t_x$ and $t'_u \leq_{\perp} t_u$ from the definition of \leq_{\perp} . Now we can apply property (*) to conclude that $t'_x\{x := t'_u\} \leq_{\perp} t_x\{x := t_u\}$ thus by (***) we have

$$t \rightarrow t' = C[t'_x\{x := t'_u\}] \leq_{\perp} C[t_x\{x := t_u\}] = u'$$

(2) there exists a context $C'[]$ such that $t = C'[d]$ and $u = C'[C''[(\lambda x.t_x \ t_u)]]$, where $C''[]$ is another context. In this case let $t' = t = C'[d]$ then as $C'[d] \leq_{\perp} C'[w]$ for any term w , $C'[d] \leq_{\perp} C'[C''[t_x\{x := t_u\}]] = u'$.

For $u = C[t_x \mid \langle x \Leftarrow t_u \rangle]$ we reason in a similar way.

Proof of 2: Suppose that $C[]$ is a context such that $t = C[(\lambda x.t_x \ t_t)]$, and $t' = C[t_x\{x := t_t\}]$. Since $t \leq_{\perp} u$ it means that $u = C'[(\lambda x.t'_x \ t'_t)]$ with $C[] \leq_{\perp} C'[]$ and $(\lambda x.t_x \ t_t) \leq_{\perp} (\lambda x.t'_x \ t'_t)$. By definition of \leq_{\perp} it implies that $t_x \leq_{\perp} t'_x$ and

$t_u \leq_{\perp} t'_u$. We can apply property (*) to conclude that $t_x\{x := t_t\} \leq_{\perp} t'_x\{x := t'_t\}$, now by (**) we have $t' = C[t_x\{x := t_t\}] \leq_{\perp} \leq_{\perp} C'[t'_x\{x := t'_t\}] = u' \leftarrow u$. We conclude in a similar way if $t = C[t_x \mid \langle x \leftarrow t_t \rangle]$. ■

We can now complete the proof of theorem 3.

Proof :

Let \mathfrak{T} be any test. Suppose that $u \Downarrow_{\top}^{\mathfrak{T}}$. It means that $\mathfrak{T} \parallel u \rightarrow^* u'$ such that u' is of the form $\theta^{\top} \parallel u''$. Moreover we have from point 1 of lemma 5 that $\mathfrak{T} \parallel t \rightarrow^* t'$ and $t' \leq_{\perp} u'$. From the definition of \leq_{\perp} we have that t' is of the form $t' = t_1 \parallel t''$, and $t'' \leq_{\perp} u''$ and $t_1 \leq_{\perp} \theta^{\top}$ which implies by monotonicity lemma that $t_1 = \theta^{\top}$ which means that $t \Downarrow_{\top}^{\mathfrak{T}}$.

We reason in a similar way to prove $t \Downarrow_{\top}^{\mathfrak{T}} \implies u \Downarrow_{\top}^{\mathfrak{T}}$ using point 2 of lemma 5. ■

Among other things this theorem insures that \perp -terms and \top -terms don't interact together. Hence, one can imagine that a user annotates his private data with \top . Now if this user receives a program through the net, he can statically check that this program will not have access to its private data.

It has to be noted that this extension could also have been done using Boudol's original typing system for the blue calculus [Bou97]. Nevertheless, this type system as a typing rule of the form:

$$\frac{\Gamma \vdash t : \tau \quad \Gamma \vdash u : \tau}{\Gamma \vdash t \parallel u : \tau}$$

With such a typing rule, it is not possible to describe interferences between t and u using techniques based on sorts. On the other hand, if we use the type system presented in this paper, one can track dependencies with sorts. Let $\tau = \text{Int}^{\top} \rightarrow \text{Int}^{\perp} \rightarrow \text{Int}^{\top}$ and $\sigma = \text{Int}^{\top} \rightarrow \text{Int}^{\top} \rightarrow \text{Int}^{\top}$. Let $\Gamma \vdash t : \tau$ and $\Gamma \vdash u : \sigma$, then by rule *PPAR* we can conclude that $\Gamma \vdash t \parallel u : \text{Pa}(\tau, \sigma)$. Now from the shape of types, we can infer that there are no interference between u 's second argument (which is a natural of sort \perp) with t 's first and second arguments. In this simplistic example we can prove that if a private data is given as second argument to the function u , there is no way for t to “learn” anything about this private data. On the other hand there *may* be interferences regarding the first argument of u with the result of t because the type of t 's result is of the same sort than the sort of the type of the first argument of u .

5 CONCLUSION

In this paper we have introduced a new calculus which includes concurrency as well as functional features (λ -calculus): λ_{ar} . Those extra features are of interest: thanks to them it is possible to design a fine grained typing system for λ_{ar} . This typing system, unlike most of the ones for process algebras (e.g. [Pot02, BCPS03, Kob03]), is much in the line with traditional functional typing systems in the sense that types precisely reflect the structure of terms. As an application we have developed a non-interference calculus, $\lambda_{\text{ar}}\mathcal{E}$, built on top of the type system. This non-interference may be directly use for program slicing and useless code elimination, e.g. [Kob03]. $\lambda_{\text{ar}}\mathcal{E}$ types have the advantage that, closeley reflecting term structure, they can precisely reflect information flows.

Another difference of our approach relatively to other works about non-interference in the context of process algebras is that our typing system is not based on input/output capabilities (e.g. [HR00, YH00, PS96, BCC04]). Moreover, our analysis is *polyvariant*. A program can be analysed (that is annotated with sorts) only with sorts variables which in turn can be abstracted over the program. For each specific use of the analysed program one has just to instantiate sorts variables to check its security behavior.

REFERENCES

- [Aba97] M. Abadi. Secrecy by typing in security protocols. In *Proc. of 3rd Theoretical Aspects of Computer Software, LNCS 1281*, pages 611–638. Springer, 1997.
- [ABHR99] M. Abadi, N. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *Proceedings of the 27th ACM Symp. on Principles of Programming Languages (POPL'99)*, pages 147–160, January 1999.
- [Bar84] H. P. Barendregt. *The Lambda Calculus; Its Syntax and Semantics*. North-Holland, 1984. Revised Edition.
- [BB92] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217–248, 1992.
- [BCC04] M. Bugliesi, D. Colazzo, and S. Crafa. Type based discretionary access control. In *Proc. of the 15th International Conference, Concurrency Theory (CONCUR 2004), LNCS 3170*, pages 225–239, London, UK, August 2004. Springer.
- [BCPS03] M. Bugliesi, S. Crafa, A. Prelic, and V. Sassone. Secrecy in untrusted network. In *Proc. of 30th Int. Col., Automata, Languages and Programming (ICALP 2003), LNCS 2719*, pages 969–983, Eindhoven, The Netherlands, July 2003.
- [Bou97] G. Boudol. The pi-calculus in direct style. In *Proc. of the 24th ACM Symp. on Principles of Programming Languages (POPL'97)*, pages 228–241, 1997.
- [Gir87] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.
- [HR98] Nevin Heintze and Jon G. Riecke. The SLam calculus: programming with secrecy and integrity. In *Proc. of the 25th ACM Symp. on Principles of Programming Languages (POPL'98)*, pages 365–377, 1998.
- [HR00] M. Hennessy and J. Riely. Information flow vs. resource access in the asynchronous pi-calculus. In *Automata, Languages and Programming, 27th Int. Colloquium, (ICALP'2000), LNCS 1853*, pages 415–427. Springer, 2000.
- [HVV00] K. Honda, V. Vasconcelos, and N. Yoshida. Secure information flow as typed process behaviour. In Springer, editor, *Proc. of 9th European Symposium on Programming (ESOP'2000), LNCS 1782*, pages 180–199, 2000.
- [Kob03] N. Kobayashi. Useless-code elimination and program slicing for the pi-calculus. In *Proc. of the 1st Asian Symposium on Programming Languages and Systems (APLAS'03), Springer LNCS 2895*, pages 55–72, Beijing, China, November 2003.
- [Pot02] F. Pottier. A simple view of type-secure information flow in the pi-calculus. In *Proc. of the 15th IEEE Computer Security Foundations Workshop (CSFW'02)*, pages 320–330, Cape Breton, Nova Scotia, 2002.
- [Pro00] F. Prost. A static calculus of dependencies for the λ -cube. In *Proc. of IEEE 15th Ann. Symp. on Logic in Computer Science (LICS'2000)*. IEEE Computer Society Press, 2000.
- [PS96] B.C. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *MSCS*, 6(5):409–454, 1996.
- [SV98] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. of the 25th ACM Symposium on Principles of Programming Languages (POPL'98)*, pages 355–364. ACM, 1998.
- [YH00] N. Yoshida and M. Hennessy. Assigning types to processes. In IEEE Computer Society Press, editor, *Proc. of IEEE 15th Ann. Symp. on Logic in Computer Science (LICS'2000)*, pages 334–345, 2000.