In CONCUR 2001, Lecture Notes in Computer Scie nce, © Springer, 2001. To appear.

Typing mobility in the Seal Calculus

Giuseppe Castagna[1], Giorgio Ghelli[2], Francesco Zappa Nardelli[1,2]

[1] C.N.R.S., Département d'Informatique, École Normale Supérieure, Paris, France
[2] Dipartimento di Informatica, Università di Pisa, Pisa, Italy

Abstract. The issue of this work is how to type mobility, in the sense that we tackle the problem of typing not only mobile agents but also their movement. This yields higher-order types for agents. To that end we first provide a new definition of the Seal Calculus that gets rid of existing inessential features while preserving the distinctive characteristics of the Seal model. Then we discuss the use of interfaces to type agents and define the type system. This type system induces a new interpretation of the types: interfaces describe interaction *effects* rather than, as it is customary, provided *services*. We discuss at length the difference of the two interpretations and justify our choice of the former.

1 Introduction

In concurrent languages based on communication over channels it is customary to type both channels and messages in order to assure that only appropriate messages will transit over channels of a given type. When these languages are endowed with *agents* and *locations*, we also need typing information about the agents that are moved around the locations. Hence, we have to decide what is described by the type of an agent, and when this type is checked.

Our proposal is expressed in terms of a simplified version of the *Seal Calculus*. The Seal Calculus was defined in [19] as a set of primitives for a secure language of mobile agents to be used to develop commercial distributed applications at the University of Geneva; these primitives constitute the core of the JavaSeal language [18, 2]. It can be considered a safety-oriented calculus. From the Ambient Calculus [6], it borrows the idea that locations are places with a "boundary", which can only be crossed with some effort. In the Seal Calculus, boundaries can only be crossed when two parties, one inside the boundary and the other one outside, agree. Moreover, this movement operation takes place over a support layer, constituted by a set of channels. Communication takes place over channels too, as in the π-calculus [11], and the dangerous *open* operation of the Ambient Calculus is not present.

In our proposal the type of an agent is a description of the requests that it may accept from its enclosing environment. This is similar to an object type, in an object-oriented language; however, we will show that the subtype relation goes "the other way round". We will discuss this fact, which means that, while an object type describes a subset of the *services* that an object offers, our interface types describe a superset of the *effects* of an agent on its environment. We shall be more precise about it later on, but for the time being the reader can think of *services* as interactions that *must eventually* occur and of *effects* as the interactions that *may possibly* occur.

Our main results are the following ones. First, we define a variant of the Seal Calculus, which retains its essential security-oriented features but is simple enough to be suited to foundational studies. Then, in this context, we define a type system where we are able both to type mobile agents and to "type their mobility", i.e., to allow one to declare, for each location, the types of the agents that can enter or exit it. This yields to the first, as far as we know, higher order type system for agent mobility.

The article is structured as follows. In Section 2 we define our variant of the Seal Calculus. In Section 3 we introduce the typed calculus and justify our choices. In Section 4 we define the type system, a sound and complete type-checking algorithm, and we state some relevant properties they satisfy. In Section 5 we analyze our system and discuss the duality of *effects* vs. *services*. Section 6 describes an example that uses the key features of our calculus, while in Section 7 we hint at how our work can be used to provide the an Java agent kernel with a minimal type system. A summary and directions for future work conclude the article.

Related Work

Many works extend the basic type systems for π-calculus as described in [12, 15] giving more informative types to processes. We comment those closest to our work.

Yoshida and Hennessy propose in [20] a type system for a higher-order π-calculus that can be used to control the effects of migrating code on local environments. The type of a process takes the form of an interface limiting the resources to which it has access, and the type at which they may be used. In their type system both input and output channels can appear in the interface, appearing strictly more expressive than the system we propose here, where input channels are only considered. However, they do not allow active agents, but only pieces of code, to be sent over a channel. When code is received it can be activated, possibly after parameter instantiation. Besides, their type system limits the application of dependent types to the instantiation of parameters, resulting in the impossibility of giving an informative type to processes in which an output operation depends on an input one.

In [8] Hennessy and Riely define Dπ, a distributed variant of the π-calculus where agents are “located” (i.e., “named”) threads. The main difference with respect to our work is that locations cannot be nested (that is, locations are not threads), and therefore mobility in [8] consist in spawning passive code rather than migrating active agents. In [8] locations types have the form $\mathsf{loc}\{x_1\colon T_1, \ldots x_n\colon T_n\}$ where x_i's are channels belonging to the location (they are located resources and as such Dπ is much closer to the Seal Calculus as defined in [19], than to the version we introduce here: see Footnote 2). These types are syntactically very close to those we introduce here for Seal but they have opposed interpretations. Location types in [8] are intended both to describe the *services* provided by a location and to regulate access to them (they work as *views* in databases). Thus they embrace an object-oriented perspective (location types are subtyped as record types) without fully assuming it (services are declared but they are not ensured to be provided[1]). As we broadly discuss in the following—in particular in Section 5— our location types take the opposite perspective and rather describe the *effects* of a possible interaction with the location at issue.

Types for locations have been extensively studied in a series of papers [7, 5, 4] on the Ambient Calculus. Seal Calculus differs from Ambient Calculus in many aspects: seals cannot be opened (i.e. they boundaries cannot be dissolved), many channels exist, and moves are performed *objectively* (i.e., agents are passively moved by their environment) rather than *subjectively* (i.e., agents autonomously decide to move to a different location) —according to the terminology of [6]. From this work's viewpoint the prominent difference between Ambient and Seal Calculus is that the former does not provide an explicit “physical” support for mobility, while in the latter this support is provided by channels. In other words while in Ambients mobility take place on some unmaterialized *ætheral* transport medium, in Seal the medium is materialized by channels. Therefore the main novelty of this work is that not only we type locations (agents or seals), but we also type mobility (more precisely, its support). In some sense we introduce higher-order typing: while in Ambient Calculus an agent can not discriminate which agents can traverse its boundaries, this is possible in our type system. For the same reason we can make a mobile location become immobile, while this is not possible in the cited works on Ambient Calculus. Moreover, the mobility

[1] This is to solve the type dependency problem we describe in Section 4.1.

model of Ambient Calculus had to be extended with objective moves in [4], since the interaction of subjective moves with the *open* operation tends to produce typings where every ambient is typed as mobile. We show here that the mobility model of Seal Calculus is free from this problem.

2 Revising Untyped Seal Calculus

Seal Calculus is basically a π-calculus extended with *nested named locations* (dubbed *seals*) and mobility primitives. In Seal, interaction consists of synchronous communication of a value or of a whole seal. Both forms of interaction take place over named channels. Thus, mobility is obtained by communicating a seal on a channel. The existence of separate locations constraints the possible interactions: a channel can only allow interactions either among processes in the same seal, or among processes in two seals that are in parent-child relationship.

Two basic security principles underlay the design of the Seal Calculus: first, each seal must be able to control all interactions of its children, both with the outside world and one with the other; second, each seal must have total control over its name-space and therefore must determine the names of its children.

Besides these two basic features the Seal Calculus defined in [19] included some other features dictated by implementation issues. More precisely the calculus in [19] allowed seal duplication and destruction, and a strictly regulated access to remote channels[2].

In what follows we define a lighter version of Seal where seal creation and destruction is not possible and the access to remote channels is replaced by the introduction of shared channels[3].

The syntax of the language (parametric on an infinite set of *names*, ranged over by u, v, x, y, and z) is defined as follows:

Processes		**Actions**		**Locations**	
$P ::= \mathbf{0}$	inactivity	$\alpha ::= x^\eta(y)$	input	$\eta ::= *$	local
$\mid\ P \mid P$	composition	$\mid\ \overline{x}^\eta(y)$	output	$\mid\ \uparrow$	up
$\mid\ !P$	replication	$\mid\ \overline{x}^\eta\{\!\mid y \mid\!\}$	send	$\mid\ z$	down
$\mid\ (\boldsymbol{\nu}\, x)P$	restriction	$\mid\ x^\eta\{\!\mid y \mid\!\}$	receive		
$\mid\ \alpha.P$	action				
$\mid\ x[P]$	seal				

The first five process constructs have the same meaning as in the π-calculus, namely: the $\mathbf{0}$ process does nothing, the composition $P \mid Q$ denotes two processes P and Q running in parallel, the replication $!P$ unleashes an unbounded number of copies of P, the restriction $(\boldsymbol{\nu}\, x)P$ introduces a new name x and limits its scope to P (the scoping is lexical), and the prefix allows one to construct complex processes using the base actions α. A seal $x[P]$ is the representation in the syntax of a place named x that is delimited by boundaries and where the computation P takes place. The bare syntax of processes is the same as Ambient Calculus.

The basic computational steps in Seal are *communication* and *movement*. Communications (inputs/outputs on channels) are as in π-calculus with the only difference that channel names are super-scripted by location denotations. These are either $*$, or $\uparrow$, or z, and denote respectively the current seal (i.e. the seal where the action occurs), the parent seal, and a child-seal named z. Thus an action on x^* synchronizes only with local processes, $x^\uparrow$ means that x is a channel shared

[2] In [19] channels are considered as resources. Each channel belongs to one and only one seal. Some syntactic constructs allow the owner of a channel to regulate remote accesses to it and, thus, to control both remote communication and mobility.

[3] A similar solution was independently proposed for a calculus without agent mobility in [17].

between the current seal and the parent seal and that actions on it will synchronize with processes in the parent, and finally the shared channel x^z admits interactions between the current seal and a child-seal named z. These interactions are expressed by the first three rules in Figure 1.

$$
\begin{array}{lr}
x^*(u)\,.\,P \mid \overline{x}^*(v)\,.\,Q \rightarrow P\{^v/_u\} \mid Q & \text{(write local)} \\
x^y(u)\,.\,P \mid y[\overline{x}^\uparrow(v)\,.\,Q \mid R] \rightarrow P\{^v/_u\} \mid y[Q \mid R] & \text{(write out)} \\
\overline{x}^y(v)\,.\,P \mid y[x^\uparrow(u)\,.\,Q \mid R] \rightarrow P \mid y[Q\{^v/_u\} \mid R] & \text{(write in)} \\
\\
x^*\{\!|u|\!\}\,.\,P \mid \overline{x}^*\{\!|v|\!\}\,.\,Q \mid v[R] \rightarrow P \mid u[R] \mid Q & \text{(move local)} \\
x^y\{\!|u|\!\}\,.\,P \mid y[\overline{x}^\uparrow\{\!|v|\!\}\,.\,Q \mid v[R] \mid S] \rightarrow P \mid u[R] \mid y[Q \mid S] & \text{(move out)} \\
\overline{x}^y\{\!|v|\!\}\,.\,P \mid v[R] \mid y[x^\uparrow\{\!|u|\!\}\,.\,Q \mid S] \rightarrow P \mid y[Q \mid S \mid u[R]] & \text{(move in)}
\end{array}
$$

Fig. 1. Reduction rules.

Mobility is achieved in a similar way: seal bodies, rather than names, are moved over channels. It should be remarked that, contrary to input, receive is not a binding action: y is free in $x^\eta\{\!|y|\!\}$. A seal identified by its name is sent over a localized named channel: the seal together with its contents will disappear from the location of the sending processes and will reappear in the location of the receiving process. The receiving process can give a new name to the received seal: seal names are seen as local pointers to the location, and the actual name of a seal makes no sense outside the current location. Thus the action $\overline{x}^\eta\{\!|y|\!\}$ sends the body of the seal named y over the channel x^η, while $x^\eta\{\!|y|\!\}$ waits for a body on x^η and reactivates it as a seal named y. The precise semantics is given by the last three rules in Figure 1.

As customary, reduction uses structural congruence $\equiv$ that is the smallest congruence that is a commutative monoid with operation $\mid$ and unit $\mathbf{0}$, and is closed for the following rules:

$$
\begin{array}{llll}
!P \equiv\, !P \mid P & (\boldsymbol{\nu}\, x)\mathbf{0} \equiv \mathbf{0} & (\boldsymbol{\nu}\, x)(P \mid Q) \equiv P \mid (\boldsymbol{\nu}\, x)Q & \text{for } x \notin fn(P) \\
(\boldsymbol{\nu}\, x)(\boldsymbol{\nu}\, y)P \equiv (\boldsymbol{\nu}\, y)(\boldsymbol{\nu}\, x)P & & (\boldsymbol{\nu}\, x)y[P] \equiv y[(\boldsymbol{\nu}\, x)P] & \text{for } x \neq y
\end{array}
$$

The reduction semantics is completed by standard rules for context and congruence:

$$
\begin{array}{ll}
P \rightarrow Q \;\Rightarrow\; (P \mid R) \rightarrow (Q \mid R) & P \rightarrow Q \;\Rightarrow\; (\boldsymbol{\nu}\, x)P \rightarrow (\boldsymbol{\nu}\, x)Q \\
P \rightarrow Q \;\Rightarrow\; u[P] \rightarrow u[Q] & P \equiv P' \;\wedge\; P' \rightarrow Q' \;\wedge\; Q' \equiv Q \;\Rightarrow\; P \rightarrow Q
\end{array}
$$

3 Typing Mobility

In the introduction we anticipated that our solution for typing mobility was to type the transport media of mobility, that is, channels. We follow the standard π-calculus solution to type channels: a channel named x has type `Ch` V if V is the type of the *values* allowed to transit over x. We saw that in Seal channels are used both for communication (in which case they transport *messages*, i.e., base values, or names) and mobility (in which case they transport *agents*, i.e., seal bodies).

It is easy to type base values (in this work the only base values we consider are synchronization messages typed by `Shh`) and we just saw how to type channel names. So to define V we still have to define the type A of agents and, consequently, the type `Id` A of names denoting agents of type (more precisely, of interface) A.

3.1 Intuition about interfaces

Seals are named agents. The idea is to type them by describing all interactions a seal may have with the surrounding environment. We know that such interactions have to take place over the channels that cross the seal boundary. Thus these channels partially specify the interaction protocol of an agent. Keeping track of the set of upward communications (that is, communications with the parent) that a seal may establish can be statically achieved by keeping track of the channels that would be employed: this gives rise to a notion of interface of an agent as a set of *upward channels* (i.e., characterized by $\uparrow$ locations). Actually not all the upward channels are interesting for describing the interaction protocol of a seal. Those the seal is listening on suffice:

> *The* interface *of a seal is the set of upward channels that the process local to the seal may be listening on, with the type expected from interactions on them.*

We will discuss this choice later on (see Sections 4.1 and 5) but, for the moment, to see why such a definition is sensible we can consider as an example a networked machine. For the outer world the interface of such a machine —the description of how it is possible to interact with it— is given by the set of ports on which a dæmon is listening, together with the type of messages that will be accepted on them. So the interface of a machine can be described as a set of pairs (*port*:*type*). For example in our system a typical ftp and mail server would be characterized by a type of the following form $[21{:}\mathit{ftp};\ 23{:}\mathit{telnet};\ 79{:}\mathit{finger};\ 110{:}\mathit{pop3};\ 143{:}\mathit{imap};\ \ldots]$. Similarly, if you consider a seal as an object, then a process contained in it that listens on a upward channel $x^{\uparrow}$ can be assimilated to a method associated with a message x. In other words the sending of a message m with argument v to an object x (that is, `x.m(v)` in Java syntax) can be modeled in Seal by the action $\overline{m}^{x}(v)$, which would be allowed, in our type system, by a pair $m{:}M$ in the type of the seal x.

Hence, we consider interfaces such as $[x_1{:}\mathtt{Shh}; x_2{:}\mathtt{Ch}\ V; x_3{:}A; x_4{:}\mathtt{Id}\ A]$ that characterizes agents that may: 1) synchronize with an input operation on the upward channel x_1; 2) read over x_2 a channel name of type $\mathtt{Ch}\ V$ (the name of a channel that transports messages of type V); 3) receive over x_3 a seal whose interface is A; 4) read over x_4 a seal name of type $\mathtt{Id}\ A$. It is important to stress the difference between what can be transmitted over x_3 and x_4, respectively seals and seal names: the former requires mobility primitives, the latter communication primitives.

3.2 Syntax

The syntax of the types is reported in the following table.

Types

$V ::=$	M	messages
$\mid$	A	agents

Annotations

$Z ::=$	$\curvearrowright$	mobile
$\mid$	$\veebar$	immobile

Message Types

$M ::=$	$\mathtt{Shh}$	silent
$\mid$	$\mathtt{Ch}\ V$	channel names
$\mid$	$\mathtt{Id}^{Z}A$	agent names

Interfaces

$A ::=$	$[x_1{:}V_1; \cdots; x_n{:}V_n]$	agents

There are four syntactic categories in the type syntax, V, M, Z, and A respectively denoting types, message types, mobility annotations and agents. In the previous section we informally described three of them, omitting annotations. Let us see them all in more detail:

V: Types V classify *values*, that is computational entities that can be sent over a channel. While in π-calculus values are just channel names, in Seal we have both messages (classified by message types M) —which includes base values, channel names and agent names— and seals (more precisely seal's bodies, classified by interfaces A).

M: Message types M classify *messages*, that is entities that can be *communicated* (sent by an i/o operation) over channels. A message can be either a synchronization message (without any content) of type $\mathtt{Shh}$, or a name. In the syntax there is no distinction between channel names and seal names. This distinction is done at the type level: if $x : \mathtt{Ch}\ V$, then x is the name of a channel that transports values of type V; if $x : \mathtt{Id}^Z A$, then x is the name of a seal with interface A and with mobility attribute Z.

Z: On the lines of [4] we use mobility attributes to specify elemental mobility properties of seals: a $\curvearrowright$ attribute characterizes a mobile seal, while a $\veebar$ attribute characterizes an immobile one. Being able to discriminate between mobile and immobile agents is one of the simplest properties related to mobility. Contrary to what happens in [4], adding this feature does not require any syntax modification for Seal.

A: Interfaces A classify the *agents* of the calculus, keeping track of their interface. The notation $[x_1{:}V_1; \cdots ; x_n{:}V_n]$ is used to record the information about the interface of an agent. It is syntactic sugar for a set of pairs *channel_name* : *type*, that represent a functional relation. If a process has this interface, then it can be enclosed in an agent with the same interface, that is whose name has type $\mathtt{Id}^Z[x_1{:}V_1; \cdots ; x_n{:}V_n]$. This agent may listen from the upward level only on channels $x_1,\dots,x_n$.

The introduction of types requires a minimal modification to the syntax of the untyped calculus of Section 2. We have to add (message) type annotations to the two binders of the language: $(\boldsymbol{\nu}\, x{:}M)$ and $x^\eta(y{:}M)$, and to redefine free names *fn* as follows.

$$
\begin{array}{lll}
\mathit{fn}(x) = \{x\} \quad \mathit{fn}((\boldsymbol{\nu}\, x{:}M)P) = (\mathit{fn}(P) \setminus \{x\}) \cup \mathit{fn}(M) & & \mathit{fn}(\uparrow) = \mathit{fn}(*) = \mathit{fn}(\mathtt{Shh}) = \emptyset \\
\mathit{fn}(x^\eta(y{:}M).P) = (\mathit{fn}(P) \setminus \{y\}) \cup \mathit{fn}(M) \cup \mathit{fn}(\eta) \cup \{x\} & & \mathit{fn}(\mathtt{Id}\ A) = \mathit{fn}(A) \\
\mathit{fn}([x_1{:}V_1; \dots ; x_n{:}V_n]) = \{x_1, \dots , x_n\} \cup \mathit{fn}(V_1) \cup \cdots \cup \mathit{fn}(V_n) & & \mathit{fn}(\mathtt{Ch}\ V) = \mathit{fn}(V)
\end{array}
$$

The rule of structural congruence that swaps binders has to be changed too

$$(\boldsymbol{\nu}\, x{:}M)(\boldsymbol{\nu}\, y{:}M')P \equiv (\boldsymbol{\nu}\, y{:}M')(\boldsymbol{\nu}\, x{:}M)P \qquad \text{for } x \notin \mathit{fn}(M') \wedge y \notin \mathit{fn}(M) \wedge x \neq y$$

The reduction rules as well as the other rules and definitions are unchanged.

4 The Type System

In this section we define the type system we informally described in the previous section. However, before that, we need to add a last ingredient in order to deal with the technical problem of type dependencies.

4.1 Type dependencies

The notion of interface introduces names of the calculus at the type level. Since names are first order terms, type dependencies may arise. Consider for example the following terms.

$$P' = x^*(y{:}\mathtt{Ch}\ M).y^\uparrow(z{:}M) \qquad P = \overline{x}^*(w) \mid P'$$

P' offers upwards input on channel y. Hence, a naive syntax based analysis would associate P' and P with the interface $[y{:}M]$, producing the following typing judgment:

$$x{:}\mathtt{Ch}(\mathtt{Ch}\ M)\,,\ y{:}\mathtt{Ch}\ M\,,\ w{:}\mathtt{Ch}\ M \vdash P : [y{:}M].$$

However, the process P may perform an internal reduction on the channel x, and then it would offer upwards input on channel w, hence changing its interface type:

$$\underbrace{\overline{x}^*(w) \mid x^*(y{:}\mathtt{Ch}\ M).y^\uparrow(z{:}M)}_{[y:M]} \quad \rightarrow \quad \underbrace{w^\uparrow(z{:}M)}_{[w:M]}$$

This is the recurrent problem when trying to define non-trivial channel-based types for processes: to solve it one may consider using dependent types and deal explicitly with types that change during computation. Dependent types work fine for calculi where the notion of interaction is syntactically well-determined, as in λ-calculus. Unfortunately in process calculi, where interaction is a consequence of parallel composition (which admits arbitrary rearrangements of sub-terms), all the tries are somewhat unsatisfactory: they are usually restricted to a subset of the calculus, allowing dependent types only in particular, well-determined constructions [20].

Following a suggestion of Davide Sangiorgi, we decide to disallow input action on names bound by an input action. In this way interfaces cannot change during reduction: for example the process P above is not well-typed, since y is first bound by an input on x and then used to perform an input from $\uparrow$.

To make the type system uniform, we impose this condition on all the input operation, not only on the input operations from $\uparrow$, which are the only ones determining the interface.

There is no harm in doing that since this restriction does not limit the expressive power of the calculus: besides being theoretically well studied (see for example [10]), nearly all programming languages based on π-calculus impose this constraint, while programs written in concurrent languages that do not, mostly seem to obey to the same condition.

4.2 Typing rules

Judgments have the form $\Gamma \vdash_\Xi \Im$, where $\Im$ is either $\diamond$, or V, or $x\colon M$, or $P\colon A$. The pair Γ, Ξ will be referred to as *typing environment* and the judgments have the following standard meaning:

$\Gamma \vdash_\Xi \diamond$	well-formed environment	$\Gamma \vdash_\Xi V$	well-formed type
$\Gamma \vdash_\Xi x{:}M$	x has message type M	$\Gamma \vdash_\Xi P : A$	P has interface A

Γ is a function (actually, an ordered list) that assigns types to names. At the same time, we need some machinery to enforce the restriction on input channels we described above, that is, that only names *not* bound by an input action (i.e., names introduced by $\boldsymbol{\nu}$) are used to perform input operations. Thus we use the set of names Ξ to record the $\boldsymbol{\nu}$-introduced names:

$$\Gamma ::= \varnothing \;\mid\; \Gamma, x{:}M \qquad\qquad \Xi ::= \varnothing \;\mid\; \Xi, x$$

The typing and subtyping rules are:

(Env Empty)
$$\frac{}{\varnothing \vdash_\varnothing \diamond}$$

(Env Add)
$$\frac{\Gamma \vdash_\Xi M}{\Gamma, x{:}M \vdash_\Xi \diamond} \quad x \notin \mathrm{dom}(\Gamma, \Xi)$$

(Env Add Xi)
$$\frac{\Gamma \vdash_\Xi M}{\Gamma, x{:}M \vdash_{\Xi,x} \diamond} \quad x \notin \mathrm{dom}(\Gamma)$$

(Type Shh)
$$\frac{\Gamma \vdash_\Xi \diamond}{\Gamma \vdash_\Xi \mathtt{Shh}}$$

(Type Id)
$$\frac{\Gamma \vdash_\Xi A}{\Gamma \vdash_\Xi \mathtt{Id}^Z A}$$

(Type Ch)
$$\frac{\Gamma \vdash_\Xi V}{\Gamma \vdash_\Xi \mathtt{Ch}\, V}$$

(Type Interface)
$$\frac{\Gamma \vdash_\Xi \diamond \quad \forall i \in 1..n \;\; \Gamma \vdash_\Xi x_i{:}\mathtt{Ch}\, V_i \;\; x_i \in \mathrm{dom}(\Xi)}{\Gamma \vdash_\Xi [x_1{:}V_1, \ldots, x_n{:}V_n]}$$

(Var)
$$\frac{\Gamma \vdash_\Xi \diamond}{\Gamma \vdash_\Xi x : \Gamma(x)}$$

(Dead)
$$\frac{\Gamma \vdash_\Xi \diamond}{\Gamma \vdash_\Xi \mathbf{0} : [\,]}$$

(Par)
$$\frac{\Gamma \vdash_\Xi P_1 : A \qquad \Gamma \vdash_\Xi P_2 : A}{\Gamma \vdash_\Xi P_1 \mid P_2 : A}$$

(Bang)
$$\frac{\Gamma \vdash_\Xi P : A}{\Gamma \vdash_\Xi\, !P : A}$$

(Res)
$$\frac{\Gamma, x{:}M \vdash_{\Xi,x} P : A}{\Gamma \vdash_\Xi (\boldsymbol{\nu}\, x{:}M)P : A} \quad x \notin \mathit{fn}(A)$$

(Seal)
$$\frac{\Gamma \vdash_\Xi x : \mathtt{Id}^Z A \qquad \Gamma \vdash_\Xi P : A}{\Gamma \vdash_\Xi x[P] : [\,]}$$

(Output Local)
$$\frac{\Gamma \vdash_\Xi x : \mathtt{Ch}\, M \qquad \Gamma \vdash_\Xi y : M \qquad \Gamma \vdash_\Xi P : A}{\Gamma \vdash_\Xi \overline{x}^*(y).P : A}$$

(Input Local)
$$\frac{\Gamma \vdash_\Xi x : \mathtt{Ch}\, M \qquad \Gamma, y{:}M \vdash_\Xi P : A}{\Gamma \vdash_\Xi x^*(y{:}M).P : A} \quad x \in \mathrm{dom}(\Xi)$$

(Output Up)
$$\frac{\Gamma \vdash_\Xi x : \mathtt{Ch}\, M \qquad \Gamma \vdash_\Xi y : M \qquad \Gamma \vdash_\Xi P : A}{\Gamma \vdash_\Xi \overline{x}^{\uparrow}(y).P : A}$$

(Input Up)
$$\frac{\Gamma \vdash_\Xi x : \mathtt{Ch}\, M \qquad \Gamma, y{:}M \vdash_\Xi P : A}{\Gamma \vdash_\Xi x^{\uparrow}(y{:}M).P : (A \oplus [x{:}M])} \quad x \in \mathrm{dom}(\Xi)$$

(Output Down)
$$\frac{\Gamma \vdash_\Xi z : \mathtt{Id}^{Z} A' \qquad \Gamma \vdash y : M \qquad \Gamma \vdash_\Xi P : A}{\Gamma \vdash_\Xi \overline{x}^{z}(y).P : A} \quad (x{:}M) \in A'$$

(Input Down)
$$\frac{\Gamma \vdash_\Xi z : \mathtt{Id}^{Z} A' \qquad \Gamma \vdash_\Xi x : \mathtt{Ch}\, M \qquad \Gamma, y{:}M \vdash_\Xi P : A}{\Gamma \vdash_\Xi x^{z}(y{:}M).P : A} \quad x \in \mathrm{dom}(\Xi)$$

(Rcv Local)
$$\frac{\Gamma \vdash_\Xi x : \mathtt{Ch}\, A \qquad \Gamma \vdash_\Xi y : \mathtt{Id}^{Z} A \qquad \Gamma \vdash_\Xi P : A'}{\Gamma \vdash_\Xi x^{*}\{y\}.P : A'} \quad x \in \mathrm{dom}(\Xi)$$

(Snd Local)
$$\frac{\Gamma \vdash_\Xi x : \mathtt{Ch}\, A \qquad \Gamma \vdash_\Xi y : \mathtt{Id}^{\frown} A \qquad \Gamma \vdash_\Xi P : A'}{\Gamma \vdash_\Xi \overline{x}^{*}\{y\}.P : A'}$$

(Rcv Up)
$$\frac{\Gamma \vdash_\Xi x : \mathtt{Ch}\, A \qquad \Gamma \vdash_\Xi y : \mathtt{Id}^{Z} A \qquad \Gamma \vdash_\Xi P : A'}{\Gamma \vdash_\Xi x^{\uparrow}\{y\}.P : (A' \oplus [x{:}A])} \quad x \in \mathrm{dom}(\Xi)$$

(Snd Up)
$$\frac{\Gamma \vdash_\Xi x : \mathtt{Ch}\, A \qquad \Gamma \vdash_\Xi y : \mathtt{Id}^{\frown} A \qquad \Gamma \vdash_\Xi P : A'}{\Gamma \vdash_\Xi \overline{x}^{\uparrow}\{y\}.P : A'}$$

(Rcv Down)
$$\frac{\Gamma \vdash_\Xi z : \mathtt{Id}^{Z'} A' \qquad \Gamma \vdash_\Xi x : \mathtt{Ch} A \qquad \Gamma \vdash_\Xi y : \mathtt{Id}^{Z} A \qquad \Gamma \vdash_\Xi P : A''}{\Gamma \vdash_\Xi x^{z}\{y\}.P : A''} \quad x \in \mathrm{dom}(\Xi)$$

(Snd Down)
$$\frac{\Gamma \vdash_\Xi z : \mathtt{Id}^{Z} A_1 \qquad \Gamma \vdash_\Xi y : \mathtt{Id}^{\frown} A \qquad \Gamma \vdash_\Xi P : A_2}{\Gamma \vdash_\Xi \overline{x}^{z}\{y\}.P : A_1} \quad (x{:}A) \in A_1$$

(Subsumption)
$$\frac{\Gamma \vdash_\Xi P : A \qquad \Gamma \vdash_\Xi A' \qquad A \leq A'}{\Gamma \vdash_\Xi P : A'}$$

(Sub Interface)
$$\frac{A \subseteq A'}{A \leq A'}$$

We discuss the most important rules:

(Env Add _): It is possible to add names with their types to Γ, and also to Ξ (rule (Res)), provided that they are not already in Γ. Notice that $\mathrm{dom}(\Xi) \subseteq \mathrm{dom}(\Gamma)$ holds for well-formed environments.

(Type Interface): An interface type is well-formed if every name in it has been previously declared with the correct type and appears in Ξ (i.e., it is not bound by an input action). The premise $\Gamma \vdash_{\Xi} \diamond$ ensures the well formation of the type environment for the case of the empty interface.

(Res): Particular attention must be paid to restrictions of channel names, as channels may occur in the interface. A first idea could be to erase the newly restricted name from the interface as in the rule aside, but this rule is not sound with respect to the structural congruence relation: if you consider the processes $(\boldsymbol{\nu}\, y{:}\mathtt{Id}^Z[\,])y[(\boldsymbol{\nu}\, x{:}\mathtt{Ch\ Shh})x^{\uparrow}()]$ and $(\boldsymbol{\nu}\, y{:}\mathtt{Id}^Z[\,])(\boldsymbol{\nu}\, x{:}\mathtt{Ch\ Shh})y[x^{\uparrow}()]$ they are structurally equivalent, but while the former would be well-typed, the latter would not.

$$\text{(\textit{Wrong} Res Ch)}\quad \frac{\Gamma, x{:}\mathtt{Ch}\ V \vdash_{\Xi,x} P : A}{\Gamma \vdash_{\Xi} (\boldsymbol{\nu}\, x{:}\mathtt{Ch}\ V)P : (A - [x{:}V])}$$

Therefore we rather use the *(Res)* rule that imposes that a restricted name can not appear in the interface of the process (see also comments right below Property 1 Section 4.4). As all the names must be declared in Γ, it may seem that this condition forces all the interfaces to be empty. But note that this restriction applies only to process interfaces not to seal identifiers. The reader must avoid confusion between the name a which has type $\mathtt{Id}^Z A$ (where A may be a very complex interface) and the process $a[P]$ which, as stated by the rule (*Seal*), has type $[\,]$. What is necessary is that the type of P (rather than the one of $a[P]$) has interface A. That is, that the process P inside a seal $a[P]$ respects the interface declared for its name a. Therefore the side condition of (*Res*) simply demands that the upward channels of a are not restricted inside $a[P]$. In other words, a channel appearing in an interface must be already known to the enclosing environment. This is a very desirable feature of the type system: the interface's names must be somewhat public.

A brief example can clarify what "somewhat" means. Consider the following two terms in the light of the *(Res)* rule, and notice that they are structurally equivalent:

$$1)\quad y[(\boldsymbol{\nu}\, x{:}\mathtt{Ch\ Shh})\ x^{\uparrow}()] \qquad\qquad 2)\quad (\boldsymbol{\nu}\, x{:}\mathtt{Ch\ Shh})y[x^{\uparrow}()]$$

Clearly, the first is not well-typed, since the process inside the seal should offer a restricted channel in the interface, and this is forbidden by the *(Res)* rule. Interestingly, the latter is not well-typed either: the type of the name y should include the channel x in the interface, but y is defined out of the scope of x; therefore process in the scope of the restriction could be typed only under a context in which x is declared twice, which is impossible (see Property 1(c) in Section 4.4). The correct term is $(\boldsymbol{\nu}\, x{:}\mathtt{Ch\ Shh})(\boldsymbol{\nu}\, y{:}\mathtt{Id}\ [x{:}\mathtt{Shh}])\ y[x^{\uparrow}()]$ in which the channel x is declared *before* the seal y. Briefly, a name that is used by a seal to read from its environment must already exist in the environment where the seal is declared.

In terms of the examples in Section 3.1, this means that we can declare that a machine x has interface [23 : *telnet*] only if the channel named 23 and the type *telnet* are both already known (that is, declared) in the environment.

(Input _): All the rules for typing a process of the form $\alpha.P$ follow a common pattern: this is especially true if we consider input and output rules separately.

The action $x^{\eta}(y{:}M).P$ binds y in P. Thus y must be added to the environment of P, provided that its type matches the type of x; y is not added to Ξ since it is bound by an input operation. Because we are doing an input, we also have to check that x is a $\boldsymbol{\nu}$-introduced name, that is $x \in \mathrm{dom}(\Xi)$. In *(Input Local)*, the input operation is local and nothing more has to be done. In *(Input Down)* we also check that the name of the seal from which the process wants to read is declared in Γ. In *(Input Up)* the input is from $\uparrow$, therefore the channel the process wants to read from must be added to the interface already deduced for P. This is done by the $\oplus$ operator, which computes the union of two interfaces, but is not defined when the result would contain two different pairs $y{:}M$ and $y{:}M'$ with the same name y but different M, M'.

(Output _): In the case of local and upward output actions the rules *(Output Local)* and *(Output Up)* check that the types of the channel and of the argument match. The rule *(Output Down)* furthermore checks that the channel appears in the interface of the target seal with the right type. This enforces the interpretation of the interfaces: a process can write inside a seal only if the processes local to the seal are possibly going to read it.

(Rcv _): The typing rules for mobility actions do not differ from the respective communication actions. The main point is that in a receive operation the object name is not bound, so it is not added to the names in the scope of the continuation[4]. Remark that in order to send a seal on a channel, it must be declared to be mobile (attribute $\curvearrowright$). In the Seal's model of mobility, when a seal is received it gets a name chosen by the receiver process. We use this feature, together with the fact that the mobility attribute is tied to seals names, to turn a mobile seal into an immobile one. For instance, $(\boldsymbol{\nu}\, x{:}\mathtt{Ch}\, A)(\boldsymbol{\nu}\, a{:}\mathtt{Id}^{\frown}\!A)(\boldsymbol{\nu}\, b{:}\mathtt{Id}^{\vee}\!A)\; \overline{x}^{*}\{a\} \mid x^{*}\{b\} \mid a[P] \;\;\rightarrow\;\; (\boldsymbol{\nu}\, b{:}\mathtt{Id}^{\vee}\!A)\; b[P]$ turns the mobile seal named a into an immobile seal named b (the opposite is also possible). This is achieved by imposing no constraints on the mobility attribute of the receiving name in the receive typing rule. Neither this nor the opposite is possible in [4].

(Subtyping) During reductions, actions can be consumed. Consider for example the process $P = x^{\uparrow}(y{:}M).\overline{z}^{*}(y)$. It is ready to input a name of type M on channel x and its type is $[x{:}M]$. Now place it in the context $\mathscr{C}[-] = \overline{x}^{a}(w) \mid a[-]$ and consider the type of P and of its reductum:

$$\overline{x}^{a}(w).Q \mid a[\underbrace{x^{\uparrow}(y{:}M).\overline{z}^{*}(y)}_{[x:M]}] \;\;\rightarrow\;\; Q \mid a[\underbrace{\overline{z}^{*}(w)}_{[\,]}]$$

To satisfy the subject reduction property we introduce a subtyping relation. We already discussed that the interface of a process should be regarded as the set of channels on which the process *may* perform input operations from $\uparrow$. This suggests that the addition of new channels in the interface of a process should not be considered as an error, since they are channels on which interaction will never take place. This is formalized by the subtyping notion defined in the *(Sub Interface)* rule, that allows channels to be added to the interface of a process.

This possibility of extending the interface is limited to the process types, and is not extended to seal interfaces. The interface of a seal is associated with its name and is immutable, hence it characterizes forever the range of interactions admitted by that seal. At the same time, subsumption allows a process with a smaller interface to be placed inside the seal. This is essential, since the more limited interface may be a consequence, as in the previous example, of the "consumption" of some actions. In this way, actions can get consumed inside a seal, while the seal preserves its crystallized interface.

4.3 Typing algorithm

The type rules in the previous section just need some slight modification to be converted into a type algorithm. As usual in type systems with subtyping, we must eliminate the subsumption rule by embedding subtyping in the other rules. Actually there are only two rules that need modifications. The first is the *(Par)* rule: in order to type-check $P_1 \mid P_2$ in the environment Γ, Ξ both P_1 and P_2 are checked resulting respectively in the two types A_1 and A_2. If the process $P_1 \mid P_2$ can perform an input at $\uparrow$ then either P_1 or P_2 must be able to perform it, and so it has been registered in one of A_1 and A_2. Thus we have to merge the type informations kept in A_1 and A_2, and this is achieved by means of the $\oplus$ operator.

The second rule we need to modify is the *(Seal)* rule, to take into account that the interface of the process inside a seal may be a subtype of the interface associated with the seal name.

(Par Algo)
$$\frac{\Gamma \rhd_{\Xi} P_1 : A_1 \qquad \Gamma \rhd_{\Xi} P_2 : A_2}{\Gamma \rhd_{\Xi} P_1 \mid P_2 : A_1 \oplus A_2}$$

(Seal Algo)
$$\frac{\Gamma \rhd_{\Xi} x : \mathtt{Id}\, A \qquad \Gamma \rhd_{\Xi} P : A'}{\Gamma \rhd_{\Xi} x[P] : [\,]} \quad A' \leq A$$

[4] This is due to the specificity of the receive action: when a seal is received it is activated at the same level as the process that received it. The movement actions look like interactions in the Fusion Calculus [14].

4.4 Properties

The typing algorithm defined above is sound and complete with respect to the type system.

Theorem 1 (Soundness and completeness).
1. *If* $\Gamma \rhd_{\Xi} P : A$ *then* $\Gamma \vdash_{\Xi} P : A$.
2. *If* $\Gamma \vdash_{\Xi} P : A$ *then* $\exists A'$ *such that* $A' \leq A$ *and* $\Gamma \rhd_{\Xi} P : A'$.

A corollary of this theorem is the minimality of the algorithmic type:

Corollary 1. $\Gamma \rhd_{\Xi} P : \min\{A \mid \Gamma \vdash_{\Xi} P : A\}$, *if the set is not empty.*

In order to prove the subject reduction property we need a substitution lemma that states that substituting names for names of the same type in well-typed terms yields well-typed terms. This would fail if we allowed names that appear in interfaces to be substituted, hence we have to add a condition $x \notin \mathrm{dom}(\Xi)$ in the theorem hypothesis. This restriction is not a problem, since, as formalized by the management of Ξ in the *(Input _)* rules, interactions can only substitute names that do not appear in $\mathrm{dom}(\Xi)$.

Thanks to Theorem 1, the substitution lemma can be stated directly on the type algorithm rather than on the type system.

Lemma 1. *If* $\Gamma, x{:}M \rhd_{\Xi} P : A$, $x \notin dom(\Xi)$, *and* $\Gamma \rhd_{\Xi} y : M$, *then* $\Gamma \rhd_{\Xi} P\{^{y}/_{x}\} : A$.

This lemma is used to prove the subject reduction property for the algorithmic system whence subject reduction for the type system can be straightforwardly derived:

Theorem 2 (Subject Reduction). *If* $\Gamma \rhd_{\Xi} P : A$ *and* $P \rightarrow Q$ *then* $\Gamma \rhd_{\Xi} Q : A$.

Besides the characteristics discussed in Section 4.2 there several subtleties hidden in the type system that make subject reduction hold while keeping the rules relatively simple. Among these it is noteworthy to remark that the provability of a judgment $\Gamma \vdash_{\Xi} \Im$ implies the following properties[5]:

Property 1. If $\Gamma \vdash_{\Xi} \Im$ si provable then:
a. Γ, Ξ are well formed (i.e., $\Gamma \vdash_{\Xi} \diamond$ is provable);
b. $\mathrm{dom}(\Xi) \subseteq \mathrm{dom}(\Gamma)$;
c. each variable has at most one type assignment in Γ.

These three properties allowed us to omit several sensible conditions from the typing rules since they are implicitly satisfied. So for example in the (Res) rule it is useless to require that $x \notin \mathrm{dom}(\Xi)$ since this already holds by the well-formation of the environment. Indeed $\Gamma, x{:}M \vdash_{\Xi,x} \diamond$ implies that $x \notin \mathrm{dom}(\Gamma, \Xi)$. Even more, $\Gamma, x{:}M \vdash_{\Xi,x} \diamond$ implies that x does not occur in Γ, since by construction Γ is an ordered list; this rules out envirements such as $y{:}\mathtt{Id}[x{:}M'], x{:}M$. Similarly, in all the rules (Input _) it always holds that y does not occur in Γ. This implies that $y \notin \mathrm{dom}(\Xi)$ since otherwise $\Gamma \not\vdash_{\Xi} \diamond$ which contradicts that $\Gamma \vdash_{\Xi} x : \mathtt{Ch}\ M$ is provable.

5 Services vs. Effects

In the introduction we hinted at two possible interpretations of agent interfaces. Interfaces may describe either *services*, that is the interactions that *must eventually* occur, or *effects*, that is the interactions that *may possibly* occur.

[5] The first property follows by straightforward induction whose base are the rules (Type Shh), (Var), and (Dead). The other two are equally straightforward.

The former interpretation is the one that characterizes the type systems for object-oriented languages, while the latter is the one of our system. Indeed, superficially our interfaces look like the types of the objects in the "objects as records" analogy: just an array of methods one can invoke (in fact, the analogy between agents and objects is not a piece of news). However, there is an important difference. In the object framework, sending a message should result in a method being activated: the type of an object reports the set of messages the object will answer to. We can say that the interface of an object characterizes the *services* that the object offers to its environment.

According to our definition, a channel that appears in the interface of an agent (a seal) does not guarantee that interaction on this channel is always going to happen (indeed the channel may be guarded or already be consumed by a previous action). A more precise intuition of our system is that an interface limits the *effects* that the agent can have on the environment: if an interaction occurs, it occurs on a channel defined in the interface and not on other channels.

There is a clear tension between the two interpretations and in this paper we opted for the second one. The reason for such a choice resides in the fact that π-calculus channels are essentially consumable resources. One of the clearest lessons we draw from this work is that there is an inherent difference between requiring a service (such as sending a message) and writing on a channel: the former does not affect the set of admissible interactions, while the latter does (by consuming a channel).

This tension is manifest at the level of subtyping: in case of effects the "may-provide" interpretation is embodied by a subtyping relation typical of *variant types* while in the case of services, we recover the classical record types relation that characterizes objects and their "must-provide" interpretation, as expressed by the rules on the side.

(effects) $\dfrac{A \subseteq A'}{A \leq A'}$ (services) $\dfrac{A' \subseteq A}{A \leq A'}$

Our analysis clearly shows that the two approaches are mutually exclusive, and that either one or the other has to be adopted according to the "consumability" of the communication layer.

In our system it is possible to recover the object/services characteristics by imposing restrictions to ensure *receptiveness* [16] of channels in the interface[6], which roughly corresponds to make all the external interactions of an agent unconsumable. The intuition is that in this way we transform interface channels into (object) methods. Receptiveness can be ensured by imposing restrictions such as those presented in [1] or, in a coarser approach, by requiring that all receive and input actions on upper channels are guarded by replications, that is they must all be of the form $!x^{\uparrow}(y).P$ and $!x^{\uparrow}\{\!|y|\!\}.P$. In the latter case some simple modifications to our type system allow us to recover the service interpretation together with its (services) subtyping rule. It just suffices to straightforwardly modify the typing rules (Input Up) and (Rcv Up) to account for the new syntax, and the results of the previous section bring forth. However we decided to follow the other approach since the presence of concurrency does not ensure that services will be eventually satisfied. Indeed, even if the remote interactions are replicated they may still be guarded. Therefore a complete treatment would require further restrictions on these interactions bringing us too far from our original subject. Nevertheless we believe that such a direction is worth exploring.

6 Example: a web crawler

In this section we give a simple example that uses mobility, higher-order types, and parametric write channels. Chapter 5 of [21] contains a much more complex example we did not include here for lack of space: in that example the toy distributed language introduced in [4] to show the expressivity of typed ambients is encoded in the Seal calculus version presented here.

[6] An alternative solution is to use the object framework but to give up with the "must provide" interpretation, as it is done in [8].

In order to show a possible application of higher order typing and mobility attributes, we suggest the specification of a possible *web crawling* protocol. Currently, most commercial web search engines periodically access all the web pages that are reachable by some starting pages and index them in a database. Web searches access the database and retrieve relevant entries.

This technique is a greed bandwidth consumer. It may be interesting to define an alternative protocol where mobile agents are spawned over the web sites, where they collect and pre-elaborate the relevant information, so that the computational effort is distributed, and bandwidth consumption is dramatically reduced.

The Seal specification of this protocol is depicted in Figure 2, where top level represents the network and hosts are immobile seals that lie inside it; crawlers are modeled by mobile seals, being able to navigate among hosts.

$$\textsc{System} = \textsc{Home} \mid \textsc{NetSupport} \mid \textbf{WebSite\_1}[\textsc{WebSite}] \mid \dots \mid \textbf{WebSite\_n}[\textsc{WebSite}]$$

$$\begin{array}{l}
\textsc{Crawler}(\textbf{start}) = \overline{\mathrm{cd}}^{\uparrow}(\textbf{start}). \\
\quad \texttt{repeat}\ (\ \mathrm{in}^{\uparrow}(\mathrm{info}{:}\mathit{info})\ .\ <\textsc{ProcessInfo}>\ . \\
\qquad \texttt{if}\ \ \textbf{nextDest}\ \ \texttt{then}\ \ \overline{\mathrm{cd}}^{\uparrow}(\textbf{nextDest})\ \texttt{else}\ \ \mathrm{result}^{\uparrow}(\mathrm{k}{:}\texttt{Ch}\ \mathit{info}).\overline{\mathrm{k}}^{\uparrow}(\mathrm{crawledInfo})\ )
\end{array}$$

$$\begin{array}{l}
\textsc{Home} = \textbf{craw}[\textsc{Crawler}(\textbf{WebSite_1})] \mid \dots \mid \textbf{craw}[\textsc{Crawler}(\textbf{WebSite_n})] \mid \\
\quad \texttt{repeat}\ (\ (\boldsymbol{\nu}\ \mathrm{k}{:}\texttt{Ch}\ \mathit{info})\ \overline{\mathrm{result}}^{\textbf{craw}}(\mathrm{k}).\mathrm{k}^{\textbf{craw}}(\mathrm{crawledInfo}{:}\mathit{info}).<\textsc{StoreCrawledInfo}>\ )
\end{array}$$

$$\textsc{WebSite} = 437^{\uparrow}\{\textbf{craw}\}.\overline{\mathrm{in}}^{\textbf{craw}}(\mathrm{info}).\overline{437}^{\uparrow}\{\textbf{craw}\} \mid <\textsc{OtherServices}>$$

$$\begin{array}{l}
\textsc{NetSupport} = \texttt{repeat}\ (\ \ (\boldsymbol{\nu}\ x{:}\texttt{Ch}\ \mathit{craw}) \\
\quad (\ \overline{x}^{*}\{\textbf{craw}\} \mid (\boldsymbol{\nu}\ \mathbf{c}{:}\texttt{Id}^\frown\mathit{craw})\ x^{*}\{\mathbf{c}\}.\mathrm{cd}^{\mathbf{c}}(\mathrm{dest}{:}\mathit{hostName}).\overline{437}^{\mathrm{dest}}\{\mathbf{c}\}.437^{\mathrm{dest}}\{\textbf{craw}\})\ )
\end{array}$$

Fig. 2. A web crawler protocol

HOME, which is a process that lives at the network level, spawns a crawler for each root web site. The crawler will go away and come back, to tell HOME about its findings, as we will see later.

CRAWLER communicates on channel "cd" ("crawler destination") the name of the first site it wants to visit[7]. This information is received by NETSUPPORT which first renames the crawler with a fresh name c; this renaming is performed by sending the crawler along the local channel x. Then, NETSUPPORT sends the crawler to the requested destination, via the port 437. Once the crawler is in the site, it reads the information via the port "in", and is sent out of WEBSITE along channel 437. The crawler processes the information (which generates a list of other possible destinations), then checks whether it has to visit more sites; if it does not, it uses the channel "result" to ask HOME for a secure channel k and sends the result on it.

HOME sends the secure channel name k along the $\mathrm{result}^{\textbf{craw}}$ channel, reads the collected information from k, and stores it.

A generic WEBSITE must have a dæmon that is ready to receive crawlers on port 437 and, after having provided them with information on channel "in", sends them out via the port 473 again.

The interface that characterizes a crawler is $\mathit{craw} = [\mathrm{in}\colon \mathit{info}\ ;\ \mathrm{result}\colon \texttt{Ch}\ (\mathit{info})]$. All the crawlers in the toplevel have the name **craw**, of type $\texttt{Id}^\frown[\mathrm{in}\colon \mathit{info}\ ;\ \mathrm{result}\colon \texttt{Ch}\ (\mathit{info})]$. The other relevant interface in the example is the one of the hosts: it is a higher-order type, since it contains the interface *craw*, i.e. it specifies the protocol of the agents it is willing to accept. This interface

[7] `repeat` is syntactic sugar for ! and `if_then_else` can be easily encoded. We use italics for types, roman font for channels, small capitals for metavariables, and boldface font for seal names.

has the following form: $[437 : \textit{craw}\ ; < \text{OTHERPORTS} >]$. Since hosts are immobile, they are denoted by names whose type is $\textit{hostName} = \mathtt{Id}^{\curlyvee}[437 : \textit{craw}\ ; < \text{OTHERPORTS} >]$.

7 Practical Applications

In order to show the potential of our type system we hint at how it can be used to provide the JavaSeal Agent Kernel [2, 18] with a minimal type system. JavaSeal is an experimental platform that provides several abstractions for constructing mobile agent systems in Java. JavaSeal uses relatively coarse grained types; in particular, objects exchanged during interaction are encapsulated in `Capsules`. Capsules are the only entities liable to be sent over channels. The contents of a capsule are widening to the generic type `Object` thus loosing all static information. Furthermore, the system does not distinguish between channel and seal identifiers as both are denoted by objects of the class `Name`. In other words, JavaSeal does little type checking and what it does is mostly performed at run time through dynamic type casts. This means that JavaSeal agents are prone to errors.

In particular, each object exchanged during interaction is encapsulated with type `Object` into a `Capsule`, being capsules the only entities liable to be sent over channels. Also there is not a clear distinction between channels and seal identifiers since they are generically classified by the class `Name`. In other words JavaSeal type checking is rather weak since it heavily relies on the use of dynamic type casts, and as such it is quite prone to errors.

JavaSeal is based on the primitives of the original Seal calculus. Therefore it does not provide shared channels: channels are localized and access to them is granted via portals opening operations. More precisely this signifies that for example a downward output operation on channel x^y synchronizes only with local input operation on x in the seal y, and that the interaction needs presence in y of an explicit permission $\mathsf{open}_{\uparrow}x$ that authorizes the parent to use the local channel x. That is the (write in) becomes the following three parties reduction rule:

$$\overline{x}^y(v)\,.\,P \mid y[x^*(u)\,.\,Q \mid \mathsf{open}_{\uparrow}x \mid R] \quad \rightarrow \quad P \mid y[Q\{{}^v\!/\!_u\} \mid R] \qquad \text{(write in)}$$

It is quite straightforward to adapt our interfaces types to located channels and portals: recall that interfaces trace all the channels on which there is an information flow from the parent to the child. Therefore the interface of a (Java)Seal agent must contain all channels the agent may perform input on and that (a) either are located in the parent (b) or are local and have a matching upward portal open operations.

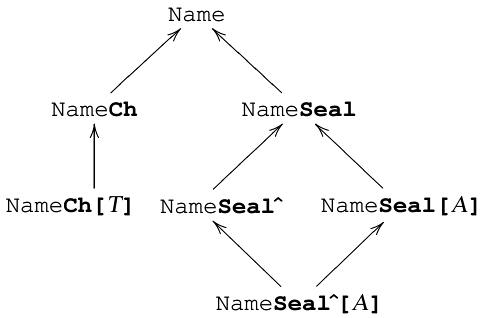
Our proposal is then to endow the actual JavaSeal syntax with some type informations that will be processed by a preprocess to type-check the source, and then will be erased yielding a standard JavaSeal program. In order to enhance readability we write the information that are to be erased by the preprocessor in boldface. More particularly we propose to add the following (preprocessor) types:

`Name`**`Ch[`**T**`]`** it is used to classify channel *names* (it corresponds to $\mathtt{Ch}\,T$). The type part **`[`**T**`]`** is optional (its absence means that the content of the channel does not need to be checked)

`Name`**`Seal^[`**A**`]`** it is used to classify seal names (it corresponds to $\mathtt{Id}^{\curlyvee}A$). Both the immobility attribute **`^`** and the interface part **`[`**A**`]`** are optional (the absence of **`^`** corresponding to $^{\frown}$, and the one of the interface meaning that outputs towards the seal do not need to be checked).

In order to have backward compatibility and let the programmer decide how fine-grained the preprocessor analysis should be, we order the newly introduced types according to the subtyping relation described by the diagram below.

Thus the `Name` type that in the current JavaSeal implementation identifies all names, will be refined by separating channel names from agent names. Agent names will allow a second refinement by specifying their interfaces or its mobility attribute. A similar specialization is possible by specifying or not the content of the channel.



The idea is that the programmer is free to decide whether the preprocessor has just to check that, say, the name used to denote a channel is indeed a channel name, or also match the type of its content. Similarly the programmer may force the check of downward write operations, or just require that they are directed to some named seal. The more the leaves of the hierarchy are used the more the check will be complete.

Thus the `Name` type that in the current JavaSeal implementation identifies all names, will be refined by separating channel names from agent names. Agent names will allow a second refinement by specifying their interfaces or its mobility attribute. A similar specialization is possible by specifying or not the content of the channel.

The idea is that the programmer is free to decide whether the preprocessor has just to check that, say, the name used to denote a channel is indeed a channel name, or also match the type of its content. Similarly the programmer may force the check of downward write operations, or just require that they are directed to some named seal. The more the leaves of the hierarchy are used the more the check will be complete.

This system is particularly interesting when it is used in conjunction with parametric classes such as they are defined for example in Pizza [13]. So the `Capsule` and `Channel` classes of JavaSeal could be rewritten as follows

```
final class Capsule<X> implements Serializable {
  Capsule(X obj);
  final X open();
}

final class Channel<X> {
  static void send(NameCh[X] chan, NameSeal seal, Capsule<X> caps);
  static Capsule<X> receive(NameCh[X] chan, NameSeal seal);
}
```

It is interesting to notice that after preprocessing, by applying the Pizza homogeneous translation of [13] to all non-erased occurrences of the type variable, one recovers the original interface of JavaSeal:

```
final class Capsule implements Serializable {
  Capsule(Object obj);
  final Object open();
}

final class Channel {
  static void send(Name chan, Name seal, Capsule caps);
  static Capsule receive(Name chan, Name seal);
}
```

8 Conclusion

In this work we presented a new definition of the Seal Calculus that gets rid of existing inessential aspects while preserving the distinctive features of the Seal model. We used it to tackle the problem of typing not only mobile agents but also their movement, so that the latter can be controlled and regulated. The solution we used, typed channels, is an old one —it is standard in π-calculus— but its use for typing mobility is new, and results into a higher order type systems for agents (as [20] is a higher order type system for processes). At the same time, we designed our type system so that it induces an interpretation of interfaces as effects that differs from the customary interpretation as services and we discussed its distinctive features.

This work is just a starting point and for some aspects it is still unsatisfactory. For example more work is needed to define a type system that captures one of the peculiar security characteristics of the Seal Calculus, that is the name-spaces separation: in the actual version if two agents residing in different locations have the same name, then the type system forces them to have the same type too.

At the same time this work already constitutes an exciting platform whence further investigation can be started. In particular we are planning to use some form of grouping similar to those in [5, 3] for a different solution to the problem of type dependencies, as well as to investigate a distributed version of the type system, on the lines of [3]. It would also be interesting to verify what the *single threaded types*, introduced in [9] for the Ambient Calculus with co-actions, would bring forth in the Seal Calculus, where co-actions are inherently present.

Acknowledgments : The authors want to thank Mariangiola Dezani, Jan Vitek, and the anonymous referees for useful comments on this paper. This work was partially supported by CNRS Program *Telecommunications*: "Collaborative, distributed, and secure programming for Internet".

References

1. R. Amadio, G. Boudol, and C. Lhoussaine. The receptive distributed π-calculus. In *FST&TCS*, number 1738 in Lecture Notes in Computer Science, pages 304–315, 1999.
2. C. Bryce and J. Vitek. The JavaSeal mobile agent kernel. *Autonomous Agents and Multi-Agent Systems*, 2002. To appear.
3. M. Bugliesi and G. Castagna. Secure safe ambients. In *Proc. of the 28th ACM Symposium on Principles of Programming Languages*, pages 222–235, London, 2001. ACM Press.
4. L. Cardelli, G. Ghelli, and A. Gordon. Mobility types for mobile ambients. In *Proceedings of ICALP'99*, number 1644 in Lecture Notes in Computer Science, pages 230–239. Springer, 1999.
5. L. Cardelli, G. Ghelli, and A. D. Gordon. Ambient groups and mobility types. In *International Conference IFIP TCS*, number 1872 in Lecture Notes in Computer Science, pages 333–347. Springer, August 2000.
6. L. Cardelli and A. Gordon. Mobile ambients. In *Proceedings of POPL'98*. ACM Press, 1998.
7. L. Cardelli and A. Gordon. Types for mobile ambients. In *Proceedings of POPL'99*, pages 79–92. ACM Press, 1999.
8. M. Hennessy and J. Riely. Resource access control in systems of mobile agents. *Information and Computation*, 2000. To appear.
9. F. Levi and D. Sangiorgi. Controlling interference in Ambients. In *POPL '00*, pages 352–364. ACM Press, 2000.
10. M. Merro. *Locality in the π-calculus and applications to distributed objects*. PhD thesis, École de Mines de Paris, October 2000.
11. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, Parts I and II. *Information and Computation*, 100:1–77, September 1992.
12. Robin Milner. The polyadic π-calculus: a tutorial. Technical Report ECS–LFCS–91–180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, October 1991. Appeared in *Proceedings of the International Summer School on Logic and Algebra of Specification*, Marktoberdorf, August 1991. Reprinted in *Logic and Algebra of Specification*, ed. F. L. Bauer, W. Brauer, and H. Schwichtenberg, Springer-Verlag, 1993.

13. M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *24th Ann. ACM Symp. on Principles of Programming Languages*, 1997.
14. J. Parrow and B. Victor. The Fusion Calculus: Expressiveness and symmetry in mobile processes. In *Logic in Computer Science*. IEEE Computer Society Press, 1998.
15. B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5), 1996.
16. D. Sangiorgi. The name discipline of uniform receptiveness. *Theoretical Computer Science*, 221(1–2):457–493, 1999.
17. P. Sewell and J. Vitek. Secure composition of insecure components. In *12th IEEE Computer Security Foundations Workshop*, 1999.
18. J. Vitek, C. Bryce, and W. Binder. Designing JavaSeal: or how to make Java safe for agents. In Dennis Tsichritzis, editor, *Electronic Commerce Objects*. University of Geneva, 1998.
19. J. Vitek and G. Castagna. Seal: A framework for secure mobile computations. In *Internet Programming Languages*, number 1686 in Lecture Notes in Computer Science. Springer, 1999.
20. N. Yoshida and M. Hennessy. Assigning types to processes. In *Proceedings, Fifteenth Annual IEEE Symposium on Logic in Computer Science*, pages 334–348, 2000.
21. F. Zappa Nardelli. Types for Seal Calculus. Master's thesis, Università degli Studi di Pisa, October 2000. Available at `ftp://ftp.di.ens.fr/pub/users/zappa/readings/mt.ps.gz`.