Confining Data and Processes in Global Computing Applications [⋆]

Rocco De Nicola [a]   Daniele Gorla [b,a]   Rosario Pugliese [a]

[a] *Dipartimento di Sistemi e Informatica, Università di Firenze*

[b] *Dipartimento di Informatica, Università di Roma "La Sapienza"*

Abstract

A programming notation is introduced that can be used for protecting secrecy and integrity of data in global computing applications. The approach is based on the explicit annotations of data and network nodes. Data are tagged with information about the allowed movements, network nodes are tagged with information about the nodes that can send data and spawn processes to them. The annotations are used to *confine* movements of data and processes. The approach is illustrated by applying it to three paradigmatic calculi for global computing, namely cKLAIM (a calculus at the basis of KLAIM), Dπ (a distributed version of the π-calculus) and Mobile Ambients Calculus. For all of these formalisms, it is shown that their semantics guarantees that computations proceed only while respecting confinement constraints. Namely, it is proven that, after successful static type checking, data can reside at and cross only authorised nodes. "Local" formulations of this property where only relevant sub-nets type check are also presented. Finally, the theory is tested by using it to model secure behaviours of a UNIX-like multiuser system.

Key words: Global Computing, Formal methods, Type systems, Data secrecy

[⋆] This work is partially based on a preliminary paper appeared in [15] and has been partially supported by EU within FET - Global Computing initiative, project MIKADO IST-2001-32222. The funding body is not responsible for any use that might be made of the results presented here.

Email addresses: denicola@dsi.unifi.it (Rocco De Nicola), gorla@di.uniroma1.it (Daniele Gorla), pugliese@dsi.unifi.it (Rosario Pugliese)

1 Introduction

In the design of programming languages for global computing, the integration of security mechanisms is a major challenge and great efforts have been recently devoted to embed such mechanisms within standard programming features. Several language-based security techniques have been proposed that range from type systems [19,4,2,13], to data flow analysis [16,25,9,1], from inlined reference monitoring [10] to proof-carrying code [24]. We refer the reader to [26] for an overview of some of these techniques.

The major goal of language-based security is to design languages that are flexible, expressive and safe. Unfortunately, these are often contrasting requirements. For example, mobile code deeply increases flexibility and thus expressivity of programming languages, but introduces new security problems related to unwanted accesses to classified data. Indeed, when programming has to take into account networks with mobile agents, existence in the environment of malicious principals, that can put security of data at risk, must be assumed. Malicious nodes can attack a mobile process and compromise its integrity through code modification or its secrecy through leakage of sensitive data. But one has also to take into account existence of malicious mobile processes that might attempt to access or forge private data of the network nodes hosting them.

A programming language for global computing should thus be equipped with a foundational model that also encompasses security features; the proof that an application is 'safe' could then be done by relying on formal methods. In our view, the language security model should consider existence of misbehaving entities in the execution environment of applications. Moreover, only local knowledge of the environment can be assumed because it would be impossible to collect global information in a network with possibly malicious nodes under the control of thousands different administrative authorities.

The major contribution of this paper is the definition of an approach that permits protecting the secrecy of data residing on hosting nodes and that of data carried by mobile processes by relying on program annotation. Our approach is inspired by Confined-λ [20] and relies on annotating data with sets of node addresses, called *regions*, that specify the network nodes that can interact with them. Also nodes may have annotations that specify which nodes can send data and spawn processes to them. *Data annotations* enable programmers to control the set of nodes that can share specific data, and permit shading them from other nodes. *Node annotations*, instead, enable node administrators to control the set of data and processes each node can host; thus, the node can refuse malicious processes and unwanted data.

The language semantics is then designed to guarantee that computations proceed while respecting the region constraints. For example, a process P can access a datum d only if P's execution does not export d outside the data region, say r, i.e. if P only writes d in network nodes included in r or, similarly, if P only carries d while migrating to nodes included in r. Enforcing similar constraints requires a form of code inspection that is performed, as much as possible, statically thus relieving the runtime semantics of the burden to make expensive checks and, then, improving efficiency. In Section 2 we shall introduce more details and a simple motivating example.

Our approach is largely independent of a specific model. Indeed, we shall show how it can be applied to different process calculi for mobile processes. These calculi have greatly improved the formal understanding of the complex mechanisms underlying global computing but, although share similar intentions and motivations, rest on different design choices. Usually, they permit describing both single *processes*, and *nets* of located, possibly migrating, processes. Nets are collections of *nodes* that can be thought of as physically distributed machines or as logical partitions of the same machine. Each node is referrable via an address, is connected to other nodes and hosts a set of processes. Depending on the design choices of the calculus, e.g. nets may be plain or hierarchically structured, and the notions of process, node and net may collapse. In the most basic setting, processes are built up from the empty process and from the basic *actions* by using standard operators, e.g., action prefixing, name restriction, parallel composition and replication. The basic actions permit data exchange, spawning of new, possibly remote, processes and creation of new resources.

In this paper, we apply the approach to three paradigmatic calculi for global computing, that are sufficiently different to provide evidence of the generality of our approach. We shall consider cKlaim (*core* Klaim [14], a simplified version of Klaim [8]) in Section 3, Dπ (*Distributed π-calculus* [19], a distributed variant of the π-calculus [23]) in Section 4, and Mobile Ambients Calculus [5] in Section 5. For each of these calculi, we add regions information to the syntax of terms, define a type system and an operational semantics that take annotations into account, and prove that the semantics guarantees that computations proceed only while respecting confinements. Thus, after successful static type checking, one can guarantee that data are manipulated only by authorised users. Moreover, since in such dynamic environments we cannot assume knowledge of the whole net, we also establish a more general result, namely that absence of violations of data annotations is guaranteed for all successfully type checked sub-nets, regardless of the configuration and of the evolution of the whole net they are in. In Section 6 we illustrate our approach by means of a significative example, where we model the secure implementation of a UNIX-like multiuser system. Comments on the differences between the three typing systems and about future and related work are postponed to Section 7.

2 Controlling Data Movement via Types

As stated in the Introduction, we would like to set up a machinery based on typing that helps in protecting exchanged and local data in global computing applications. To this aim, we suggest annotating data with sets of network addresses, describing the sub-net where data can be used; these sets will be called *regions*. The annotations allow programmers to fix the nodes that can share a given datum, and to avoid that the datum is accessed by untrusted processes (from untrusted nodes). Also network nodes are annotated with regions that specify the nodes that can send data and those that can spawn processes to them. This mechanism allows the administrator of a node to control the data/processes the node can host, and to refuse malicious processes and unwanted data. Thus, nodes are annotated with two regions, say r_d and r_p. We should have $r_p \subseteq r_d$ since accepting processes is, in general, more dangerous than accepting data; however, no restriction on the model is imposed to deal with this issue.

Our typing approach can be implemented by letting *regions* to be either finite subsets of addresses and input parameters or the distinct element $\top$ (used to refer to the whole net). The set of all regions $\mathcal{R}$, ranged over by r, can be partially ordered by the subset inclusion relation $\subseteq$, and has $\top$ as top element. Data annotation is rendered as $[data]_r$; we shall assume that absence of region annotations stands for $\top$.

The language semantics guarantees that computations proceed according to region constraints. This property, that we call *safety*, can be stated as

> A net N is *safe* if, for any datum d occurring in N associated to region r and for all possible evolutions of N, it holds that d will only cross and reside at nodes whose addresses are in r.

To better understand the properties we want to model and the impact of our approach on system security, we present a simple example that, for the time being, is modelled by means of a sort of abstract language (in the following sections, the same example shall be modelled by exploiting each of the calculi we mentioned in the previous section). Operator '.' stands for action prefixing while '$*$' denotes replication of processes. We assume the following process actions:

- $\textsc{Snd}(data, tgt)$ sends the information *data* by exploiting the communication medium *tgt*;
- $\textsc{Rcv}(par, tgt)$ receives from the communication medium *tgt* information that is then bound to parameter *par*;
- $\textsc{Res}(name)$ creates a fresh name *name*, and restricts its visibility to the creating process, shading the name to any other process of the net.

Moreover, we shall use function $access(\_)$ for associating 'access points' to nodes. The exact nature of the access points and of the communication medium tgt, and the way Snd and Rcv exchange data depends on the chosen communication paradigm. For instance, in case of channel-based synchronous communication, $access(addr_S)$ will return a valid channel for communicating with $addr_S$, tgt will be a suitable communication channel and actions Snd/Rcv will be executed simultaneously.

For the sake of simplicity, in the following and in the other examples we shall use polyadic communication, although we develop our theoretical framework by considering the monadic and first order variants of the calculi. Here we further rely on remote communications and on a mechanism that permits decomposing received data according to the structure of the parameters specified for receiving them.

Let us now describe the scenario we want to model. Suppose that a client C requires a service to a server S. Once S has verified the credentials of C (e.g. its identity or its credit card information), it sends back a secret password, that C can change. C could then access the service by using the last set password. This protocol can be modelled by assuming two network addresses, $addr_C$ and $addr_S$, hosting the processes P_C and P_S, respectively, that are defined below.

$$
\begin{aligned}
P_C \triangleq\ & \textsc{Snd}(addr_C,\ [creditCard\_info]_{\{addr_C, addr_S\}},\ access(addr_S)).\\
& \textsc{Rcv}(y, access(addr_C)). <\ \textit{modify password y and access the service}\ >
\end{aligned}
$$

$$
\begin{aligned}
P_S \triangleq\ & * \textsc{Rcv}(x_1, x_2, access(addr_S)). <\ \textit{check credit card info}\ x_2\ > .\\
& \quad \textsc{Res}(PWD).\textsc{Snd}([PWD]_{\{x_1, addr_S\}}\ ,\ access(x_1)).\\
& \quad <\ \textit{handle password modifications and provide the service}\ >
\end{aligned}
$$

Notice that, since the information on C's credit card is marked with region $\{addr_C, addr_S\}$, only processes at the locations of C and S will be enabled to capture C's request. Thus, no attacks mounted from other nodes aimed at cancelling the request can take place. Similar considerations do hold for the restricted name PWD that S sends back to C (it represents a secret password shared between processes at C's and S's locations).

To make our theoretical framework properly working, we need to control the processes arriving at C's and S's locations; this is why our typing discipline requires also nodes to be annotated with regions. Server S can then accept only processes coming from trusted nodes, but it should accept data coming from any user; this is necessary to model a setting where S accepts any service request, while it supplies the service only to accredited users. It has to be said that we are implicitly assuming the ability of determining the origin (the source node) of data and processes. By relying on it, we can then check compliance with regions annotations.

In the example above we have exploited remote communication; if only local communication is allowed, we would need to replace $\textsc{Snd}(\ldots, access(addr_S))$ with something like $\textsc{Exec}(\, \textsc{Snd}(\ldots, access(addr_S))\, , \, addr_S)$, supposing that $\textsc{Exec}(P, dest)$ spawns process P for execution to the node with address $dest$.

3 cKlaim: Core Klaim

We start by applying our approach to cKlaim [14], a calculus at the core of the language Klaim (Kernel Language for Agents Interaction and Mobility, [8]). The theory developed here simplifies that of [15] because the calculus only permits monadic communication and uses replication (instead of recursion) to model infinite behaviours.

The syntax of cKlaim is given in Table 1. There is only one category of names, namely that of *locality names* $\mathcal{L}$, ranged over by l. Identifiers, ranged over by ℓ, can be locality names or variables (ranged over by x), and represent both the communicable data and the target of (possibly remote) actions. T denotes templates for pattern matching and may either be a parameter $!x$, for some variable x, or a locality name. Data are represented as special processes $\langle l \rangle$, thus we may say that each node hosts processes and a (possibly empty) multiset of data. In the following, we assume that in (well formed) processes data are never prefixed by an action or replication. By using Linda [11] terminology, we shall call *tuple space* (TS, for short) the multiset of data hosted by a node and we let it to represent the repository of the node. Communication can be remote and relies on multiple distributed tuple spaces. cKlaim nodes are written $l \;{}_{r_d}{::}_{r_p}\; P$. The two region annotations control the nodes that can send data or processes to l, as established by the node administrator. Process actions are:

- $\mathbf{out}([\ell']_r)@\ell$: creates a new datum ℓ' (whose region is r) in the TS at ℓ.
- $\mathbf{in}(T)@\ell$: if $T = !x$, a datum $\langle l \rangle$ is withdrawn from the TS at ℓ and x is replaced by l in the continuation; if $T = \ell'$, then the action will look for (and retrieve) a datum $\langle \ell' \rangle$ at the TS of node ℓ (if any). This second kind of input action is a form of *name matching* operator.
- $\mathbf{eval}(P)@\ell$: spawns process P for execution to the node referred to by ℓ.
- $\mathbf{newloc}(l)$: creates a fresh locality name l that is used as the address of a new node tagged by the region annotations of the creating one and hosting process $\mathbf{nil}$.

Identifiers occurring in process terms can be *bound*. More precisely, prefix $\mathbf{in}(!x)@\ell.P$ binds variable x, while $\mathbf{newloc}(l).P$ binds locality l; in both cases,

$l, h, k, \ldots \in \mathcal{L}$			Locality names
$x, y, z, \ldots$			Variables
ℓ	$::=$	$l \mid x$	Identifiers
T	$::=$	$\ell \mid\ !\,x$	Templates
N	$::=$	$l\ {}_{r_d}{::}_{r_p}\ P \quad \mid \quad N_1 \parallel N_2$	Nets
P	$::=$		Processes
		nil	(empty)
	$\mid$	$\langle [l]_r \rangle$	(datum)
	$\mid$	$\alpha.P$	(prefixing)
	$\mid$	$P_1 \mid P_2$	(parallel composition)
	$\mid$	$*P$	(replication)
α	$::=$		Actions
		out$([\ell']_r)@\ell$	(send)
	$\mid$	**in**$(T)@\ell$	(receive)
	$\mid$	**eval**$(P)@\ell$	(execute)
	$\mid$	**newloc**(l)	(creation/restriction)

Table 1
cKlaim *Syntax*

P is the scope of the binding. An identifier that is not bound is called *free*. We let $\text{FV}(P)$ to denote the set of free variables in P. As usual, *α-conversion* allows to freely rename bound identifiers without captures. In the sequel, we shall assume that bound identifiers in processes are all distinct and different from the free ones (this is always possible by using α-conversion). Finally, we shall only consider for execution *closed nets*, i.e. nets where each occurrence of a variable is bound by an **in** prefix (similarly to many real compilers, we consider terms with free variables as programming errors). In the rest of the paper, we will omit trailing occurrences of the empty process, as usual.

3.1 *Typing* cKlaim *Nets*

The language presented in the previous section is a mean to program applications where, during the computation, a datum can only appear in localities contained in its region annotation. The runtime semantics can enforce this requirement by performing appropriate checks. These (runtime) checks are necessary because the pattern matching based communication does not permit making any static assumption on the actual structure of tuples hosted by a tuple space. To make the semantics as efficient as possible, a preliminary typing phase is introduced. Static typing of cKlaim nets aims at guaranteeing that:

(1) a datum $[l]_r$ can be seen at (i.e. can cross) ℓ if $\ell \in r$
(2) a process retrieving a datum $[l]_r$ cannot exhibit l outside r.

The typing phase performs check 1. statically and annotates parameters occurring in templates with regions to enable efficient execution of check 2. at runtime. To better distinguish the annotations put by the programmers/administrators from those put by the type system, we shall write the latter ones as superscripts and the former ones as subscripts. Hence, the syntax of templates becomes

$$T ::= \ell \mid [!\,x]^r$$

Intuitively, $[!\,x]^r$ states that the datum replacing x will cross at most the localities in r.

The typing procedure for cKlaim nets is given in Table 2. Net typings are written $N \succ N'$. The typing step includes a *type checking phase*, to verify that nets are written according to the region annotations therein, and a *type inference phase*, to annotate parameters occurring in templates. Intuitively, the inference phase takes a net N (written according to the syntax in Table 1) and returns a net N' obtained from N by annotating all the parameters with a region containing the nodes that the received values will cross. E.g., in process $\mathbf{in}(!x)@l.\mathbf{out}([x]_r)@l'$ the declaration $!x$ of variable x must be associated to region r. The type checker verifies that each process located at a node l contains only data that can be seen by l (this is done by the judgement $\succ_l$) and verifies that actions **out** and **eval** send data/code to nodes where the data/code can appear without violating the region annotations.

Judgement $\succ$ relies on an auxiliary procedure $\Gamma \vdash P \;\succ_\ell\; \Gamma' \vdash P'$ where the *type environment* Γ is a finite map from variables to regions such that $\mathrm{FV}(P) \subseteq dom(\Gamma)$. Thus, the procedure $\emptyset \vdash P \;\succ_\ell\; \emptyset \vdash P'$ is defined only if P is closed; in that case, for each parameter in P, a region annotation describing the use of that parameter in the continuation process (i.e. where it will be sent) is determined and used to decorate P (thus obtaining P'). Such regions are determined by the type inference by considering the locality where the process runs (the ℓ decorating $\succ_\ell$) and by examining the localities where the variables can appear upon execution of actions **out** and/or **eval**. Notice, however, that care is needed to avoid that closed nets become open. As an example, consider the nodes (both of them are legal)

$$l :: \mathbf{in}(!x)@l'.\mathbf{in}(!y)@l''.\mathbf{out}([x]_{\{l,y\}})@l \qquad (\star)$$

$$l :: \mathbf{in}(!y)@l'.\mathbf{out}([y]_{\{l,y\}})@l \qquad (\star\star)$$

Typing Nets:

(cK-T-Net)
$$\frac{N_1 \succ N_1' \qquad N_2 \succ N_2'}{N_1 \parallel N_2 \ \succ\ N_1' \parallel N_2'}$$

(cK-T-Node)
$$\frac{r_d, r_p \in \{\top\} \cup 2^{\mathcal{L}} \qquad \emptyset \vdash P \ \succ_l\ \emptyset \vdash P'}{l\ {}_{r_d}\!::_{r_p} P \ \succ\ l\ {}_{r_d}\!::_{r_p} P'}$$

Typing Processes:

(cK-T-Nil)
$$\frac{}{\Gamma \vdash \mathbf{nil} \ \ \succ_\ell \ \ \Gamma \vdash \mathbf{nil}}$$

(cK-T-Par)
$$\frac{\Gamma \vdash P_1 \ \succ_\ell\ \Gamma' \vdash P_1' \qquad \Gamma' \vdash P_2 \ \succ_\ell\ \Gamma'' \vdash P_2'}{\Gamma \vdash P_1|P_2 \ \ \succ_\ell \ \ \Gamma'' \vdash P_1'|P_2'}$$

(cK-T-Repl)
$$\frac{\Gamma \vdash P \ \succ_\ell\ \Gamma' \vdash P'}{\Gamma \vdash *P \ \ \succ_\ell \ \ \Gamma' \vdash *P'}$$

(cK-T-New)
$$\frac{\Gamma \vdash P \ \succ_\ell\ \Gamma' \vdash P'}{\Gamma \vdash \mathbf{newloc}(l).P \ \ \succ_\ell \ \ \Gamma' \nearrow^{l} \vdash \mathbf{newloc}(l).P'}$$

(cK-T-Datum)
$$\frac{l \in r}{\Gamma \vdash \langle [l']_r \rangle \ \ \succ_l \ \ \Gamma \vdash \langle [l']_r \rangle}$$

(cK-T-Match)
$$\frac{\Gamma \vdash P \ \succ_\ell\ \Gamma' \vdash P'}{\Gamma \vdash \mathbf{in}(\ell'')@\ell'.P \ \ \succ_\ell \ \ \Gamma' \vdash \mathbf{in}(\ell'')@\ell'.P'}$$

(cK-T-In)
$$\frac{\Gamma \uplus \{x : \{\ell\}\} \vdash P \ \succ_\ell\ \Gamma' \uplus \{x : r\} \vdash P'}{\Gamma \vdash \mathbf{in}(!x)@\ell'.P \ \ \succ_\ell \ \ \Gamma' \nearrow^{x} \vdash \mathbf{in}([!x]^{r-\{x\}})@\ell'.P'}$$

(cK-T-Out)
$$\frac{\{\ell, \ell'\} \subseteq r \qquad \Gamma \vdash P \ \succ_\ell\ \Gamma' \vdash P'}{\Gamma \vdash \mathbf{out}([\ell'']_r)@\ell'.P \ \ \succ_\ell \ \ \Gamma' + \{x : r\}_{x \in \mathrm{FV}(\ell'')} \vdash \mathbf{out}([\ell'']_r)@\ell'.P'}$$

(cK-T-Eval)
$$\frac{\ell \in \mathit{reg}(P_1) \qquad \Gamma \vdash P_1 \ \succ_{\ell'}\ \Gamma' \vdash P_1' \qquad \Gamma' \vdash P_2 \ \succ_\ell\ \Gamma'' \vdash P_2'}{\Gamma \vdash \mathbf{eval}(P_1)@\ell'.P_2 \ \ \succ_\ell \ \ \Gamma'' + \{x : \{\ell'\}\}_{x \in \mathrm{FV}(P_1)} \vdash \mathbf{eval}(P_1')@\ell'.P_2'}$$

Table 2
Typing Procedure for cKlaim

Blindly annotating these nodes would result in

$$l :: \mathbf{in}([!x]^{\{l,y\}})@l'.\mathbf{in}([!y]^{\{l\}})@l''.\mathbf{out}([x]_{\{l,y\}})@l$$
$$l :: \mathbf{in}([!y]^{\{l,y\}})@l'.\mathbf{out}([y]_{\{l,y\}})@l$$

that are open because of the occurrence of y in the regions of $!x$ and $!y$, respectively. The solution we designed to accept $(\star)$ is to assign $!x$ the region annotation $\top$. This is reasonable since $\mathbf{in}([!x]^{\{l,y\}})@l'$ means 'retrieve a datum from l' and share it with *a generic* locality of the net' (indeed y can be dynamically replaced with any locality name). The solution we designed to accept $(\star\star)$ is to remove y from $!y$ region annotation and assume that a locality can always occur in the node having that locality as address.

An anomaly somehow related to $(\star)$ is

$$l :: \mathbf{in}(!x)@l'.\mathbf{newloc}(l'').\mathbf{out}([x]_{\{l,l''\}})@l \qquad (\dagger)$$

that would result in the annotated process

$$l :: \mathbf{in}([!x]^{\{l,l''\}})@l'.\mathbf{newloc}(l'').\mathbf{out}([x]_{\{l,l''\}})@l$$

Here the problem is that the l'' occurring in the annotation associated to $!x$ by the inference system escapes from the binder **newloc** that declares l''. Thus, these two occurrences of l'' are *not* the same! For the sake of simplicity, we overcome this problem like before, i.e. by assigning $\top$ to the region annotation of $!x$.

To rule out anomalies like $(\star)$ and $(\dagger)$, in Table 2 we use function $\Gamma\nearrow^{\ell}$, that is inductively defined as

$$\begin{aligned} \emptyset \nearrow^{\ell} &\triangleq \emptyset \\ (\Gamma \uplus \{x : r\}) \nearrow^{\ell} &\triangleq \begin{cases} \Gamma\nearrow^{\ell} \ \uplus \{x : \top\} & \text{if } \ell \in r \\ \Gamma\nearrow^{\ell} \ \uplus \{x : r\} & \text{otherwise} \end{cases} \end{aligned}$$

where $\uplus$ denotes union between environments with disjoint domains.

Function $+$ extends the information of an environment through another environment and is undefined if the domain of the second environment is not included in that of the first one; formally

$$\begin{aligned} \Gamma + \emptyset &\triangleq \Gamma \\ \Gamma + \{x : r\} &\triangleq \Gamma' \uplus \{x : r \cup r'\} \qquad \text{if } \Gamma = \Gamma' \uplus \{x : r'\} \\ \Gamma + (\{x : r\} \uplus \Gamma') &\triangleq (\Gamma + \{x : r\}) + \Gamma' \end{aligned}$$

Finally, we write $\{\ldots\}_{i \in I}$ to mean $\bigcup_{i \in I}\{\ldots\}$.

Before concluding this section, we briefly comment on some typing rules. Notice that the typing of N also verifies that N is closed. Moreover, it can be easily seen that typing $P_1|P_2$ and $P_2|P_1$ yields the same typing; this relies on commutativity of sets union, since Γ grows up by union of regions. In rule (cK-T-New), the resulting environment is $\Gamma'\nearrow^{l}$ to rule out anomalies like $(\dagger)$. In rule (cK-T-In), the procedure should type P in the environment Γ extended by associating x to region $\{\ell\}$. At the end of this typing phase, the region annotation r calculated for x is associated to the parameter $!x$. Notice that x can occur in x's region r, generating anomalies like ($\star\star$); to avoid this, the annotation for x must be $r - \{x\}$. Moreover, it is possible that x occurs in region annotations within Γ' because of anomalies like ($\star$); thus, the environment resulting from this phase must be $\Gamma' \nearrow^{x}$. In rule (cK-T-Out), the type checker verifies that ℓ'' can stay both in the hosting locality ℓ and in the target locality ℓ'. The continuation process P is typed in the environment Γ, thus obtaining the annotated process P' and the environment Γ'. Hence, the result of the typing will be $\mathbf{out}([\ell'']_r)@\ell'.P'$ together with Γ' extended with the information that the variables occurring in ℓ'' (i.e. x if $\ell'' = x$) could be seen at r. Similar observations also hold for rule (cK-T-Eval) too; in particular, the check that the process can cross the locality where it is hosted is performed whenever the process is going to migrate. To this aim, we exploit the auxiliary function $reg(\_)$ that returns the intersection of the data regions occurring in its argument. Its formal definition is

$$reg(\mathbf{nil}) \triangleq \top \qquad\qquad reg(\mathbf{newloc}(l).P) \triangleq reg(P) - \{l\}$$

$$reg(\mathbf{out}([\ell']_r)@\ell.P) \triangleq r \cap reg(P) \qquad\qquad reg(\mathbf{in}(T)@\ell.P) = reg(*P) \triangleq reg(P)$$

$$reg(P_1|P_2) = reg(\mathbf{eval}(P_1)@\ell.P_2) \triangleq reg(P_1) \cap reg(P_2)$$

We deem *well-typed* those nets that successfully passed a typing phase.

Definition 1 (Well-Typed cKlaim Nets) *A net N is* well-typed *if there exists a net N' (written according to the syntax of Table 1) such that $N' \succ N$.*

3.2 cKlaim *Typed Operational Semantics*

cKlaim nets are executed according to the reduction relation $\succ\!\!\longrightarrow$ defined in Table 3. $\succ\!\!\longrightarrow$ relates configurations of the form $L \rhd N$, where L is such that $loc(N) \subseteq L \subset_{fin} \mathcal{L}$ and function $loc(N)$ returns the set of localities occurring in N. In a configuration $L \rhd N$, L is needed to ensure global freshness of new addresses. For the sake of readability, when a reduction does not generate any fresh addresses, we write $N \succ\!\!\longrightarrow N'$ instead of $L \rhd N \succ\!\!\longrightarrow L \rhd N'$. We denote with $L \uplus L'$ the disjoint union of sets L and L', and with $\succ\!\!\longrightarrow^*$ the reflexive and transitive closure of $\succ\!\!\longrightarrow$.

The semantics exploits *substitutions*, replacing variables with locality names; the substitution mapping x to l will be written as $\{l/x\}$. The application of $\{l/x\}$ to any syntactic term (variable/region/process/type environment) t, denoted by $t\{l/x\}$, replaces each free occurrence of x in t with l, with renaming of bound variables possibly involved to avoid captures. We remark that the application of a substitution to a process P also acts on the region annotations in P.

(cK-Out)

$$\frac{l \in r'_d}{l\ {}_{r_d}{::}_{r_p}\ \mathbf{out}([l'']_r)@l'.P \parallel l'\ {}_{r'_d}{::}_{r'_p}\ P' \rightarrowtail l\ {}_{r_d}{::}_{r_p}\ P \parallel l'\ {}_{r'_d}{::}_{r'_p}\ P' \,|\, \langle [l'']_r \rangle}$$

(cK-Eval)

$$\frac{l \in r'_p}{l\ {}_{r_d}{::}_{r_p}\ \mathbf{eval}(Q)@l'.P \parallel l'\ {}_{r'_d}{::}_{r'_p}\ P' \rightarrowtail l\ {}_{r_d}{::}_{r_p}\ P \parallel l'\ {}_{r'_d}{::}_{r'_p}\ P' \,|\, Q}$$

(cK-In)

$$\frac{r' \subseteq r}{l\ {}_{r_d}{::}_{r_p}\ \mathbf{in}([!x]^{r'})@l'.P \parallel l'\ {}_{r'_d}{::}_{r'_p}\ \langle [l'']_r \rangle \rightarrowtail l\ {}_{r_d}{::}_{r_p}\ P\{l''/x\} \parallel l'\ {}_{r'_d}{::}_{r'_p}\ \mathbf{nil}}$$

(cK-Match)

$$\frac{l \in r}{l\ {}_{r_d}{::}_{r_p}\ \mathbf{in}(l'')@l'.P \parallel l'\ {}_{r'_d}{::}_{r'_p}\ \langle [l'']_r \rangle \rightarrowtail l\ {}_{r_d}{::}_{r_p}\ P \parallel l'\ {}_{r'_d}{::}_{r'_p}\ \mathbf{nil}}$$

(cK-New)

$$\frac{}{L\ \triangleright\ l\ {}_{r_d}{::}_{r_p}\,\mathbf{newloc}(l').P \rightarrowtail L \uplus \{l'\}\ \triangleright\ l\ {}_{r_d \cup \{l'\}}{::}_{r_p \cup \{l'\}} P \parallel l'\ {}_{r_d \cup \{l'\}}{::}_{r_p \cup \{l'\}}\ \mathbf{nil}}$$

(cK-Call)

$$\frac{}{l\ {}_{r_d}{::}_{r_p}\ {*P} \rightarrowtail l\ {}_{r_d}{::}_{r_p}\ {*P} \,|\, P}$$

(cK-Split)

$$\frac{L \triangleright l\ {}_{r_d}{::}_{r_p}\ P_1 \parallel l\ {}_{r_d}{::}_{r_p}\ P_2 \parallel N \rightarrowtail L' \triangleright l\ {}_{r'_d}{::}_{r'_p}\ P'_1 \parallel l\ {}_{r_d}{::}_{r_p}\ P'_2 \parallel N'}{L \triangleright l\ {}_{r_d}{::}_{r_p}\ P_1 \,|\, P_2 \parallel N \rightarrowtail L' \triangleright l\ {}_{r'_d}{::}_{r'_p}\ P'_1 \,|\, P'_2 \parallel N'}$$

(cK-Par)

$$\frac{L \triangleright N_1 \rightarrowtail L' \triangleright N'_1}{L \triangleright N_1 \parallel N_2 \rightarrowtail L' \triangleright N'_1 \parallel N_2}$$

(cK-Struct)

$$\frac{N_1 \equiv N'_1 \qquad L \triangleright N'_1 \rightarrowtail L' \triangleright N'_2 \qquad N'_2 \equiv N_2}{L \triangleright N_1 \rightarrowtail L' \triangleright N_2}$$

Table 3
cKlaim *Operational Semantics*

The reduction relation relies on a *structural congruence* relation, $\equiv$, equating α-convertible processes, stating that "$\parallel$" is commutative and associative, and that **nil** acts as the identity for "$|$".

We now comment on the semantics rules. Rules (cK-Out) and (cK-Eval) say that a datum/process can be put at the target of the **out**/**eval** only if such a node accepts the datum/process (i.e. $l \in r'_d$ and $l \in r'_p$). This is necessary to prevent an untrusted node l to send data/code to l'. Notice that no static check could enforce this property without loss of expressivity: e.g., in $\mathbf{in}(!x)@l.\mathbf{eval}(\ldots)@x$, it is statically impossible to know which locality will replace x without limiting the possible exchanges at l. Thus, it cannot be determined if the locality executing the **eval** is trusted by the target locality or not. Rule (cK-In) says that a process can retrieve a datum only if the continuation process respects the datum annotation (i.e. $r' \subseteq r$). If a datum is present in the target of the action for which this check succeeds, then the datum is retrieved and replaces the input variable in the continuation; otherwise, the process is suspended until such a datum is available (if ever). Rule (cK-Match) verifies if a datum l'' is present in l'. If this is the case, the datum is removed and the continuation proceeds; otherwise, the process is suspended. Notice that, in order to complete this task, the node executing the action must be authorised by the region r. In rule (cK-New) the set L of localities already in use is exploited to choose a fresh address l' for naming the new node. Moreover, we assume that a node l trusts every node l' it creates. This is reasonable since, once created, l' is not known to any other node in the net; thus, l can use it as a *private* resource and can decide the nodes of the net that can know it (by also exploiting region annotations). For the sake of simplicity, l' is assigned the trust regions of l. However, it would be easy to extend the language for allowing the programmer to explicitly specify the trust regions of a newly created node. Rule (cK-Call) unfolds a replicated process and corresponds to a procedure call. Rule (cK-Split) permits splitting the parallel processes running at a node thus enabling the application of the main reduction rules that, in fact, can be used when there is only one thread running at l. Technically, a parallel between processes is transformed into a parallel between nodes. Rules (cK-Par) and (cK-Struct) are standard: the former says that, if part of a composed net evolves, the whole net evolves accordingly and the latter says that structural congruent nets have the same reductions.

We now give two simple properties of the operational semantics. The first one describes the relationship between the set of localities L in a configuration $L \triangleright N$ and the localities occurring in the net obtained after a reduction step. The second one describes the way parallel components located at node l could have been arrived there: they could have been either allocated at l in the initial setting or placed at l by authorised nodes as the result of subsequent computations.

Proposition 1 *If $L \triangleright N \rightarrowtail L' \triangleright N'$ and $loc(N) \subseteq L$, then $loc(N') \subseteq L'$.*

Proof: It is easy to prove that, if $L \triangleright N \rightarrowtail L' \triangleright N'$ then $L \subseteq L'$ and $loc(N') - loc(N) = L' - L$. Hence, we get $loc(N') = (loc(N') - loc(N)) \cup (loc(N') \cap loc(N)) \subseteq (L' - L) \cup loc(N) \subseteq (L' - L) \cup L' = L'$. $\square$

Proposition 2 *Let $L \triangleright N \rightarrowtail L' \triangleright N'$, $l \notin L' - L$ and $l \ {}_{r_d}\!::_{r_p} P$ be a node of N'. Then, for any parallel component P' in P it holds that: (i) either P' was located at l in the initial configuration N, or (ii) P' is a datum written at l by a node in r_d, or (iii) P' is a process spawned to l by a node in r_p.*

Proof: By a straightforward induction on the length of inference for $L \triangleright N \rightarrowtail L' \triangleright N'$ and by exploiting the premises of rules (cK-Out) and (cK-Eval). □

To conclude this section, we implement in cKlaim the example presented in Section 2. In this setting, the addresses are l_C and l_S with region annotations such that $l_S \in r_d^C$ and $l_C \in r_d^S$ (usually, $r_d^S = \top$), while $r_p^C = r_p^S = \emptyset$. Processes P_C and P_S become

$$P_C \triangleq \mathbf{out}(l_C, [cc\_info]_{\{l_C, l_S\}})@l_S.\mathbf{in}(!y)@l_C. \\ < modify\ password\ y\ and\ access\ the\ service >$$

$$P_S \triangleq *\,\mathbf{in}(!x_1, !x_2)@l_S. < check\ credit\ card\ info\ x_2 > . \\ \mathbf{newloc}(p).\mathbf{out}([p]_{\{x_1, l_S\}})@x_1. \\ < handle\ pwd\ modifications\ and\ provide\ the\ service >$$

By reasonably assuming that the password modification is carried on by only involving l_C and l_S, the inference system annotates P_C as follows:

$$P'_C \triangleq \mathbf{out}(l_C, [cc\_info]_{\{l_C, l_S\}})@l_S.\mathbf{in}([!y]^{\{l_C, l_S\}})@l_C. \cdots$$

Similarly, if we assume that credit card checking is performed locally by the server and never used anymore, P_S is annotated as:

$$P'_S \triangleq *\,\mathbf{in}(!x_1, [!x_2]^{\{l_S\}})@l_S. \cdots .\mathbf{newloc}(p).\mathbf{out}([p]_{\{x_1, l_S\}})@x_1. \cdots$$

Now, the dynamic checks of rule (cK-In) are respected; thus, the resulting net can evolve as expected:

$$\begin{array}{l} l_C \ {}_{r_d^C}\!::_{r_p^C} P'_C \parallel l_S \ {}_{r_d^S}\!::_{r_p^S} P'_S \\ \rightarrowtail^* l_C \ {}_{r_d^C}\!::_{r_p^C} \mathbf{in}([!y]^{\{l_C, l_S\}})@l_C. \cdots \parallel \\ \qquad l_S \ {}_{r_d^S}\!::_{r_p^S} P'_S \mid < check\ cc\_info > .\mathbf{newloc}(p).\mathbf{out}([p]_{\{l_C, l_S\}})@l_C. \cdots \\ \rightarrowtail^* l_C \ {}_{r_d^C}\!::_{r_p^C} < modify\ password\ p\ and\ access\ the\ service > \parallel \\ \qquad l_S \ {}_{r_d^S}\!::_{r_p^S} P'_S \mid < handle\ pwd\ modifications\ and\ provide\ the\ service > \end{array}$$

Notice that, in the reductions above, we omitted the sets L of localities in use: they can be easily inferred. Moreover, as usual, we used $\rightarrowtail^*$ to denote the reflexive and transitive closure of $\rightarrowtail$.

Our main results state that well-typedness is preserved along reductions and that well-typed nets do respect region annotations. The former result is called *subject reduction*; the latter result is called *type safety* and states that well-typedness guarantees that there are no immediate violations of data regions. Together, these results imply the *soundness* of our theory, i.e. no violation of data regions will ever occur during the evolution of well-typed nets.

We start by proving two standard technical results for a type system. The first one states that structural congruence preserves well-typedness and its proof is standard. The second one freely permits to discharge some entries from a typing environment by replacing them with localities in all terms involved.

Lemma 1 (Subject Congruence) *If N is well-typed and $N \equiv N'$ then N' is well-typed.*

Lemma 2 (Substitutivity) *If* $\Gamma \uplus \{x : r\} \vdash P \quad \succ_\ell \quad \Gamma' \uplus \{x : r'\} \vdash P'$ *and* $\sigma = \{{}^{l}\!/\!{}_{x}\}$, *then* $\Gamma\sigma \vdash P\sigma \quad \succ_{\ell\sigma} \quad \Gamma'\sigma \vdash P'\sigma$.

Proof: The proof proceeds by induction on the length of the inference used to derive the typing judgement. The base case is when only rules (cK-T-Nil) and (cK-T-Datum) are used: in both cases it is trivial to conclude. Let us consider the inductive case and reason by case analysis on the last rule used to infer the judgement. We explicitly show the most significant cases; the remaining ones are easier.

(cK-T-In): By definition, $P = \mathbf{in}(!y)@\ell'.Q$ and $P' = \mathbf{in}([!y]^{r''-\{y\}})@\ell'.Q'$, where $\Gamma \uplus \{x : r\} \uplus \{y : \{\ell\}\} \vdash Q \quad \succ_\ell \quad \Gamma'' \uplus \{x : r'''\} \uplus \{y : r''\} \vdash Q'$. By hypothesis, $y \neq x$; thus, by induction, $\Gamma\sigma \uplus \{y : \{\ell\sigma\}\} \vdash Q\sigma \quad \succ_{\ell\sigma} \quad \Gamma''\sigma \uplus \{y : r''\sigma\} \vdash Q'\sigma$. Moreover, $\Gamma' = \Gamma''\diagup^{y}$ and thus $\Gamma'\sigma = (\Gamma''\diagup^{y})\sigma = (\Gamma''\sigma)\diagup^{y}$. Hence, by using rule (cK-T-In), we can conclude the wanted $\Gamma\sigma \vdash \mathbf{in}(!y)@\ell'\sigma.Q\sigma \quad \succ_{\ell\sigma} \quad \Gamma'\sigma \vdash \mathbf{in}([!y]^{r''\sigma-\{y\}})@\ell'\sigma.Q'\sigma$.

(cK-T-Out): By definition, $P = \mathbf{out}([\ell_1]_{r_1})@\ell_2.Q$ and $P' = \mathbf{out}([\ell_1]_{r_1})@\ell_2.Q'$, where $\{\ell_1, \ell_2\} \subseteq r_1$ and $\Gamma \uplus \{x : r\} \vdash Q \quad \succ_\ell \quad \Gamma'' \uplus \{x : r_2\} \vdash Q'$. Trivially, $\{\ell_1\sigma, \ell_2\sigma\} \subseteq r_1\sigma$ and, by induction, $\Gamma\sigma \vdash Q\sigma \quad \succ_{\ell\sigma} \quad \Gamma''\sigma \vdash Q'\sigma$. We now distinguish three cases:

1. $\ell_1 \in \mathcal{L}$. In this case $\Gamma' = \Gamma''$ and $r' = r_2$; thus, $\Gamma'\sigma = \Gamma''\sigma$. By using rule (cK-T-Out), we can conclude the wanted $\Gamma\sigma \vdash \mathbf{out}([\ell_1\sigma]_{r_1\sigma})@\ell_2\sigma.Q\sigma \quad \succ_{\ell\sigma} \quad \Gamma'\sigma \vdash \mathbf{out}([\ell_1\sigma]_{r_1\sigma})@\ell_2\sigma.Q'\sigma$.
2. $\ell_1 = x$. Now $r' = r_2 \cup r_1$ but $\Gamma' = \Gamma''$; this suffices to conclude like in the previous case.
3. $\ell_1 = y \neq x$. In this case $r' = r_2$ and $\Gamma' = \Gamma'' + \{y : r_1\}$; thus, $\Gamma'\sigma = \Gamma''\sigma + \{y : r_1\sigma\}$ is defined. We can then conclude like before.

□

Theorem 1 (Subject Reduction) *If N is well-typed and $L \triangleright N \succ\!\!\longrightarrow L' \triangleright N'$ then N' is well-typed.*

Proof: The proof proceeds by induction on the length of the inference of $L \triangleright N \succ\!\!\longrightarrow L' \triangleright N'$. Notice that the sets of localities L and L' do not play any role (namely, they do not affect the definition of well-typed net) and will be ignored in the rest of the proof.

Base Step: We reason by case analysis on the axioms (i.e. the first six rules) of Table 3.

(cK-Out). By hypothesis, $N = l\ {}_{r_d}{::}_{r_p}\ \mathbf{out}([l'']_r)@l'.P \parallel l'\ {}_{r'_d}{::}_{r'_p}\ P'$ and there exists a net M such that $M \succ N$. By definition, $M = l\ {}_{r_d}{::}_{r_p}\ \mathbf{out}([l'']_r)@l'.Q \parallel l'\ {}_{r'_d}{::}_{r'_p}\ Q'$ where $\emptyset \vdash \mathbf{out}([l'']_r)@l'.Q \succ_l \emptyset \vdash \mathbf{out}([l'']_r)@l'.P$ and $\emptyset \vdash Q' \succ_{l'} \emptyset \vdash P'$. By the premises of rule (cK-T-Out), $\emptyset \vdash Q \succ_l \emptyset \vdash P$ and $l' \in r$. This suffices to conclude that $N' = l\ {}_{r_d}{::}_{r_p}\ P \parallel l'\ {}_{r'_d}{::}_{r'_p}\ P' \mid \langle [l'']_r \rangle$ is well-typed.

(cK-Eval). This case is similar. Indeed, by the premise of rule (cK-T-Eval), it holds that there exists a process Q' such that $\emptyset \vdash Q' \succ_{l'} \emptyset \vdash Q$.

(cK-In). By hypothesis, N results from the typing of a net $M = l\ {}_{r_d}{::}_{r_p}\ \mathbf{in}(!x)@l'.Q \parallel l'\ {}_{r'_d}{::}_{r'_p}\ \langle [l'']_r \rangle$. The main thing to prove is that the well-typedness of $l\ {}_{r_d}{::}_{r_p}\ \mathbf{in}([!x]^{r'})@l'.P$ implies the well-typedness of $l\ {}_{r_d}{::}_{r_p}\ P\{l''/x\}$. By the premise of rule (cK-T-In), it holds that $\{x : \{l\}\} \vdash Q \succ_l \{x : r''\} \vdash P$ for $r' = r'' - \{x\}$. Hence, by Lemma 2, $\emptyset \vdash Q\{l''/x\} \succ_l \emptyset \vdash P\{l''/x\}$. This suffices to conclude.

(cK-Match) and (cK-New). These cases are easy.

(cK-Call). By hypothesis, there exists a process Q such that $\emptyset \vdash *Q \succ_l \emptyset \vdash *P$. By the premise of rule (cK-T-Repl), $\emptyset \vdash Q \succ_l \emptyset \vdash P$. Thus, by using rule (cK-T-Par), we can conclude.

Inductive Step: We reason by case analysis on the last applied inference rule of Table 3.

(cK-Split). By hypothesis, we have that $N = l\ {}_{r_d}{::}_{r_p}\ P_1|P_2 \parallel N''$ results from the typing of a net $M = l\ {}_{r_d}{::}_{r_p}\ Q_1|Q_2 \parallel M''$. In particular, $\emptyset \vdash Q_1|Q_2 \succ_l \emptyset \vdash P_1|P_2$ that, by rule (cK-T-Par), implies that $\emptyset \vdash Q_1 \succ_l \Gamma \vdash P_1$ and $\Gamma \vdash Q_2 \succ_l \emptyset \vdash P_2$. However, Γ must be $\emptyset$ as well; indeed, it can be easily checked that $\Gamma_1 \vdash P' \succ_l \Gamma_2 \vdash P''$ implies $dom(\Gamma_1) = dom(\Gamma_2)$. Hence, $l\ {}_{r_d}{::}_{r_p}\ P_1 \parallel l\ {}_{r_d}{::}_{r_p}\ P_2 \parallel N''$ is well-typed and, by induction, $l\ {}_{r'_d}{::}_{r'_p}\ P'_1 \parallel l\ {}_{r_d}{::}_{r_p}\ P'_2 \parallel N'$ is well-typed. This implies that $l\ {}_{r'_d}{::}_{r'_p}\ P'_1|P'_2 \parallel N'$ is well-typed, as required.

(cK-Par) and (cK-Struct). By a straightforward induction; the latter case relies on Lemma 1. □

We now turn to type safety. As we have already said, it states that well-typedness guarantees absence of immediate violations of data regions. However, the wanted safety property requires that data regions are respected along all possible computations. To properly formalise this property we need to define a finer semantics. Indeed, deeming a net to be safe when "for any node $l \;{}_{r_d}{::}_{r_p}\; P$ it holds that l occurs in the region of each datum in P" would not be satisfactory because the regions annotating data disappear upon data withdrawal. Thus, it would become impossible to formalise the requirement that the region specification associated to a datum when it is produced is respected during all the datum life-time (i.e. also after its retrieval). For example, consider the net $N = l \;{}_{r_d}{::}_{r_p}\; \mathbf{in}([!x]^{r'})@l'.P \parallel l' \;{}_{r'_d}{::}_{r'_p}\; \langle [l'']_r \rangle$. Upon execution of action **in**, the net becomes $N' = l \;{}_{r_d}{::}_{r_p}\; P' \parallel l' \;{}_{r'_d}{::}_{r'_p}\; \mathbf{nil}$, where $P' = P\{l''/x\}$. Now, all the occurrences of l'' in P' are *not* annotated anymore with region r. Hence, in N' we have no mean to formalise the statement that l can use l'' by respecting the original annotation r.

In other terms, with the calculus introduced in Table 1, we cannot express absence of data regions violations syntactically, because in general we lack information about the region that originally annotated data carried by processes. To overcome this problem, we design a *tagged language*, where each occurrence of a locality in a process is tagged with a region determining its visibility. To this aim, we slightly adapt the syntax of cKlaim, by letting identifiers to be

$$\ell \;::=\; [l]_r \;\mid\; x$$

We can now formalise when a net is safe. To this aim, we extend function *reg* defined in Section 3.1 by taking into account also the locality tags when calculating the region intersection. For example, $reg(\mathbf{out}([[l]_{r_1}]_{r_2})@[l']_{r_3}.P) = r_1 \cap r_2 \cap r_3 \cap reg(P)$. Moreover, we let $reg(\langle [l]_r \rangle) = r$.

Definition 2 (Safety) *A net N is* safe *if for any $l \;{}_{r_d}{::}_{r_p}\; P$ in N, it holds that $l \in reg(P)$.*

The tagged semantics generalises that in Table 3. Indeed, processes like $\mathbf{out}([[l]_{r_1}]_{r_2})@[l']_{r_3}$ or $\mathbf{in}([l]_{r_1})@[l']_{r_2}$ can evolve. These terms may arise upon application of substitutions that now map variables into localities tagged with regions. We let the application of the substitution to a region to replace variables only with localities (hence omitting their tags) thus ensuring that regions are still sets of identifies. The reduction relation, however, ignores the tags and considers tagged names as plain ones. This should have been somehow expected because, as we said before, the only role of tags is to enable formalising and checking that a net is safe. Thus, rules (cK-Out) and (cK-In) now become

$$\frac{l \in r'_d}{l \;{}_{r_d}{::}_{r_p}\; \mathbf{out}([[l'']_{r_1}]_{r_2})@[l']_{r_3}.P \parallel l' \;{}_{r'_d}{::}_{r'_p}\; P' \;\rightarrowtail\!\!\!\!\rightarrow\; l \;{}_{r_d}{::}_{r_p}\; P \parallel l' \;{}_{r'_d}{::}_{r'_p}\; P' \mid \langle [l'']_{r_2} \rangle}$$

$$\frac{r_1 \subseteq r}{l \;{}_{r_d}{::}_{r_p}\; \mathbf{in}([!x]^{r_1})@[l']_{r_2}.P \parallel l' \;{}_{r'_d}{::}_{r'_p}\; \langle [l'']_r \rangle \;\rightarrowtail\!\!\!\!\rightarrow\; l \;{}_{r_d}{::}_{r_p}\; P\{[l'']_r/x\} \parallel l' \;{}_{r'_d}{::}_{r'_p}\; \mathbf{nil}}$$

$$\frac{pid(\ell) \in reg(\ell') \cap reg(\ell'') \qquad \Gamma \vdash P \ \ggcurly_{\ell}\ \Gamma' \vdash P'}{\Gamma \vdash \mathbf{in}(\ell'')@\ell'.P \ \ggcurly_{\ell}\ \Gamma' \vdash \mathbf{in}(\ell'')@\ell'.P'}$$

$$\frac{pid(\ell) \in reg(\ell') \qquad \Gamma \uplus \{x : \{pid(\ell)\}\} \vdash P \ \ggcurly_{\ell}\ \Gamma' \uplus \{x : r\} \vdash P'}{\Gamma \vdash \mathbf{in}(!x)@\ell'.P \ \ggcurly_{\ell}\ \Gamma' \nearrow^{x} \vdash \mathbf{in}([!x]^{r-\{x\}})@\ell'.P'}$$

$$\frac{pid(\ell) \in reg(\ell') \qquad \{pid(\ell), pid(\ell')\} \subseteq r \qquad \Gamma \vdash P \ \ggcurly_{\ell}\ \Gamma' \vdash P'}{\Gamma \vdash \mathbf{out}([x]_r)@\ell'.P \ \ggcurly_{\ell}\ \Gamma' + \{x : r\} \vdash \mathbf{out}([x]_r)@\ell'.P'}$$

$$\frac{pid(\ell) \in reg(\ell') \qquad \{pid(\ell), pid(\ell')\} \subseteq r_2 \subseteq r_1 \qquad \Gamma \vdash P \ \ggcurly_{\ell}\ \Gamma' \vdash P'}{\Gamma \vdash \mathbf{out}([[l]_{r_1}]_{r_2})@\ell'.P \ \ggcurly_{\ell}\ \Gamma' \vdash \mathbf{out}([[l]_{r_1}]_{r_2})@\ell'.P'}$$

$$\frac{pid(\ell) \in reg(\ell') \cap reg(P_1) \qquad \Gamma \vdash P_1 \ \ggcurly_{\ell'}\ \Gamma' \vdash P_1' \qquad \Gamma' \vdash P_2 \ \ggcurly_{\ell}\ \Gamma'' \vdash P_2'}{\Gamma \vdash \mathbf{eval}(P_1)@\ell'.P_2 \ \ggcurly_{\ell}\ \Gamma'' + \{x : \{pid(\ell')\}\}_{x \in \mathrm{FV}(P_1)} \vdash \mathbf{eval}(P_1')@\ell'.P_2'}$$

Table 4
Tagged Typing Rules

To avoid confusion, we use the arrow $\rightarrowtail\!\!\!\twoheadrightarrow$ to relate tagged terms. The other rules extend those in Table 3 in the expected way.

The typing procedure for tagged terms is denoted by $\ggcurly$ and its most significant rules are given in Table 4 (the other ones are smooth adaptations of those in Table 2). We use functions $pid(\ell)$ and $reg(\ell)$ to denote, respectively, the plain identifier and the region of the tagged identifier ℓ. The intuition underlying $\ggcurly$ is that, whenever an identifier occurs at a locality, the locality must be included in the region tagging the identifier.

Given a plain net N, we use $tag(N)$ to denote the set containing all the well-typed (w.r.t. $\ggcurly$) tagged nets obtained by tagging localities in N. Given a tagged net N, we denote with $untag(N)$ the plain net obtained from N by removing all the locality tags. Notice that $tag(N)$ is not empty because it contains at least the net obtained by tagging each locality in N with $\top$. We call the latter net the *outset tagging* of N.

Predictably, the tagged language and the original one are strongly related. Moreover, the typing of tagged terms is preserved along (tagged) reductions. The following results formalises these properties.

Proposition 3

(1) If $N \ggcurly M$ then $untag(N) \succ untag(M)$.

(2) If $N \succ M$, then for all $M' \in tag(M)$ there exists $N' \in tag(N)$ such that $N' \ggcurly M'$.
(3) If $L \triangleright N \succ\!\!\twoheadrightarrow L' \triangleright N'$ then $L \triangleright untag(N) \succ\!\!\longrightarrow L' \triangleright untag(N')$.

Proof: All properties easily follow from definitions of $\ggcurly$ and $\succ\!\!\twoheadrightarrow$. □

Corollary 1 (Tagged Subject Reduction) *If N is a well-typed tagged net and $L \triangleright N \succ\!\!\twoheadrightarrow L' \triangleright N'$ then N' is a well-typed tagged net.*

Proof: By Propositions 3.1 and .3, it holds that $untag(N)$ is well-typed and that $L \triangleright untag(N) \succ\!\!\longrightarrow L' \triangleright untag(N')$. Because of Theorem 1, this implies that $untag(N')$ is well-typed and, by Proposition 3.2, we can conclude. □

We are now ready to prove the type safety theorem.

Theorem 2 (Type Safety) *If N is a well-typed tagged net then N is safe.*

Proof: By definition, N is a well-typed (tagged) net if there exists a net M such that $M \ggcurly N$. The proof proceeds by induction on the length of the inference leading to this judgement and heavily relies on checking the premise $pid(\ell) \in reg(\ell')$ contained in each rule of Table 4. □

Corollary 2 (Type Soundness) *Let N be a (plain) well-typed net and N' be its outset tagging. Then $L \triangleright N' \succ\!\!\twoheadrightarrow^* L' \triangleright N''$ implies that N'' is safe.*

Proof: By Proposition 3.2 and by the fact that $N' \in tag(N)$, it holds that N' is a well-typed tagged net. We now proceed by induction on the length of $\succ\!\!\twoheadrightarrow^*$. The base case is Theorem 2; the inductive case trivially follows by exploiting Corollary 1. □

The results given above can be generalised by requiring only a subnet of the whole net to be well-typed. By using the convention that absence of a region annotation means $\top$, a not well-typed net can be executed according to the (tagged versions of) rules in Table 3 by safely considering all its variable annotations as $\top$. We call *r-subnet* of N the net formed by all the nodes $l\ {}_{r_d}\!::_{r_p}\ P$ in N such that $\{l\} \cup r_d \cup r_p \subseteq r$. Notice that such a net is not necessarily defined for all r; of course it is always defined for $r = \top$ and coincides with N (in this case Theorem 3 coincides with Corollary 2).

Theorem 3 (Localised Type Soundness) *Let N be a plain net and N' be its outset tagging. If the r-subnet of N' is defined and well-typed, and if $L \triangleright N' \succ\!\!\twoheadrightarrow^* L' \triangleright N''$, then the r'-subnet of N'' is defined and safe, where $r' = r \cup (L' - L)$.*

Proof: By exploiting Theorem 2, we only need to show that the r'-subnet of N'' is defined and well-typed. We just consider the case for $L \triangleright N' \rightarrowtail\!\!\!\twoheadrightarrow L' \triangleright N''$; the more general case is recovered by using an inductive argument similar to that in Corollary 2. The proof proceeds like that of Theorem 1. Just notice that, when the operational rule used to infer the reduction is (the tagged version of) (cK-Out) or (cK-Eval) resp., the premise $l \in r'_d$ or $l \in r'_p$ respectively turns out to be crucial to maintain well-typedness. Moreover, the only non trivial case for establishing if the r'-subnet is defined, is when the operational rule used is (cK-New). In this case, the claim is easily proved since the new node is assigned the regions of the creating one. $\square$

To conclude, we want to remark that the language can be easily extended to enable explicit specification of the regions of the new nodes. In this case, existence of the r'-subnet could be ensured by adding a premise to rule (cK-New) requiring that the regions of the new nodes are included in those of the creating node.

4 Dπ: Distributed π-calculus

We now apply our approach to Dπ [19], a variant of the π-calculus [23] with process distribution and mobility. The syntax of Dπ is given in Table 5. There are two categories of names: *locality names* $\mathcal{L}$, ranged over by l, and *channel names* $\mathcal{C}$, ranged over by a. The symbol e is used for channel or locality names, while u, v, called *identifiers*, denote names and *variables* (ranged over by x). The exchanged messages, ranged over by W, can be both identifiers and compound identifiers of the form $v@u$ (where u is expected to be a locality name or variable, while v is expected to be a channel name or variable). Similarly, input parameters, generically referred to as X, can either be a simple variable x or a compound variable $z@y$ (y is a locality variable and z is a channel variable). Dπ nodes (*located threads*, in the original terminology) will be written as $l^{r_p}_{r_d}[\![P]\!]$. Communication is local, synchronous and channel based. Process actions are:

- $u!\langle [W]_r \rangle$: makes available message W (with associated region r) along the channel u of the locality where the action is fired.
- $u?(X)$: retrieves a message W from the channel u of the locality where the action is fired and replaces the parameter X with the message in the continuation process. If X is a variable x, the message retrieved must be a name e. Otherwise, if X is $z@y$, then the message must be of the form $a@l$, and z will be replaced by a and y will be replaced by l.
- **go** u: spawns the continuation process for execution at the node referred to by u.

$l, h, k, \ldots \in \mathcal{L}$			Locality names
$a, b, c, \ldots \in \mathcal{C}$			Channel names
$x, y, z, \ldots$			Variables
e	$::=$	$l \mid a$	Names
u	$::=$	$e \mid x$	Identifiers
X	$::=$	$x \mid z@y$	Input parameters
W	$::=$	$u \mid v@u$	Messages
N	$::=$	$l^{r_p}_{r_d}[\![P]\!] \quad \mid \quad N_1 \parallel N_2$	Nets
P	$::=$		Processes
		stop	(empty)
	$\mid$	$\alpha.P$	(prefixing)
	$\mid$	$P_1 \mid P_2$	(parallel composition)
	$\mid$	$(\nu e)P$	(restriction)
	$\mid$	$*P$	(replication)
α	$::=$		Actions
		$u!\langle [W]_r \rangle$	(send)
	$\mid$	$u?(X)$	(receive)
	$\mid$	**go** u	(migrate)

Table 5
Dπ Syntax

Identifiers occurring in process terms can be *bound*; more precisely, prefix $u?(X).P$ binds the variables in X (i.e. it binds x if $X = x$ and binds both y and z if $X = z@y$), while $(\nu e)P$ binds name e; in both cases, P is the scope of the binding. The set of free variables $\mathrm{FV}(\_)$, α-conversion and closed nets are defined accordingly.

To conclude the presentation, we want to argue for the need to associate two regions to each Dπ node. Indeed, differently from cKlaim, no remote operation is allowed (a part, of course, process spawning), hence the data region could seem useless. However, using only the process region would be too restrictive: in fact, if a node l does not know or trust another node k, then k has no mean to come into contact with l. Our solution permits to distinguish generic processes from processes that are not very risky because, for example, they only perform an output and then terminate. These last processes are of the form $u!\langle W \rangle.$**stop** and we deem them *output processes*. However, processes of different form could be accepted as well: e.g. process $u!\langle W\rangle.v!\langle W'\rangle.\mathbf{stop}$ is as risky as $u!\langle W\rangle.\mathbf{stop}$. Since we do not want to take a definite standing on the set of output processes, we use a predicate $output(P)$, that holds true if and only if P is an output process, but leave aside its exact definition. Thus, output processes coming from k are accepted by node $l^{r_p}_{r_d}[\![\cdot]\!]$ only if $k \in r_d$; all the other processes are accepted if $k \in r_p$ (see rule (D-T-Go) in Table 6).

4.1 Typing Dπ Nets

The typing system for cKlaim of Table 2 could be straightforwardly adapted to deal with Dπ nets; see Remark 1 in Section 4.2. However, in a channel-based setting region compatibility checks can be statically performed (on the contrary, they are dynamically performed in cKlaim – see rule (cK-In)) because it is natural to associate each channel with a region annotation describing the region of the data exchangeable along it. Thus, if a channel a can carry data visible within r, then messages with region $r_1 \supseteq r$ can be sent along a and input parameters with region $r_2 \subseteq r$ can be used to retrieve data from a. By transitivity, we get $r_2 \subseteq r_1$ thus ensuring that the use of the data respects the specifications of the data region. Hence, in this setting, parameters do not need to be annotated because the correct use of the data can be statically enforced by the typing system.

To properly deal with name passing, we take advantage of some of the theory from [19]. The resulting type system is very different from that in Table 2 but shows how our approach can be adapted to different languages. We assume the following types:

$$\begin{array}{ll} \text{TYPES:} & \tau ::= \phi \mid \gamma \mid \gamma@\phi \\ \text{LOCALITY TYPES:} & \phi ::= r \triangleright r'[\widetilde{u:\gamma}]^{r_p}_{r_d} \\ \text{CHANNEL TYPES:} & \gamma ::= r(\tau) \end{array}$$

Intuitively, if v has type $r(\tau)$ then it is a channel that can be seen by nodes in r and can carry messages of type τ. Similarly, if v has type $r \triangleright r'[\widetilde{u:\gamma}]^{r_p}_{r_d}$ then it is a locality whose name can be one of the names in r' (this is useful only when v is a variable; if v is a name, then $r' = \{v\}$, see requirement (‡) below), that can be seen by nodes in r, accepts data/code from nodes in r_d/r_p resp., and hosts channels $\widetilde{u}$, in an orderly way of types $\widetilde{\gamma}$. As usual, $\widetilde{\_}$ denotes a (possibly empty) set of entities $\_$. Finally, $\gamma@\phi$, with $\phi = r \triangleright r'[\widetilde{u:\gamma}]^{r_p}_{r_d}$, can be assigned to a message $u@v$ where u is a channel of type γ and v is a locality of type $r \triangleright r'[\widetilde{u:\gamma}, u:\gamma]^{r_p}_{r_d}$.

For a type τ, we let $reg(\tau)$ to denote the region that can see values of type

Typing Nets:

(D-T-Net)
$$\frac{\Gamma \vdash N_1 \qquad \Gamma \vdash N_2}{\Gamma \vdash N_1 \parallel N_2}$$

(D-T-Node)
$$\frac{\Gamma(l) = r \triangleright r'[\widetilde{a:\gamma}]_{r_d}^{r_p} \qquad \Gamma \vdash_l P}{\Gamma \vdash l_{r_d}^{r_p}[\![P]\!]}$$

Typing Processes:

(D-T-Nil)
$$\frac{}{\Gamma \vdash_u \textbf{stop}}$$

(D-T-Repl)
$$\frac{\Gamma \vdash_u P}{\Gamma \vdash_u *P}$$

(D-T-Par)
$$\frac{\Gamma \vdash_u P_1 \qquad \Gamma \vdash_u P_2}{\Gamma \vdash_u P_1|P_2}$$

(D-T-CRes)
$$\frac{\Gamma,\ {}_u a : \gamma \vdash_u P}{\Gamma \vdash_u (\nu a)P}$$

(D-T-LRes)
$$\frac{\Gamma,\ {}_u l : \top \triangleright \{l\}[\emptyset]_{dreg(\Gamma(u))\cup\{l\}}^{preg(\Gamma(u))\cup\{l\}} \vdash_u P}{\Gamma \vdash_u (\nu l)P}$$

(D-T-In)
$$\frac{\Gamma \vdash_u v : r(\tau) \qquad val(\Gamma(u)) \subseteq r \qquad \Gamma,\ {}_u X : \tau \vdash_u P}{\Gamma \vdash_u v?(X).P}$$

(D-T-Out)
$$\frac{\begin{array}{l}\Gamma \vdash_u v : r'(\tau) \qquad val(\Gamma(u)) \subseteq r' \qquad \Gamma \vdash_u W : \tau' \qquad \tau \sqsubseteq \tau' \\ u \in r \qquad \bigcup_{w \in r} val(\Gamma(w)) \subseteq reg(\tau) \qquad \Gamma \vdash_u P\end{array}}{\Gamma \vdash_u v!\langle [W]_r \rangle.P}$$

(D-T-Go)
$$\frac{val(\Gamma(u)) \subseteq reg_\Gamma^u(\textbf{go}\, v.P) \qquad \begin{array}{r}\text{if } output(P) \text{ then } val(\Gamma(u)) \subseteq dreg(\Gamma(v)) \\ \text{else } val(\Gamma(u)) \subseteq preg(\Gamma(v))\end{array} \qquad \Gamma \vdash_v P}{\Gamma \vdash_u \textbf{go}\, v.P}$$

Typing Messages:

(D-T-Chan)
$$\frac{\Gamma(u)(v) = \gamma}{\Gamma \vdash_u v : \gamma}$$

(D-T-Loc)
$$\frac{\Gamma(v) = \phi}{\Gamma \vdash_u v : \phi}$$

(D-T-Compound)
$$\frac{\Gamma(w) = r \triangleright r'[\widetilde{v':\gamma'}, v : \gamma]_{r_d}^{r_p}}{\Gamma \vdash_u v@w : \gamma@r \triangleright r'[\widetilde{v':\gamma'}]_{r_d}^{r_p}}$$

Table 6
Type Checking for Dπ

τ, i.e. $reg(r(\tau)) = reg(r \triangleright r'[\widetilde{u:\gamma}]_{r_d}^{r_p}) \triangleq r$. Similarly, for a locality type $\phi = r \triangleright r'[\widetilde{u:\gamma}]_{r_d}^{r_p}$, we let $val(\phi)$, $dreg(\phi)$ and $preg(\phi)$ to denote, resp., regions r', r_d and r_p.

The typing system for Dπ nets is given in Table 6. The main judgement is $\Gamma \vdash N$, stating that N is well-typed in the environment Γ. A *type environment* is a finite partial function mapping locality names and variables to locality types. Therefore, since locality types contain information about the allocated channels, it is also possible to extract from a typing environment the channel

Environment Extension :

$$\Gamma,\ {}_u w : \phi \ \triangleq\ \Gamma' \quad \text{s.t. } \Gamma'(v) = \begin{cases} \Gamma(v) & \text{if } v \neq w \text{ and } w \notin \mathit{dom}(\Gamma) \\ \phi & \text{if } v = w \notin \mathit{dom}(\Gamma) \end{cases}$$

$$\Gamma,\ {}_u w : \gamma \ \triangleq\ \Gamma' \quad \text{s.t. } \Gamma'(v) = \begin{cases} \Gamma(v) & \text{if } v \neq u \\ r \triangleright r'[w : \gamma, \widetilde{w' : \gamma'}]^{r_p}_{r_d} & \text{if } v = u \text{ and } w \notin \widetilde{w'} \\ & \text{and } \Gamma(u) = r \triangleright r'[\widetilde{w' : \gamma'}]^{r_p}_{r_d} \end{cases}$$

$$\Gamma,\ {}_u x_1 @ x_2 : \gamma @ \phi \ \triangleq\ \Gamma' \quad \text{s.t. } \Gamma'(v) = \begin{cases} \Gamma(v) & \text{if } v \neq x_2 \text{ and } x_2 \notin \mathit{dom}(\Gamma) \\ r \triangleright r'[x_1 : \gamma, \widetilde{u : \gamma}]^{r_p}_{r_d} & \text{if } v = x_2 \notin \mathit{dom}(\Gamma) \\ & \text{and } \phi = r \triangleright r'[\widetilde{u : \gamma}]^{r_p}_{r_d} \end{cases}$$

Subtyping Relation :

$$\tau \sqsubseteq \tau \qquad\qquad \frac{\tau \sqsubseteq \tau' \qquad \tau' \sqsubseteq \tau}{\tau = \tau'} \qquad\qquad \frac{\tau_1 \sqsubseteq \tau_2 \qquad \tau_2 \sqsubseteq \tau_3}{\tau_1 \sqsubseteq \tau_3}$$

$$\frac{r \subseteq s \qquad r' \supseteq s' \qquad r_d \subseteq s_d \qquad r_p \subseteq s_p \qquad n \leq m \qquad \forall i = 1, \cdots, n.\gamma_i \sqsubseteq \gamma'_i}{r \triangleright r'[u_1 : \gamma_1, \ldots, u_n : \gamma_n]^{r_p}_{r_d} \quad \sqsubseteq \quad s \triangleright s'[u_1 : \gamma'_1, \ldots, u_m : \gamma'_m]^{s_p}_{s_d}}$$

$$\frac{r \subseteq r' \quad \tau \sqsubseteq \tau'}{r(\tau) \sqsubseteq r'(\tau')} \qquad\qquad \frac{\gamma \sqsubseteq \gamma' \quad \phi \sqsubseteq \phi'}{\gamma @ \phi \sqsubseteq \gamma' @ \phi'}$$

Table 7
Technicalities of Dπ Typing

types associated to channel names and variables. In particular, if $\Gamma(u) = r \triangleright r'[v : \gamma, \widetilde{v' : \gamma'}]^{r_p}_{r_d}$, we shall write $\Gamma(u)(v) = \gamma$. We shall only consider typing environments satisfying the following constraint:

> Let $\Gamma(v) = r \triangleright r'[\widetilde{u : \gamma}]^{r_p}_{r_d}$. If $v \in \mathcal{L}$ then $r' = \{v\}$; otherwise, for each $l \in r'$, it must be that $r \subseteq \mathit{reg}(\Gamma(l))$ and $r_d \subseteq \mathit{dreg}(\Gamma(l))$ and $r_p \subseteq \mathit{preg}(\Gamma(l))$. (‡)

This condition states that the component r' is really useful only when v is a variable; in this case, it collects the possible names v can assume at run-time. Moreover, it states that regions $r/r_d/r_p$ must respect the corresponding specifications contained in Γ for all the values v can assume.

We assume that $\Gamma \vdash N$ holds true only if Γ satisfies (‡), Γ does not contain variables and $\text{FV}(N) = \emptyset$. The main judgement relies on two auxiliary judgements for typing processes and messages. Judgement $\Gamma \vdash_u P$ states that P can be properly executed at u while respecting Γ; we always assume that $\text{FV}(P) \subseteq \mathit{dom}(\Gamma)$. Judgement $\Gamma \vdash_u W : \tau$ states that message W can be assigned type τ at u under the assumptions Γ. Some aspects, like the extension of an environment with a new item (written $\Gamma,\ {}_uW : \tau$) and the subtyping relation (written $\tau \sqsubseteq \tau'$), have been straightforwardly adapted from [19] and are given in Table 7. We omit comments on these features and refer the interested reader to [19].

We now briefly comment on some of the typing rules. In rule (D-T-LRes), we assume that the created node is assigned the regions of the creating one (this is similar to cKlaim – see rule (cK-New)). In rule (D-T-In), it is checked that u can access channel v (the fact that v is a channel is ensured by the fact that Γ indirectly assigns v a channel type through the type of the locality where v is placed) and that the continuation properly uses the received message (i.e. P is typeable in an environment obtained by extending Γ with the information that X has type at most τ, the type of the value carried by v). Similarly, in rule (D-T-Out), it is checked that u can access channel v, that message W can be assigned at least type τ in u by Γ, that u can see W and that the region specified for W is at most the region of the values that v can carry. Finally, rule (D-T-Go) verifies that u can see v and all the identifiers occurring in P by exploiting function $reg^u_\Gamma(\cdot)$ defined inductively as follows:

$$reg^u_\Gamma(\mathbf{stop}) \triangleq \top \qquad reg^u_\Gamma(*P) \triangleq reg^u_\Gamma(P) \qquad reg^u_\Gamma(P_1|P_2) \triangleq reg^u_\Gamma(P_1) \cap reg^u_\Gamma(P_2)$$

$$reg^u_\Gamma((\nu e)P) \triangleq reg^u_\Gamma(P) - \{e\} \qquad reg^u_\Gamma(\mathbf{go}\, v.P) \triangleq reg(\Gamma(v)) \cap reg^v_\Gamma(P)$$

$$reg^u_\Gamma(v!\langle[W]_r\rangle.P) \triangleq reg(\Gamma(u)(v)) \cap r \cap reg^u_\Gamma(P)$$

$$reg^u_\Gamma(v?(X).P) \triangleq reg(\Gamma(u)(v)) \cap reg^u_\Gamma(P)$$

The premises of the rule also check if u can send P to v (by exploiting the data/process region of v according to the fact that P is an output process or not) and if P typechecks at v.

To conclude, we define *well-typed* Dπ nets.

Definition 3 (Well-Typed Dπ Nets) *A net N is* well-typed in Γ *if* $\Gamma \vdash N$. *A net N is* well-typed *if there exists a typing environment Γ such that N is well-typed in Γ.*

4.2 Dπ Operational Semantics

Dπ nets evolve according to the reduction relation $\longmapsto$ defined in Table 8. Like in cKlaim, $\longmapsto$ relates configurations of the form $K \triangleright N$, where K is a set of localities and localised channels (thus, $K = \{l_1, l_2, \ldots, a_1@l_1, a_2@l_2, \ldots\}$) such that $n(N) \subseteq K \subset_{fin} \mathcal{L} \cup (\mathcal{C} \times \mathcal{L})$, and function $n(N)$ returns the set of all (possibly compound) names occurring in N. For example, a suitable K for

(Dπ-Comm)

$$\frac{}{l^{r_p}_{r_d}[\![a?(X).P \mid a!\langle[W]_r\rangle.Q]\!] \;\longmapsto\; l^{r_p}_{r_d}[\![P\{^W\!/X\} \mid Q]\!]}$$

(Dπ-Go)

$$\frac{}{l^{r_p}_{r_d}[\![\mathbf{go}\ k.P]\!] \parallel k^{r'_p}_{r'_d}[\![Q]\!] \;\longmapsto\; l^{r_p}_{r_d}[\![\mathbf{stop}]\!] \parallel k^{r'_p}_{r'_d}[\![P \mid Q]\!]}$$

(Dπ-NewLoc)

$$\frac{}{K \;\triangleright\; l^{r_p}_{r_d}[\![(\nu k)P]\!] \;\longmapsto\; K \uplus \{k\} \;\triangleright\; l^{r_p\cup\{k\}}_{r_d\cup\{k\}}[\![P]\!] \parallel k^{r_p\cup\{k\}}_{r_d\cup\{k\}}[\![\mathbf{stop}]\!]}$$

(Dπ-NewChan)

$$\frac{}{K \;\triangleright\; l^{r_p}_{r_d}[\![(\nu a)P]\!] \;\longmapsto\; K \uplus \{a@l\} \;\triangleright\; l^{r_p}_{r_d}[\![P]\!]}$$

(Dπ-Call)

$$\frac{}{l^{r_p}_{r_d}[\![*P]\!] \;\longmapsto\; l^{r_p}_{r_d}[\![*P \mid P]\!]}$$

(Dπ-Split)

$$\frac{K \triangleright l^{r_p}_{r_d}[\![P]\!] \parallel l^{r_p}_{r_d}[\![Q]\!] \parallel N \;\longmapsto\; K' \triangleright l^{r'_p}_{r'_d}[\![P']\!] \parallel l^{r_p}_{r_d}[\![Q']\!] \parallel N'}{K \triangleright l^{r_p}_{r_d}[\![P \mid Q]\!] \parallel N \;\longmapsto\; K' \triangleright l^{r'_p}_{r'_d}[\![P' \mid Q']\!] \parallel N'}$$

(Dπ-Par)

$$\frac{K \triangleright N_1 \;\longmapsto\; K' \triangleright N_1'}{K \triangleright N_1 \parallel N_2 \;\longmapsto\; K' \triangleright N_1' \parallel N_2}$$

(Dπ-Struct)

$$\frac{N_1 \equiv N_1' \qquad K \triangleright N_1' \;\longmapsto\; K' \triangleright N_2' \qquad N_2' \equiv N_2}{K \triangleright N_1 \;\longmapsto\; K' \triangleright N_2'}$$

Table 8
Dπ Operational Semantics

the net $l^{r_p}_{r_d}[\![a?(X).\mathbf{stop}]\!]$ is $\{l, a@l\} \cup r_p \cup r_d$. Like before, $\longmapsto^*$ denotes the reflexive and transitive closure of $\longmapsto$.

Substitutions are now generalised so that they map input parameters to messages and their application keeps into account also the structure of the message/parameter involved. In particular, in $P\{^W\!/X\}$, if W is a name e, then X must be a variable x and the application replaces x with e in P; otherwise, if W is a compound message $a@l$, then X must be a compound variable $z@y$ and the application replaces z with a and y with l in P. Like in cKlaim, substitution application also acts on the region annotations in P.

The reduction relation relies on a *structural congruence* relation, $\equiv$, equating α-convertible processes, stating that "$\|$" is commutative and associative, and that **stop** acts as the identity for "$|$".

We now comment on the Dπ peculiar operational rules; the others are similar to the corresponding ones of cKlaim. Notice that, differently from cKlaim, region annotations are not exploited to infer reductions, thanks to the powerful static typing. Rule (Dπ-Comm) states that the producer and the consumer of a datum must locally synchronise along a named channel a. Rule (Dπ-Go) moves the continuation process to the node target of the **go** ; notice that the static typing has already verified that k accepts data/code from l (i.e. $l \in r'_d$ or $l \in r'_p$, according to the fact that P is an output process or not). Rules (Dπ-NewLoc) and (Dπ-NewChan) handle name restriction. The first one creates a new node addressed by the fresh locality name k; k enlarges the creating node's regions that, similarly to cKlaim, are assigned to the new node too. The second rule allocates a new channel in the current locality. In both cases, the set K of names already in use is exploited to choose a fresh name. This corresponds to the intuition that, rather than declaring something as local, one can give it a syntactically different name. The effect is the same as in the standard semantics [19], where structural congruence is exploited to extend the scope of names by moving name restriction to the outermost level.

Before concluding, let us implement in Dπ the example given in Section 2. Like in cKlaim, the addresses are l_C and l_S with region annotations such that $l_S \in r_d^C$ and $l_C \in r_d^S$ (usually, $r_d^S = \top$), while $r_p^C = r_p^S = \emptyset$. Processes P_C and P_S now become

$$P_C \triangleq (\nu\texttt{pwd}).(\mathbf{go}\, l_S.\texttt{req}!\langle \texttt{pwd}@l_C, [creditCard\_info]_{\{l_C,l_S\}}\rangle \mid \\ \texttt{pwd}?(y). < modify\ password\ y\ and\ access\ the\ service >)$$

$$P_S \triangleq *\,\texttt{req}?(x_1@x'_1, x_2). < check\ credit\ card\ info\ x_2 > . \\ (\nu p)(\mathbf{go}\, x'_1.x_1!\langle [p]_{\{x'_1,l_S\}}\rangle \mid \\ < handle\ password\ modifications\ and\ provide\ the\ service >)$$

Channel `req` is used as the access point to the server for requiring the service, while `pwd` is a fresh channel used to transmit secret passwords between the client and the server. For the sake of presentation, we also assume basic values (of type *val*) for passwords and credit card information. Under these assumptions, a possible type environment to typecheck the net

$$l_C{}^{r_p^C}_{r_d^C}[\![P_C]\!] \ \| \ l_S{}^{r_p^S}_{r_d^S}[\![P_S]\!]$$

is

$$\begin{array}{l}\Gamma : l_C \mapsto \phi_C \\ \qquad l_S \mapsto \top \rhd \{l_S\}[\texttt{req} : \top(\gamma_{\text{pwd}}@\phi_C \times val)]^{r_p^S}_{r_d^S}\end{array}$$

where $\phi_C = \top \rhd \{l_C\}[\ ]^{r_p^C}_{r_d^C}$ and $\gamma_{\text{pwd}} = \{l_C, l_S\}(val)$. Clearly, we are not considering the activities of modifying/handling passwords, nor those of accessing/providing the service. A sketch of the type checking for P_C is given below. It requires to establish that

$$\Gamma \vdash_{l_C} (\nu\texttt{pwd}).(\textbf{go}\, l_S.\texttt{req}!\langle \texttt{pwd}@l_C, [creditCard\_info]_{\{l_C,l_S\}}\rangle \ \mid\ \texttt{pwd}?(y).\cdots)$$

By applying rules (D-T-LRes) and (D-T-Par), this requires to infer that

$$\Gamma_1 \vdash_{l_C} \textbf{go}\, l_S.\texttt{req}!\langle \texttt{pwd}@l_C, [creditCard\_info]_{\{l_C,l_S\}}\rangle$$

where $\Gamma_1 = \Gamma, {}_{l_C}\texttt{pwd} : \gamma_{\text{pwd}}$. The judgement above can be inferred by using rule (D-T-Go). To this aim, first we establish that $\{l_C\} \subseteq reg^{l_C}_{\Gamma_1}(\textbf{go}\, l_S.\texttt{req}!\langle \texttt{pwd}@l_C, [creditCard\_info]_{\{l_C,l_S\}}\rangle) = \{l_C, l_S\}$ and $\{l_C\} \subseteq r_d^S$, then we prove that

$$\Gamma_1 \vdash_{l_S} \texttt{req}!\langle \texttt{pwd}@l_C, [creditCard\_info]_{\{l_C,l_S\}}\rangle$$

This last judgement is inferred by using rule (D-T-Out). Indeed, the following conditions are all satisfied:

$$\begin{array}{lll}\Gamma_1 \vdash_{l_C} \texttt{req} : \top(\gamma_{\text{pwd}}@\phi_C \times val) & \{l_S\} \subseteq \top & \Gamma_1 \vdash_{l_C} \texttt{pwd}@l_C : \gamma_{\text{pwd}}@\phi_C \\ \Gamma_1 \vdash_{l_C} creditCard\_info : val & \Gamma_1 \vdash_{l_S} \textbf{stop} & \end{array}$$

Remark 1 (Typing Dπ Nets *à la* cKlaim) The typing approach used for cKlaim, where the values exchanged in communications are checked at runtime, can be easily adapted to Dπ. The resulting type system is simpler, e.g. types are simply regions and do not need complex subtyping relations. Of course, the modified setting requires more runtime checks. The definition of well-typed nets, like the type inference phase, can be straightforwardly adapted from Table 2. Clearly, the syntax of Table 5 must be adapted by letting parameters to be defined as

$$X \ ::= \ [x]^r \ \mid \ [z]^{r_1}@[y]^{r_2}$$

As regards the operational semantics, we need to replace rules (Dπ-Comm) and (Dπ-Go) in Table 8 with the following ones:

$$\frac{r' \subseteq r}{l^{r_p}_{r_d}[\![a?([x]^{r'}).P \mid a!\langle [e]_r \rangle.Q]\!] \;\longmapsto\; l^{r_p}_{r_d}[\![P\{^e\!/\!_x\} \mid Q]\!]}$$

$$\frac{r_1 \cup r_2 \subseteq r}{l^{r_p}_{r_d}[\![a?([z]^{r_1}@[y]^{r_2}).P \mid a!\langle [b@k]_r \rangle.Q]\!] \;\longmapsto\; l^{r_p}_{r_d}[\![P\{^{b@k}\!/\!_{z@y}\} \mid Q]\!]}$$

$$\frac{\text{if } \ \mathit{output}(P) \ \text{ then } \ l \in r'_d \ \text{ else } \ l \in r'_p}{l^{r_p}_{r_d}[\![\mathbf{go}\ k.P]\!] \parallel k^{r'_p}_{r'_d}[\![Q]\!] \;\longmapsto\; l^{r_p}_{r_d}[\![\mathbf{stop}]\!] \parallel k^{r'_p}_{r'_d}[\![P \mid Q]\!]}$$

4.3 Type Soundness

We now prove subject reduction and type safety for the type system of Table 6. The type soundness will be an easy corollary of these properties. We start with the corresponding versions for Dπ of Lemmas 1 and 2; then we state and prove subject reduction and type safety. To this aim, we define Γ **with** K **for** N as the typing environment Γ extended with the fresh names in K that can type N. Formally,

$$\Gamma \ \mathbf{with}\ K\ \mathbf{for}\ N \triangleq \begin{cases} \Gamma & \text{if } K = \emptyset \\ (\Gamma\ \mathbf{with}\ K'\ \mathbf{for}\ N),\ {}_l a : \gamma & \text{if } K = K' \uplus \{a@l\} \\ (\Gamma\ \mathbf{with}\ K'\ \mathbf{for}\ N) \uplus k : \phi & \text{if } K = K' \uplus \{k\} \end{cases}$$

where we let $\uplus$ to denote both the disjoint union of sets and the union of functions with disjoint domains. The γ and the ϕ in the second and third case above are the minimal types (w.r.t. $\sqsubseteq$) such that $(\Gamma$ **with** K **for** $N) \vdash N$.

Remark 2 Notice that it is not always the case that Γ **with** K **for** N is defined; however, if it is defined then N is well-typed.

Lemma 3 (Subject Congruence) *If $\Gamma \vdash N$ and $N \equiv N'$ then $\Gamma \vdash N'$.*

Lemma 4 (Substitutivity) *Let W be such that $\mathrm{FV}(W) = \emptyset$. Then, the following facts hold:*

(1) If $\Gamma,\ {}_vX : \tau \vdash_u W' : \tau''$ and $\Gamma \vdash_v W : \tau'$, for some $\tau' \sqsupseteq \tau$, then $\Gamma \vdash_{u\{W/X\}} W'\{^W\!/\!_X\} : \tau''$.

(2) If $\Gamma,\ {}_vX : \tau \vdash_u P$ and $\Gamma \vdash_v W : \tau'$, for some $\tau' \sqsupseteq \tau$, then $\Gamma \vdash_{u\{W/X\}} P\{^W\!/\!_X\}$.

Proof: The proof of the first claim is similar to the corresponding one in [19], once the subtyping relation used is that defined in Table 7. To prove the second claim, we distinguish three cases (the last one relies on the first two) according to the structure of parameter X: X can be u (and clearly u is a variable), or X can be $x \neq u$, or X can be $y@z$. We now proceed by induction on judgement $\Gamma,\, {}_vX : \tau \vdash_u P$ and considers only the first two cases above; the third one is recovered by considering $\{^{a@l}/_{y@z}\}$ as the composition of the two substitutions $\{^a/_y\}$ and $\{^l/_z\}$.

The base case is trivial. For the inductive cases, the proof is tedious. We just show the most delicate case, i.e. when (D-T-Out) is the last rule applied; for notational convenience, we let $\Gamma' = \Gamma,\, {}_vX : \tau$. By hypothesis, $P = v_1!\langle[W']_r\rangle.P'$; thus, $\Gamma' \vdash_u v_1 : r'(\tau_1)$, $val(\Gamma'(u)) \subseteq r'$, $\Gamma' \vdash_u W' : \tau_2$ for some $\tau_2 \sqsupseteq \tau_1$, $u \in r$, $\bigcup_{w\in r} val(\Gamma(w)) \subseteq reg(\tau_1)$ and $\Gamma' \vdash_u P'$. Clearly, $u\{^W/_X\} \in r\{^W/_X\}$; moreover, by claim (1) of this Lemma and by induction, we have that $\Gamma \vdash_{u\{W/X\}} v_1\{^W/_X\} : r'(\tau_1)$, $\Gamma \vdash_{u\{W/X\}} W'\{^W/_X\} : \tau_2$ and $\Gamma \vdash_{u\{W/X\}} P'\{^W/_X\}$. We still have to prove that $val(\Gamma(u\{^W/_X\})) \subseteq r'$ and $\bigcup_{w\in\, r\{W/X\}} val(\Gamma(w)) \subseteq reg(\tau_1)$. By hypothesis, $\tau \sqsubseteq \tau'$; thus, it holds that $val(\Gamma(W)) \subseteq val(\Gamma(X))$. Hence, we can state that $val(\Gamma(u\{^W/_X\})) \subseteq val(\Gamma'(u)) \subseteq r'$ and $\bigcup_{w\in\, r\{W/X\}} val(\Gamma(w)) \subseteq \bigcup_{w\in r} val(\Gamma(w)) \subseteq reg(\tau_1)$ (it suffices to distinguish if $X = u$ or not). Thus, $\Gamma \vdash_{u\{W/X\}} (v_1\{^W/_X\})!\langle[W'\{^W/_X\}]_{r\{W/X\}}\rangle.P'\{^W/_X\}$, as required. □

Theorem 4 (Subject Reduction) *If N is well-typed and $K \triangleright N \longmapsto K' \triangleright N'$ then N' is well-typed.*

Proof: By definition, there exists a typing environment Γ such that $\Gamma \vdash N$. Thus, because of Remark 2, it suffices to prove that Γ **with** $(K' - K)$ **for** N' is defined. Similarly to Theorem 1, we proceed by induction on the length of the inference leading to $K \triangleright N \longmapsto K' \triangleright N'$ and we do not consider K and K' anymore.

Base Step: We reason by case analysis on the axioms (i.e. the first five rules) of Table 8.

(Dπ-Comm). In this case, $K' - K = \emptyset$; thus, Γ **with** $(K' - K)$ **for** $N' = \Gamma$ if and only if $\Gamma \vdash l^{r_p}_{r_d}[\![P\{^W/_X\} \mid Q]\!]$. By hypothesis, $\Gamma \vdash_l a?(X).P \mid a!\langle[W]_r\rangle.Q$; thus, by rule (D-T-Par), $\Gamma \vdash_l a?(X).P$ and $\Gamma \vdash_l a!\langle[W]_r\rangle.Q$. By rules (D-T-In) and (D-T-Out), we know that $\Gamma \vdash_l a : r'(\tau)$, $\Gamma \vdash_l W : \tau'$ for some $\tau' \sqsupseteq \tau$; moreover $\Gamma,\, {}_lX : \tau \vdash_l P$ and $\Gamma \vdash_l Q$. Since we assumed that Γ does not contain variables (otherwise $\Gamma \not\vdash N$), we can state that W is a closed message. Thus, we can apply Lemma 4 and obtain that $\Gamma \vdash_l P\{^W/_X\}$. This suffices to conclude.

(Dπ-Go). This case is simpler.

(Dπ-NewLoc). In this case, $K' - K = \{k\}$; thus, Γ **with** $(K' - K)$ **for** N' is defined if and only if $\Gamma,\, {}_lk : \phi \vdash l^{r_p\cup\{k\}}_{r_d\cup\{k\}}[\![P]\!] \parallel k^{r_p\cup\{k\}}_{r_d\cup\{k\}}[\![\mathbf{stop}]\!]$ for some locality type ϕ. By hypothesis, $\Gamma \vdash l_{r_d}^{r_p}[\![(\nu k)P]\!]$; thus, by rule (D-T-LRES), it holds that $\Gamma, {}_l k : \top \triangleright \{k\}[\emptyset]_{r_p \cup \{k\}}^{r_d \cup \{k\}} \vdash_l P$. We can now easily conclude.

(Dπ-NEWCHAN). This case is similar.

(Dπ-CALL). This case proceeds like in Theorem 1.

Inductive Step: We reason by case analysis on the last applied operational rule, i.e. (Dπ-SPLIT), (Dπ-PAR) or (Dπ-STRUCT). These cases are similar to Theorem 1 and, thus, are omitted. □

We now consider type safety. We could proceed like for cKLAIM, by exploiting a tagged language. However, in the Dπ setting, we can formulate and prove the safety property in a simpler (but coarser) way. The intuition is that a Dπ typing environment already associates a region to each (free) name of a net. Thus, we can define a notion of safety *w.r.t. a typing environment* in the following way:

Definition 4 (Safety) *A net N is* Γ-safe *if for any $l_{r_p}^{r_d}[\![P]\!]$ in N, it holds that $l \in reg_\Gamma^l(P)$.*

This definition is somehow "less accurate" than Definition 2 in that all the occurrences of the same name are now associated with the same tag. To obtain the finer property, we should tune the theory presented in Section 3.3; we omit the details and go on working with Definition 4.

Theorem 5 (Type Safety) *If $\Gamma \vdash N$ then N is Γ-safe.*

Proof: The proof proceeds by induction on the length of the inference of judgement $\Gamma \vdash N$. The proof relies on the check $val(\Gamma(u)) \subseteq reg_\Gamma^u(\mathbf{go}\ v.P)$ in rule (D-T-GO), the check $val(\Gamma(u)) \subseteq r$ in (D-T-IN) and $val(\Gamma(u)) \subseteq r'$ in (D-T-OUT). □

Type soundness now easily follows (the definition of r-subnet and the proof of the claim are similar to the corresponding ones in Section 3.3). Notice that type soundness can be recovered as an instance of *localised* type soundness.

Corollary 3 (Localised Type Soundness) *Let the r-subnet of N be defined and well-typed in Γ. If $K \triangleright N \longmapsto^* K' \triangleright N'$ then the r'-subnet of N' is defined and Γ'-safe, where $r' = r \cup (K' - K)$ and $\Gamma' = \Gamma$ **with** $(K' - K)$ **for** N'.*

5 Mobile Ambients Calculus

Finally, we apply our approach to Mobile Ambients Calculus [5] (in the following, we will shorten it as Ambients). The calculus relies on the notion of

$n, m, p, \ldots \in \mathcal{A}$			Ambients names
$x, y, z, \ldots$			Variables
u	$::=$	$n \mid x$	Identifiers
P	$::=$		Processes
		$\mathbf{0}$	(empty)
	$\mid$	$\langle [u]_r \rangle$	(datum)
	$\mid$	$\alpha.P$	(prefixing)
	$\mid$	$P_1 \mid P_2$	(parallel composition)
	$\mid$	(νn)	(name restriction)
	$\mid$	$*P$	(replication)
	$\mid$	$u[\![P]\!]$	(ambient)
α	$::=$		Actions
		(x)	(receive)
	$\mid$	$\mathbf{in}\_u$	(enter u)
	$\mid$	$\mathbf{out}\_u$	(exit u)
	$\mid$	$\mathbf{open}\_u$	(open u)

Table 9
Ambients Syntax

ambient that can be thought of as a bounded place where processes cooperate. This notion is similar to that of node, but differently from cKlaim and Dπ nodes, ambients can be hierarchically structured and can be moved as a whole under the control of processes.

The syntax of the calculus is given in Table 9. There is only one category of names, namely that of *ambient names* $\mathcal{A}$, ranged over by n. Identifiers, ranged over by u, v, w, denote ambient names and variables (ranged over by x), and represent both the target of process actions and the data exchanged during communication. Communication is asynchronous and anonymous (no place or communication channel is explicitly referred), and takes place locally within a single ambient.

In Ambients, everything is a process, namely, differently from cKlaim and Dπ, there is no distinction among processes, nodes and nets. Other than the standard process operators, i.e. empty process, prefixing, parallel composition, name restriction and replication, we have $\langle [u]_r \rangle$, that represents message u tagged with region r within the current ambient, and $n[\![P]\!]$, that represents an ambient with name n and process P running inside. An ambient, hence, has a name, a collection of local processes and a collection of subambients. Notice that nothing prevents existence of two or more ambients with the same name, possibly enclosed within the same ambient. Process actions are:

- (x): receives a message u within the current ambient and replaces x with u in the continuation;
- $\mathbf{in}\_u$: moves the ambient enclosing the process executing the action in a sibling ambient whose name is u;
- $\mathbf{out}\_u$: moves the ambient enclosing the process executing the action out of its enclosing ambient provided that this is named u;
- $\mathbf{open}\_u$: dissolves the boundary of an ambient named u and unleashes u's content.

Identifiers occurring in processes can be *bound*. More precisely, prefix $(x).P$ binds variable x, while $(\nu n)P$ binds name n; in both cases, P is the scope of the binding. The set of free variables $\mathrm{FV}(\_)$, α-conversion and closed nets are defined accordingly.

Differently from the calculi previously presented, in Ambients there is no need to associate regions r_d/r_p to ambients. Indeed, as processes are confined within ambients, new data/code can enter an ambient n only because an ambient boundary is dissolved by an action **open** executed within n. Since this action is under the control of n, no (static nor dynamic) check is needed to prevent the *unwanted* arrival of undesired data/code. At most, some control can be carried on the ambients that n can open; but this is an orthogonal task.

5.1 Typing Ambients Nets

We adopt a static typing approach, like for Dπ. For the sake of presentation, here we only illustrate the key features of the application of our approach to Ambients and relegate all technical details to Appendix A. Ambients types are defined as follows:

$$T ::= \mathtt{Shh} \quad | \quad r_1 \triangleright r_2 \triangleright r_3[T]$$

Intuitively, an ambient u has type $r_1 \triangleright r_2 \triangleright r_3[T]$ if its name is in r_3 (this is useful only if u is a variable), it can be seen by all ambients in r_1 and enclosed within all ambients in r_2. Moreover, the ambient hosts processes exchanging data of type T, the *local conversation topic*. Topics of conversations were introduced in [4]; here we use them in a similar way and denote with Shh the absence of exchanges in the ambient. Moreover, we always assume that types are well-formed, i.e., for all $r_1 \triangleright r_2 \triangleright r_3[\cdot]$, it holds that $r_2 \subseteq r_1$. Finally, we let $\mathit{cont}(r_1 \triangleright r_2 \triangleright r_3[\cdot]) = r_2$ and $\mathit{look}(r_1 \triangleright r_2 \triangleright r_3[\cdot]) = r_1$.

The main judgement is $\Gamma \vdash P$ and states that P is well-typed under the assumptions Γ. A *type environment* Γ is a finite partial function mapping ambient names and variables to types (a well-formedness condition similar to (‡) in Section 4.1 is assumed, see Appendix A). The key requirement to typecheck Ambients processes is that

whenever ambient n is contained in ambient m (i.e., $m[\![n[\![\cdots]\!] \mid \cdots]\!]$), it must hold that $\mathit{cont}(\Gamma(m)) \subseteq \mathit{cont}(\Gamma(n))$ (§)

By construction, we have that $\mathit{cont}(\Gamma(n)) \subseteq \mathit{look}(\Gamma(n))$. These conditions together ensure that, if m is opened while still containing n, n can be seen by the ambient enclosing m. For example, consider the following process:

$$k[\![\ m[\![n[\![\langle [d]_r \rangle \mid \cdots]\!] \mid \mathbf{open\_}n]\!] \ \mid\ \mathbf{open\_}m\ ]\!]$$

If the process is well-typed, we know that $k \in \mathit{cont}(\Gamma(m)) \subseteq \{m\} \cup \mathit{cont}(\Gamma(m)) \subseteq \mathit{cont}(\Gamma(n)) \subseteq r$ (these inclusions follow by the premises of the typing rules). This means that both k and m can see datum d; hence, the execution of actions $\mathbf{open\_}n$ and $\mathbf{open\_}m$ does not break well-typedness, as intended.

Well-typedness for Ambients processes is defined as follows.

Definition 5 (Well-Typed Ambients Processes) *A process P is* well-typed in Γ *if $\Gamma \vdash P$. P is* well-typed *if there exists a typing environment Γ such that P is well-typed in Γ.*

Remark 3 To conclude this section, we want to remark that most of the intricacies in the setting of the Ambients calculus (especially, the need for requirement (§) above) are related to the presence of action **open**. Indeed, other calculi, that have been derived from Ambients by removing such a primitive, can be typed very similarly to Dπ. As a first example, we consider M^3 [6] where action **open** is replaced by a primitive for process migration, **to**, which is in the same vein of cKlaim's **eval** and Dπ's **go** . In this setting, types look like Dπ's locality types and are defined as $r \triangleright r'[T]^{r_p}_{r_d}$ (the meaning of r, r', r_d and r_p is like in Dπ, while T is the topic of conversation). As another example, we consider Boxed Ambients [2] where action **open** is replaced with primitives for (non local) parent/child communication. Types still take the form $r \triangleright r'[T]^{r_p}_{r_d}$, but more checks are needed in the typing phase to ascertain that data are exchanged correctly. Indeed, the ability of performing (limited forms of) remote communications introduces new possibilities to forge data regions.

(A-In)

$$\frac{}{n[\mathbf{in\_}m.P \mid Q] \mid m[R] \longrightarrow m[n[P|Q] \mid R]}$$

(A-Call)

$$\frac{}{*P \longrightarrow *P \mid P}$$

(A-Out)

$$\frac{}{m[n[\mathbf{out\_}m.P \mid Q] \mid R] \longrightarrow n[P|Q] \mid m[R]}$$

(A-Res)

$$\frac{}{A \triangleright (\nu n)P \longrightarrow A \uplus \{n\} \triangleright P}$$

(A-Open)

$$\frac{}{\mathbf{open\_}m.P \mid m[R] \longrightarrow P \mid R}$$

(A-Par)

$$\frac{A \triangleright P_1 \longrightarrow A' \triangleright P_1'}{A \triangleright P_1|P_2 \longrightarrow A' \triangleright P_1'|P_2}$$

(A-Comm)

$$\frac{}{\langle [m]_r \rangle \mid (x).Q \longrightarrow Q\{m/x\}}$$

(A-Amb)

$$\frac{A \triangleright P \longrightarrow A' \triangleright P'}{A \triangleright n[P] \longrightarrow A' \triangleright n[P']}$$

(A-Struct)

$$\frac{P_1 \equiv P_1' \qquad A \triangleright P_1' \longrightarrow A' \triangleright P_2' \qquad P_2' \equiv P_2}{A \triangleright P_1 \longrightarrow A' \triangleright P_2}$$

Table 10
Ambients Operational Semantics

5.2 Ambients Operational Semantics

Ambients processes are executed according to the reduction relation $\longrightarrow$ defined in Table 10. Like for the previous calculi, $\longrightarrow$ relates configurations of the form $A \triangleright P$, where A is now a set of ambient names such that $n(P) \subseteq A \subset_{fin} \mathcal{A}$. Function $n(P)$ returns the set of ambient names occurring in P. As usual, $\longrightarrow^*$ stands for the reflexive and transitive closure of $\longrightarrow$.

The semantics is given modulo a *structural congruence* relation, $\equiv$, equating α-convertible processes and stating that "$|$" is commutative, associative and with $\mathbf{0}$ as identity.

We now comment on the Ambients peculiar operational rules. Rule (A-In) says that the ambient n performing the action enters the sibling ambient m. If no sibling m can be found, the operation gets stuck until such a sibling exists; if more than one sibling m exists, any one of them can be chosen. Symmetrically, rule (A-Out) says that the ambient n performing the action exits its enclosing ambient if this is named m; otherwise, the action gets stuck. Rule (A-Open) says that the boundary of ambient n is dissolved and n's content is unleashed, possibly within the ambient performing the action. If no ambient

n is found, the operation gets stuck until such an ambient exists; if more than one ambient n exists, any one of them can be chosen. Rule (A-Comm) accounts for asynchronous communication between co-located processes; again, the static typing enables the communication without any runtime overhead. In rule (A-Res), the set A of names already in use is exploited to choose a fresh name n. Finally, rule (A-Amb) states that, if the content of an ambient evolves, then the whole ambient evolves accordingly.

Before concluding, let us implement in Ambients the example presented in Section 2. The server and the client are modelled as two sibling ambients whose names are n_S and n_C. Processes P_C and P_S now become

$$
\begin{aligned}
P_C \triangleq\ & (\nu \texttt{pwd})(\texttt{req}[\![\mathbf{out}\_n_C.\mathbf{in}\_n_S.\langle n_C, \texttt{pwd}, [cc\_info]_{\{n_C,n_S,\texttt{req}\}}\rangle]\!] \\
& \qquad \mid \mathbf{open}\_\texttt{pwd}.(y).<\mathit{modify\ pwd\ y\ and\ access\ the\ service}>) \\
P_S \triangleq\ & *\,\mathbf{open}\_\texttt{req}.(x_1,x_2,x_3).<\mathit{check\ credit\ card\ info}\ x_3>. \\
& (\nu p)(x_2[\![\mathbf{out}\_n_S.\mathbf{in}\_x_1.\langle [p]_{\{x_1,x_2,n_S\}}\rangle]\!] \mid \\
& \qquad <\mathit{handle\ password\ modifications\ and\ provide\ the\ service}>)
\end{aligned}
$$

Ambient `req` is used as an access point to the server; indeed, it brings message $\langle n_C, \texttt{pwd}, [cc\_info]_{\{n_C,n_S,\texttt{req}\}}\rangle$ out of the client and then in the server, where it is dissolved, thus enabling the reception of the message. This is carried on by the following reductions (where, like before, the sets A of names in use are omitted):

$$
\begin{aligned}
& n_C[\![P_C]\!] \mid n_S[\![P_S]\!] \\
& \longrightarrow^* n_C[\![\mathbf{open}\_\texttt{pwd}.\cdots]\!] \mid n_S[\![P_S]\!] \mid \texttt{req}[\![\mathbf{in}\_n_S.\langle n_C, \texttt{pwd}, [cc\_info]_{\{n_C,n_S,\texttt{req}\}}\rangle]\!] \\
& \longrightarrow n_C[\![\mathbf{open}\_\texttt{pwd}.\cdots]\!] \mid n_S[\![P_S \mid \texttt{req}[\![\langle n_C, \texttt{pwd}, [cc\_info]_{\{n_C,n_S,\texttt{req}\}}\rangle]\!]]\!] \\
& \longrightarrow n_C[\![\mathbf{open}\_\texttt{pwd}.\cdots]\!] \mid n_S[\![P_S \mid \langle n_C, \texttt{pwd}, [cc\_info]_{\{n_C,n_S,\texttt{req}\}}\rangle \mid (x_1,x_2,x_3).\cdots]\!]
\end{aligned}
$$

Upon verification of the credit card information, the server creates a new password p that is delivered back to n_C by the ambient `pwd`. Again, this last ambient serves as an access point to the client and acts like ambient `req` above.

By comparing this implementation with that in Dπ, one can notice how channels can be implemented in Ambients. Intuitively, channels (e.g., `req` and `pwd` in the example above) are rendered as pilot ambients that bring messages from the sender to the receiver, by following possibly complex routing paths. Once they have reached their final destination, such ambients are opened so that the messages they carry on are unleashed and can be retrieved by the receiver.

6 A Realistic Example: Implementing a Multiuser System

In this section we want to further illustrate our approach. To this aim, we use the framework presented so far to program a simple but meaningful example in cKlaim; the implementations in the other calculi can be derived straightforwardly. For the sake of readability, we will use parameterised process definitions and strings. Moreover, we borrow from [13] polyadic communication, i.e. the possibility of exchanging tuples of data, and the primitive **read** that behaves similarly to **in** but, after its execution, it leaves the accessed data in the TS. Clearly, the type inference for actions **read** works similarly to that for actions **in** (by adding region annotations to parameters occurring in templates).

We present the behaviour of a simple UNIX-like multiuser system, where users can login (exploiting a password-based approach) and use the system functionalities, which consist in reading/writing files or executing programs. For the sake of presentation, we shall present the system in three steps and, finally, we shall merge them together. Let l_S be the address of the server, $\top$ be its data trust region and $\emptyset$ be its process trust region (thus no user can spawn code to l_S).

User Identification. We start with programming the identification of different users via passwords. Localities play the role of user IDs. Let l_p be a private repository used by l_S to record the registered users and their passwords. Thus, l_p hosts the tuples

$$\langle l_1, [pwd_1]_{\{l_1,l_p,l_S\}}\rangle \quad | \quad \ldots \quad | \quad \langle l_n, [pwd_n]_{\{l_n,l_p,l_S\}}\rangle$$

Let l be a user wanting to log in l_S. If l is already known to l_S (i.e. it is one of the l_is), then l can use a process like

$$\mathbf{out}(\text{“}login\text{”}, l, [pwd]_{\{l,l_S\}})@l_S.\mathbf{in}(\text{“}logged\text{”})@l_S. \ldots$$

for communicating with the server process

$$Login(l_p) \triangleq * \mathbf{in}(\text{“}login\text{”}, !u, !z)@l_S.\mathbf{read}(u,z)@l_p.\mathbf{out}([\text{“}logged\text{”}]_{\{l_S,u\}})@l_S$$

Intuitively, l requires a connection by sending its user ID (its locality) and its password; the server checks if this information is correct and sends back an ack, activating the continuation of the computation at l. Notice that the region annotations of pwd and “*logged*” rule out attacks of a nasty intruder aimed at cancelling the request of login or the corresponding ack, and preserve the secrecy of the password.

If the user is not registered at l_S yet, he can send an "hello" request to the server containing its address and wait for a password

$$\mathbf{out}([\text{``}hello\text{''}]_{\{l,l_S\}}, l)@l_S.\mathbf{in}(\text{``}registered\text{''}, !pwd)@l_S. \ldots$$

The server then handles this request with the process

$$\begin{aligned} NewUser(l_p) \quad &\triangleq * \mathbf{in}(\text{``}hello\text{''}, !u)@l_S.\mathbf{newloc}(pwd).\mathbf{out}(u, [pwd]_{\{u,l_S,l_p\}})@l_p. \\ &\quad \mathbf{out}(\text{``}registered\text{''}, [pwd]_{\{l_S,u\}})@l_S \end{aligned}$$

Of course, a locality l' different from l can send l_S a request for a new password pretending to be l: the only difference with the "hello" message given above is that the message now should contain also l' in the data region. However, the server will report the new password to l and the region associated to the password will ensure that pwd will not leave l. Thus, l' can withdraw pwd only by sending a process to l and then acting at l with the new password. This can be possible only if l trusts l', implying that l accepts this 'suspicious' activity of l'.

We now show the use of our typing theory in the setting just presented. In particular, we give evidence on how we can prevent attacks aimed at cancelling messages and the activity of malicious users pretending to play the role of other users.

- Let l_{canc} be a locality hosting a process that aims at interfering with the login procedures by performing action $\mathbf{in}(\text{``}hello\text{''}, !x)@l_S$. In this way, it removes the *hello* message sent by a new user l wanting to be connected with the server l_S. The system is modelled as follows

$$\begin{aligned} & l_{\text{canc}} :: \mathbf{in}(\text{``}hello\text{''}, !x)@l_S.DONE \parallel l_S :: NewUser(l_p) \\ & \parallel l :: \mathbf{out}([\text{``}hello\text{''}]_{\{l,l_S\}}, l)@l_S.\mathbf{in}(\text{``}registered\text{''}, !pwd)@l_S. \cdots \\ & \quad \rightarrowtail \quad l_{\text{canc}} :: \mathbf{in}(\text{``}hello\text{''}, !x)@l_S.DONE \\ & \qquad\qquad \parallel l :: \mathbf{in}(\text{``}registered\text{''}, !pwd)@l_S. \cdots \\ & \qquad\qquad \parallel l_S :: NewUser(l_p) \mid \langle [\text{``}hello\text{''}]_{\{l,l_S\}}, l \rangle \\ & \quad \not\rightarrowtail \quad l_{\text{canc}} :: DONE \parallel l :: \mathbf{in}(\text{``}registered\text{''}, !pwd)@l_S. \cdots \\ & \qquad\qquad \parallel l_S :: NewUser(l_p) \end{aligned}$$

  Notice that the last transition cannot take place. As intended, the intruder running at l_{canc} is not enabled to withdraw the tuple $\langle [\text{``}hello\text{''}]_{\{l,l_S\}}, l \rangle$ because $l_{\text{canc}} \notin \{l, l_S\}$ (see the runtime check of rule (cK-Match)).
- Let now l_{pret} be a locality pretending to act on behalf of l, by trying to acquire a log to l_S under the identity of l. Let us examine the possible

evolutions of the system:

$$
\begin{array}{l}
l_{\text{pret}} :: \mathbf{out}(\text{``}\mathit{hello}\text{''}, l)@l_S.\mathbf{in}(\text{``}\mathit{registered}\text{''}, !pwd)@l_S.DONE \\
\parallel l_S :: NewUser(l_p) \\
\quad \rightarrowtail\ \rightarrowtail\ \rightarrowtail\ \rightarrowtail\ l_{\text{pret}} :: \mathbf{in}(\text{``}\mathit{registered}\text{''}, !pwd)@l_S.DONE \\
\qquad\qquad\qquad\qquad \parallel l_S :: NewUser(l_p) | \langle \text{``}\mathit{registered}\text{''}, [pwd]_{\{l_S, l\}} \rangle \\
\quad \not\rightarrowtail\ l_{\text{pret}} :: DONE \parallel l_S :: NewUser(l_p)
\end{array}
$$

Again, the last reduction cannot take place because $l_{\text{pret}} \notin \{l_S, l\}$. The only way for l_{pret} to withdraw datum $\langle \text{``}\mathit{registered}\text{''}, [pwd]_{\{l_S, l\}} \rangle$ is to spawn a process to $l$ (if it exists in the net) executing action $\mathbf{in}(\text{``}\mathit{registered}\text{''}, !pwd)@l_S$ (that would be enabled, because $l \in \{l_S, l\}$). Such a migration, however, must be authorised by l (indeed, it can take place only if $l_{\text{pret}} \in r_p^l$, where r_p^l is the node region controlling migrations to l).

The File System. We now consider a server handling a file system where different users can write/read data. Let l_f be a private repository used by l_S to store the files. A file named N, whose content is the string S, that can be read by users in r and written by users in r', is stored in l_f as the process

$$C_N \triangleq \langle N, [\text{``}\mathit{read}\text{''}]_{r \cup \{l_S, l_f\}}, [\text{``}\mathit{written}\text{''}]_{r' \cup \{l_S, l_f\}} \rangle \quad | \quad \langle N, S \rangle$$

Intuitively, "*read*" and "*written*" are just dummy data used to properly store regions r and r'. Then, the server handles requests for reading and writing files with the following processes

$$
\begin{array}{rl}
Read(l_f) & \triangleq * \mathbf{in}(\text{``}\mathit{read}\text{''}, !u, !n)@l_S.\mathbf{read}(n, !z_r, !z_w)@l_f.\mathbf{read}(n, !z)@l_f. \\
& \quad \mathbf{out}([z_r]_{\{l_f, l_S, u\}}, n, z)@u \\
Write(l_f) & \triangleq * \mathbf{in}(\text{``}\mathit{write}\text{''}, !u, !n, !z)@l_S.\mathbf{read}(n, !z_r, !z_w)@l_f.\mathbf{in}(n, !z')@l_f. \\
& \quad \mathbf{out}(n, z)@l_f.\mathbf{out}([z_w]_{\{u, l_f, l_S\}}, n)@u
\end{array}
$$

Intuitively, the first **in** action collects the request for reading/writing the file named n performed by locality u; then the following **read** action, once type checked, verifies if the locality replacing u has the read/write privilege on file n (see below). Finally, the required operation is performed (the content of the file is read or the old content is replaced with the new one) and an acknowledgement (containing the kind of operation performed, the name of the file and, in the "read" case, also its content) is reported to u.

We now show how our types can control read accesses to files. There are two features devoted to this aim: the type inference phase carried on process $Read(l_f)$ and the runtime checks of the operational semantics (in particular, the premise of rule (cK-In)). We first give the type inference; recall that absence of region annotations stands for $\top$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\{l_S, u\} \subseteq s \qquad\qquad \Gamma_1 \vdash \mathbf{nil} \ \succ_{l_S} \ \Gamma_1 \vdash \mathbf{nil}}{\Gamma_1 \vdash \mathbf{out}\ ([z_r]_s, n, z)@u \ \ \succ_{l_S} \ \ \Gamma_2 \vdash \mathbf{out}\ ([z_r]_s, n, z)@u}}{\begin{array}{c}\Gamma_3 \vdash \mathbf{read}\ (n, !z)@l_f.\mathbf{out}([z_r]_s, n, z)@u \quad \succ_{l_S} \\ \Gamma_4 \vdash \mathbf{read}\ (n, !z)@l_f.\mathbf{out}([z_r]_s, n, z)@u\end{array}}}{\begin{array}{c}\Gamma_5 \vdash \mathbf{read}\ (n, !z_r, !z_w)@l_f.\mathbf{read}(n, !z)@l_f.\mathbf{out}([z_r]_s, n, z)@u \quad \succ_{l_S} \\ \Gamma_6 \vdash \mathbf{read}\ (n, [!z_r]^s, [!z_w]^{\{l_S\}})@l_f.\mathbf{read}(n, !z)@l_f.\mathbf{out}([z_r]_s, n, z)@u\end{array}}}{\begin{array}{l}\emptyset \vdash \mathbf{in}\ (\text{“}read\text{”}, !u, !n)@l_S.\mathbf{read}(n, !z_r, !z_w)@l_f. \\ \qquad \mathbf{read}(n, !z)@l_f.\mathbf{out}([z_r]_s, n, z)@u \\ \succ_{l_S} \ \ \emptyset \vdash \mathbf{in}\ (\text{“}read\text{”}, [!u]^{\{l_s\}}, !n)@l_S.\mathbf{read}(n, [!z_r]^s, [!z_w]^{\{l_S\}})@l_f. \\ \qquad\qquad \mathbf{read}(n, !z)@l_f.\mathbf{out}([z_r]_s, n, z)@u\end{array}}$$

where we let $s \triangleq \{l_f, l_S, u\}$ and

$$\begin{array}{ll}\Gamma_1 \triangleq \Gamma_3 \uplus \{z : \{l_S\}\} & \Gamma_2 \triangleq \Gamma_1 + \{z_r : s, n : \top, z : \top\} \\ \Gamma_3 \triangleq \Gamma_5 \uplus \{z_r : \{l_S\}, z_w : \{l_S\}\} & \Gamma_4 \triangleq \{u : \{l_S\}, n : \top, z_r : s, z_w : \{l_S\}\} \\ \Gamma_5 \triangleq u : \{l_S\}, n : \{l_S\} & \Gamma_6 \triangleq u : \{l_S\}, n : \top\end{array}$$

We now call $TRead(l_f)$ the process obtained from the typing inference above. Let l be a user wanting to read a file named $FILE$ associated to a read region $\rho = \{l_f, l_S, l, \ldots\}$. $FILE$ is then stored at l_f as the process

$$C_{FILE} \ \triangleq \ \langle FILE\,,\, [\text{“}read\text{”}]_\rho\,,\, [\text{“}written\text{”}]_{\rho'}\rangle \ \ | \ \ \langle FILE, content\rangle$$

The evolution of user l is

$$\begin{array}{l} l :: \mathbf{out}(\text{“}read\text{”}, l, FILE)@l_S.\mathbf{in}(\text{“}read\text{”}, FILE, !cont)@l.P \\ \parallel l_f :: C_{FILE} \parallel l_S :: TRead(l_f) \\ \quad \rightarrowtail \quad \rightarrowtail \quad l :: \mathbf{in}(\text{“}read\text{”}, FILE, !cont)@l.P \parallel l_f :: C_{FILE} \\ \qquad\qquad\qquad \parallel l_S :: TRead(l_f) \mid \mathbf{read}(FILE, [!z_r]^{\{l_f, l_S, l\}}, [!z_w]^{\{l_S\}})@l_f.\cdots \end{array}$$

Now, action $\mathbf{read}(FILE, [!z_r]^{\{l_f, l_S, l\}}, [!z_w]^{\{l_S\}})@l_f$ is enabled, because $\{l_f, l_S, l\} \subseteq \rho$; thus, the content of $FILE$ will be transferred to l that, in turn, will be enabled to retrieve it (by binding *content* to the variable *cont*) and use it in P. Notice that, if a user $l' \notin \rho$ had tried to carry on the same task, these actions would not have been enabled, since $\{l_f, l_S, l'\} \not\subseteq \rho$.

Executing Code-on-Demand. In this last setting, a user can dynamically download some code from the server to perform a given task. The server stores all the downloadable processes as executable (named) files in a private locality l_c. For each executable file named N, whose code is P and that is downloadable by nodes in r, the server stores in l_c the component

$$\begin{aligned} C_N \ \triangleq\ & \langle N, [\text{“}\mathit{downloaded}\text{”}]_{r\cup\{l_S,l_c\}}\rangle \quad \mid \\ & *\mathbf{in}(req, N, !u)@l_c.\mathbf{read}(N, !z_e)@l_c. \\ & \ \mathbf{eval}(\ \mathbf{eval}(\mathbf{out}([z_e]_{\{l_c,l_S,u\}}, N)@u.P)@u\ )@l_S \end{aligned}$$

Then, when a user wants to download some code, the server handles its request with the process

$$\mathit{Execute}(l_c)\ \triangleq\ *\mathbf{in}(\text{“}\mathit{execute}\text{”}, !u, !n)@l_S.\mathbf{out}(req, n, u)@l_c$$

Notice that l_c cannot directly send P for execution to u because (the locality associated to) u cannot have l_c in its trust region (since l_c is fresh). Thus, P must firstly cross l_S and then, if l_S is in the process trust region of u (which we assume it is the case), the code-on-demand procedure successfully terminates, by also reporting an ack to the user.

The System. Finally, we can put together the activities shown so far to obtain the implementation of the whole server. Thus, the (not yet typed) initial configuration of l_S would be

$$\begin{aligned} l_S\ \ {}_{\top}\!::_{\emptyset}\ & \mathbf{newloc}(u_1).\mathbf{newloc}(u_2).\mathbf{newloc}(u_3)\,. \\ & < set\ up\ \ u_1\ with\ the\ identites\ and\ passwords\ of\ the\ users >\ . \\ & < set\ up\ \ u_2\ with\ the\ data\ of\ the\ file\ system >\ . \\ & < set\ up\ \ u_3\ with\ the\ downloadable\ processes >\ . \\ & \big(\ NewUser(u_1)\ \mid\ Login(u_1)\ \mid\ Read(u_2)\ \mid \\ & \quad Write(u_2)\ \mid\ Execute(u_3)\ \big) \end{aligned}$$

Our example simplifies UNIX behaviour in two major aspects. Firstly, we did not require that a user must login before using the functionalities offered by the system; secondly, the files/programs are put by the system and not by the users. Both these choices were driven by the aim of simplifying the presentation; however, our setting could be easily enriched with more refined and realistic features.

Finally, we want to remark that, by exploiting the dummy data “*read*”, “*written*” and “*downloaded*”, we have been able to enforce an access control policy by only using region annotations. This confirms that, in spite of its simplicity, the approach we presented in this paper is very powerful.

7 Conclusions and Related Work

The main contribution of this paper is the introduction of a typing discipline for fixing the network region where data and processes can safely move. Our types can prevent execution of those actions that could compromise region specifications. To provide evidence of the generality of our approach, it has been applied to three paradigmatic calculi for global computing with quite different design choices. The technique developed works even when only a local knowledge of the net during the compilation can be assumed and misbehaving entities are present in the system. A few example applications implemented in all the three calculi have been also shown.

We want to remark that our theory permits to naturally implement a security mechanism based on *sandboxes*. We already noticed, in the introduction, that nodes can be seen as logical partitions of a single physical machine. By exploiting this intuition, one can split each machine into an appropriate number of nodes each with its own security policy. In this way, fine grained security policies can be programmed to guarantee that untrusted processes (e.g., coming from unchecked nodes) are accepted only at dedicated nodes and that from these nodes remote operations and spawning of threads are not permitted.

We conclude by commenting on the three instantiations of our region based approach; this should also shed light on the different choices and paradigms underlying the considered calculi. We shall also discuss possible extensions and confront our work with that of other researchers.

Assessment. Differently from that of Dπ and Ambients, the typing of cKLAIM requires some dynamic checks; this feature, however, is orthogonal to data secrecy and only depends on the underlying process calculus. In fact, these runtime checks could be avoided, and a typing discipline similar to that of Ambients could be developed for cKLAIM too, by additionally requiring that all data exchanged within a node have the same type. However, we find it too demanding to force a cKLAIM node to contain only data of the same type. On the contrary, Dπ channels can be reasonably assigned a fixed type because they can be seen as methods a node supplies to the processes it hosts, and ambients can be reasonably assigned the same type because, due to their hierarchical organisation, they could be thought of as logical partitions of the same memory space.

There is a thread-off between simplicity, efficiency and implementability. The type system for cKlaim is quite simple and easily implementable (types are just sets and operations on types are unions, intersections, subset inclusions, ...). Clearly, its runtime semantics is less efficient because of the dynamic checks in rules (cK-Out), (cK-Eval), (cK-In) and (cK-Match). Nevertheless, we consider reasonable this dynamic burden, since it only involves efficiently implementable operations on sets. On the contrary, Dπ and Ambients have very efficient runtime semantics (no type related check is present) at the expense of more involved static semantics. Also implementability is not straightforward: a preliminary implementation for the (simpler) type system of [19] appeared in a very technical paper [21]; type inference algorithms for Ambients-like calculi are even more complex (see, e.g., [29,6,12]). Since our type systems for Dπ and Ambient do rely on [19] and [4] respectively, we believe that they will also inherit the problems related to the implementation.

Finally, notice that in the setting of Dπ and Ambient explicit tagging of locations/data could seem useless, because no runtime check is ever performed. It might then appear more natural to leave the syntax untagged and record all type related information in the typing environment. The main drawback of this solution is that it would require a global (centralised) knowledge of all types. Tagging, instead, permits storing and inferring typing information locally, and keeps the formalisms closer to programmers' needs.

Possible extensions. Node regions could be handled more dynamically by extending the calculi we presented with actions for adding and removing nodes from regions (in this way, e.g., nodes could choose whether to trust newly created ones). However, this more expressive framework would require additional runtime checks. In particular, none of the two guarantees illustrated at the beginning of Section 3.1 could then be issued after static checks; the type system would then only permit inferring the regions of the arguments of process actions, and render dynamic checks more efficient.

Related Work. Much work has been recently devoted to designing languages for mobile processes that come equipped with security mechanisms based on, e.g., type systems [19,4,2,6,13] or control and data flow analysis [16,25,9,1]. The approach presented here is related to both these techniques. It exploits a type system to ensure confinement of data, and guarantees that the semantics respects the annotations by relying on the typing phase. Typing keeps track of data movements with a technique similar to control flow analysis.

Our work has been inspired by that on Confined-λ [20], a higher-order functional language that supports distributed computing by allowing expressions at different localities to communicate via channels. To limit the movement of values, programmers can assign regions to them; a type system is defined that guarantees that each value can roam only within the allowed region. There are however some differences with our approach. First of all, we consider not only channels but also other communication media that require a more dynamic typing mechanism. Then, we permit annotating only the relevant data while in [20] a programmer must declare a type (i.e. a region) for any constant, function and channel. When typing a net, we do not rely on any form of global knowledge of the system; only the annotations in the process are considered. We can infer this information by the local use of channels/ambients (in Dπ and Ambients resp.) or by just examining the code (in cKlaim). On the contrary, the type system in [20] assumes a global typing environment for handling shared channels; this somehow conflicts with the features of a global computing setting. Finally, we also give 'localised' formulations of the soundness theorem stating that well-typedness of a given subnet is preserved also in presence of untyped contexts. This is a crucial property for global computing systems where little assumptions on the behaviour of the context can be made.

Confinement has been also explored in the context of Java. In [27], *confined types* are introduced to confine classes and objects within specific packages. Then, in [28], a static type system based on confined types is defined for a Java-based calculus and its soundness is proved. Hence, in Java software modules play the role of our network regions, and confinement is associated rather with objects encapsulation than with movements of data and processes.

The group types, originally proposed for the Ambients calculus [4] and then recast to the π-calculus [3], have purposes similar to our region annotations. A group type is just a name that can be dynamically created but cannot be communicated (i.e. the scope of a group name cannot be extended). It permits to control name visibility in different regions of a net: a fresh name belonging to a fresh group can never be communicated to any process outside the scope of the group. Group types can then be used to handle processes and ambients movements, and in general to prevent accidental or malicious leakage of private names without using more complex dependent types (see, e.g., [22]). Notice that restricted names can be handled in a more flexible way in our framework by readily adapting the use of groups or, even better, of the *abstract names* from [22]. However, differently from our approach, when exploiting groups or abstract names some global knowledge is still necessary for taking into account the types of the names occurring free in a net.

Group types have also been used in region-based memory management where the focus is on efficiency, rather than on distribution and mobility. For instance, in [7] a connection between memory regions and group types is established and a variant of the π-calculus equipped with group types is used as a device to simplify the proof of correctness of dynamic memory management.

Finally, we want to consider a lower level approach to protect visibility of data via *encryption*. Encrypted data can appear everywhere in the net, but

(A-T-Empty)
$$\frac{}{\Gamma \vdash \mathbf{0}}$$

(A-T-Par)
$$\frac{\Gamma \vdash P_1 \qquad \Gamma \vdash P_2}{\Gamma \vdash P_1|P_2}$$

(A-T-Repl)
$$\frac{\Gamma \vdash P}{\Gamma \vdash *P}$$

(A-T-Amb)
$$\frac{\Gamma \vdash_n P}{\Gamma \vdash n[\![P]\!]}$$

(A-T-Res)
$$\frac{\Gamma, n \mapsto r_1 \rhd r_2 \rhd \{n\}[T] \vdash P}{\Gamma \vdash (\nu n)P}$$

Table A.1
Main Judgement for Typing Ambients

can be effectively used only by those users that know the decryption key. At an abstract level, we can consider the content of an encrypted message to be visible only within the region containing the nodes knowing the decryption key. Thus, it might appear that our approach could be implemented by resorting to cryptographic primitives. However, we would like to stress an important difference. When encryption is used, the producer of encrypted data can control the access to (plain) data only by controlling visibility of the decryption key. But this can be hardly controlled: once a decryption key has been passed on, information leakage can reveal the key, thus breaking the controllability of data. By exploiting our approach, the data producer can decide in advance which are the users enabled to access the data; this information is preserved during any evolution of the system. However, it should be noticed that indirect information flows can be generated; for an account of these problems and some possible solutions we refer the interested reader to [17,18].

Acknowledgements. We would like to thank Riccardo Focardi and Gianluigi Zavattaro for inviting us to present our work at the *SecCo'03* Workshop and for stimulating the preparation of this manuscript. The reviewers gave fruitful comments that helped in improving the paper.

A Technical Details for Ambients

The typing procedure for Ambients processes is presented in Tables A.1 and A.2. It is somehow inspired by the basic typing of [4] and also includes some features we have already presented for Dπ in Section 4.1.

The main judgement $\Gamma \vdash P$ is defined by the rules in Table A.1. Intuitively, it states that P is well-typed under the assumptions Γ. Differently from [5], we do not assign a type to processes and consider ill-typed those processes with actions or messages occurring outside any ambient boundary. The latter choice reflects our intuition of ambients as nodes of a net: a process cannot perform

(AA-T-Nil)
$$\frac{}{\Gamma \vdash_u \mathbf{0}}$$

(AA-T-Par)
$$\frac{\Gamma \vdash_u P_1 \qquad \Gamma \vdash_u P_2}{\Gamma \vdash_u P_1 | P_2}$$

(AA-T-Repl)
$$\frac{\Gamma \vdash_u P}{\Gamma \vdash_u *P}$$

(AA-T-Amb)
$$\frac{\mathit{val}(\Gamma(u)) \cup \mathit{cont}(\Gamma(u)) \subseteq \mathit{cont}(\Gamma(v)) \quad \Gamma \vdash_v P}{\Gamma \vdash_u v[P]}$$

(AA-T-Res)
$$\frac{\Gamma, n \mapsto r_1 \triangleright r_2 \triangleright \{n\}[T] \vdash_u P}{\Gamma \vdash_u (\nu n)P}$$

(AA-T-In)
$$\frac{\begin{array}{l}\mathit{val}(\Gamma(v)) \cup \mathit{cont}(\Gamma(v)) \subseteq \mathit{look}(\Gamma(u)) \\ \mathit{val}(\Gamma(u)) \cup \mathit{cont}(\Gamma(u)) \subseteq \mathit{cont}(\Gamma(v)) \qquad \Gamma \vdash_v P\end{array}}{\Gamma \vdash_v \mathbf{in}\_u.P}$$

(AA-T-Out)
$$\frac{\mathit{val}(\Gamma(v)) \cup \mathit{cont}(\Gamma(v)) \subseteq \mathit{look}(\Gamma(u)) \qquad \Gamma \vdash_v P}{\Gamma \vdash_v \mathbf{out}\_u.P}$$

(AA-T-Open)
$$\frac{\mathit{val}(\Gamma(v)) \cup \mathit{cont}(\Gamma(v)) \subseteq \mathit{look}(u) \qquad \Gamma \vdash_v P}{\Gamma \vdash_v \mathbf{open}\_u.P}$$

(AA-T-Rcv)
$$\frac{\Gamma(u) = r_1 \triangleright r_2 \triangleright r_3[T] \qquad \Gamma, x \mapsto T \vdash_u P}{\Gamma \vdash_u (x).P}$$

(AA-T-Snd)
$$\frac{\Gamma(v) = r_1 \triangleright r_2 \triangleright r_3[T] \quad \Gamma(u) = T \quad \mathit{val}(\Gamma(v)) \cup \mathit{cont}(\Gamma(v)) \subseteq \bigcup_{w \in r} \mathit{val}(\Gamma(w)) \subseteq \mathit{look}(T)}{\Gamma \vdash_v \langle [u]_r \rangle}$$

Table A.2
Auxiliary Judgement for Typing Ambients

any computational activity if it has not been allocated within some ambient. The main judgement relies on the auxiliary judgement $\Gamma \vdash_u P$ defined by the rules in Table A.2. This judgement is invoked in rule (A-T-Amb) of the main judgement and states that, when located within ambient u, process P can be typed in the environment Γ. As a matter of notation, $\Gamma, u \mapsto T$ will stand for the type environment Γ' such that $\Gamma'(v) = \Gamma(v)$, if $v \neq u$ and $u \notin \mathit{dom}(\Gamma)$, and $\Gamma'(v) = T$ if $v = u \notin \mathit{dom}(\Gamma)$.

We use functions $\mathit{look}(T)$, $\mathit{cont}(T)$ and $\mathit{val}(T)$ to denote the regions r_1, r_2 and r_3, respectively, when $T = r_1 \triangleright r_2 \triangleright r_3[\cdot]$; $\mathit{look}(\texttt{Shh})$ denotes $\top$. Finally, we also assume the following well-formedness condition on environments:

Let $\Gamma(v) = r_1 \triangleright r_2 \triangleright r_3[T]$. For every $n \in r_2$ it must be that $\mathit{cont}(\Gamma(n)) \subseteq r_2$. Moreover, if $v \in \mathcal{A}$ then $r_3 = \{v\}$; otherwise, for each $n \in r_3$, it must be that $r_1 \subseteq \mathit{look}(\Gamma(n))$ and $r_2 \subseteq \mathit{cont}(\Gamma(n))$.

We now briefly comment on some key features of the type system. For each occurrence of an identifier u, it is verified that the ambient containing the occurrence, and all the possibly enclosing ambients, can see u (see rules (AA-T-In), (AA-T-Out), (AA-T-Open) and (AA-T-Snd) – and, indirectly, also rule (AA-T-Amb)). The crucial rules are (AA-T-Amb) and (AA-T-In): they ensure that the ambient hierarchy always maintains the invariant (§) described in Section 5.1. Finally, rules (AA-T-Rcv) and (AA-T-Snd) exploit the topic of conversation to handle communications, i.e. to assign types to input variables or to verify that messages are sent at the type required by the ambient where the exchanges take place.

We now prove subject reduction and type safety for Ambients; the proofs are similar to those presented for Dπ in Section 4.3. Here, we only illustrate the most significant differences. First, Γ **with** A **for** P is the least environment that type checks P such that

$$\Gamma \textbf{ with } A \textbf{ for } P \triangleq \begin{cases} \Gamma & \text{if } A = \emptyset \\ (\Gamma \textbf{ with } A' \textbf{ for } P), n \mapsto r_1 \triangleright r_2 \triangleright \{n\}[T] & \text{if } A = A' \uplus \{n\} \end{cases}$$

The technical lemmas to establish subject reduction are, like before, subject congruence and substitutivity. Moreover, we give a Lemma (crucial in the case for **open**) that formally justifies the invariant (§) given in Section 5.1.

Lemma 5 (Subject Congruence) *If $\Gamma \vdash_u P$ and $P \equiv P'$ then $\Gamma \vdash_u P'$. The claim holds also when replacing $\vdash_u$ with $\vdash$.*

Lemma 6 (Substitutivity) *If $\Gamma, x \mapsto T \vdash_v P$ and $\Gamma(n) = T$, then $\Gamma \vdash_{v\{n/x\}} P\{n/x\}$.*

Proof: The proof proceeds by induction on $\Gamma, x \mapsto T \vdash_v P$. The proof is long and tedious because we must inspect all typing rules; here, we explicitly consider one of them, namely (AA-T-In). In what follows, for the sake of readability, we let $\Gamma' = \Gamma, x \mapsto T$.

Whenever (AA-T-In) is the last rule applied in the inference, it must be that $P = \mathbf{in}_{\_}u.Q$. Thus, $\mathit{val}(\Gamma'(v)) \cup \mathit{cont}(\Gamma'(v)) \subseteq \mathit{look}(\Gamma'(u))$, $\mathit{val}(\Gamma'(u)) \cup \mathit{cont}(\Gamma'(u)) \subseteq \mathit{cont}(\Gamma'(v))$ and $\Gamma' \vdash_v Q$. By induction, we have that $\Gamma \vdash_{v\{n/x\}} Q\{n/x\}$. Moreover, $\Gamma(u\{n/x\}) = \Gamma'(u)$; indeed, if $u = x$ then $\Gamma(u\{n/x\}) = \Gamma(n) = T = \Gamma'(x)$, otherwise $\Gamma(u\{n/x\}) = \Gamma(u) = \Gamma'(u)$. Similarly, $\Gamma(v\{n/x\}) = \Gamma'(v)$; this suffices to conclude the wanted $\Gamma \vdash_{v\{n/x\}} \mathbf{in}_{\_}(u\{n/x\}).Q\{n/x\}$. $\square$

Lemma 7 *If* $\Gamma \vdash_v P$ *and* $val(\Gamma(u)) \subseteq cont(\Gamma(v))$*, then* $\Gamma \vdash_u P$*.*

Proof: By induction on the length of judgement $\Gamma \vdash_v P$. The proof is easy because of well-formedness of Γ. $\square$

Theorem 6 (Subject Reduction) *Let* $A \triangleright P \longrightarrow A' \triangleright P'$*. Then*

(1) $\Gamma \vdash_n P$ *implies that* $(\Gamma$ **with** $A' - A$ **for** $P') \vdash_n P'$*;*
(2) P *is well-typed implies that* P' *is well-typed.*

Proof: The first claim is proved standardly, by induction on the length of the inference for $A \triangleright P \longrightarrow A' \triangleright P'$. The most interesting cases are the following ones:

(A-Open). In this case $P = \textbf{open}\_m.Q \mid m[\![R]\!]$, while $P' = Q \mid R$. By hypothesis, $\Gamma \vdash_n m[\![R]\!]$ and $\Gamma \vdash_n \textbf{open}\_m.Q$. By rules (AA-T-Amb) and (AA-T-Open), we also have that $\Gamma \vdash_m R$ and $\Gamma \vdash_n Q$; moreover, by Lemma 7, we have that $\Gamma \vdash_n R$, because, by the premise of rule (AA-T-Amb), we know that $n \in cont(\Gamma(m))$. This suffices to conclude.

(A-Comm). In this case $P = \langle [m]_r \rangle \mid (x).Q$, while $P' = Q\{m/x\}$. By hypothesis, $\Gamma \vdash_n \langle [m]_r \rangle$ and $\Gamma \vdash_n (x).Q$. These judgements imply that $\Gamma(m) = T$ and $\Gamma, x \mapsto T \vdash_n Q$, where $\Gamma(n) = r_1 \triangleright r_2 \triangleright \{n\}[T]$. By Lemma 6 it holds that $\Gamma \vdash_n Q\{m/x\}$, as required.

The second claim is proved similarly to Theorem 4. Indeed, it suffices to prove that Γ **with** $A' - A$ **for** P' is defined, where Γ is such that $\Gamma \vdash P$. The proof is by induction on $A \triangleright P \longrightarrow A' \triangleright P'$. Both for the base and for the inductive case, we only consider the most significant cases; the other ones are simpler.

(A-In). In this case $P = n[\![\textbf{in}\_m.Q_1 \mid Q_2]\!] \mid m[\![R]\!]$, while $P' = m[\![n[\![Q_1 \mid Q_2]\!] \mid R]\!]$; moreover, $A' = A$. Thus, we only prove that N' is well-typed in Γ. By hypothesis, $\Gamma \vdash_m R$, $\Gamma \vdash_n Q_2$ and $\Gamma \vdash_n \textbf{in}\_m.Q_1$. The last judgement implies that $\{m\} \cup cont(\Gamma(m)) \subseteq cont(\Gamma(n))$ and $\Gamma \vdash_n Q_1$. We can easily conclude.

(A-Res). In this case, $P = (\nu n)P'$ and $A' - A = \{n\}$. Now, Γ **with** $A' - A$ **for** P' is $\Gamma, n \mapsto r_1 \triangleright r_2 \triangleright \{n\}[T]$ if there exist some r_1, r_2 and T that type P'. The existence of these r_1, r_2 and T is ensured by the premise of rule (A-T-Res), that is the last rule used to infer $\Gamma \vdash (\nu n)P'$.

(A-Amb). By hypothesis, $P = n[\![Q]\!]$ and $P' = n[\![Q']\!]$, because $A \triangleright Q \longrightarrow A' \triangleright Q'$. Moreover, $\Gamma \vdash_n Q$; by the first claim of this Theorem, this implies that $(\Gamma$ **with** $A' - A$ **for** $Q') \vdash_n Q'$. Thus, $n[\![Q']\!]$ is well-typed, as required. $\square$

We now formulate and prove type safety and type soundness by following the guidelines of Section 4.3. We exploit function $reg_\Gamma(\cdot)$ that is defined inductively as follows:

$$reg_\Gamma(\mathbf{0}) \triangleq \emptyset \qquad reg_\Gamma(P|Q) \triangleq reg_\Gamma(P) \cap reg_\Gamma(Q) \qquad reg_\Gamma(u[\![P]\!]) \triangleq look(\Gamma(u))$$

$$reg_\Gamma(\mathbf{in}\,\_u.P) = reg_\Gamma(\mathbf{out}\,\_u.P) = reg_\Gamma(\mathbf{open}\,\_u.P) \triangleq look(\Gamma(u)) \cap reg_\Gamma(P)$$

$$reg_\Gamma(\langle [u]_r \rangle) \triangleq \bigcup_{w \in r} val(\Gamma(w)) \qquad reg_\Gamma(*P) = reg_\Gamma((x).P) \triangleq reg_\Gamma(P)$$

Definition 6 (Safety) *A process P is* Γ-safe *if for any sub-ambient $u[\![Q]\!]$ of P, it holds that $val(\Gamma(u)) \subseteq reg_\Gamma(Q)$.*

Theorem 7 (Type Safety) *If $\Gamma \vdash P$ then P is Γ-safe.*

Proof: It suffices to prove that $\Gamma \vdash_u P$ implies that $val(\Gamma(u)) \subseteq reg_\Gamma(P)$. This is done by a standard induction on the typing judgement. □

Like in Dπ, a localised formulation of type soundness can be given. Indeed, we can define the Γ-r-subprocess of P as the process containing all the sub-ambients $n[\![Q]\!]$ of P such that $\{n\} \cup \{m : n \in cont(\Gamma(m))\} \cup cont(\Gamma(n)) \subseteq r$. Thus, $n[\![Q]\!]$ is part of the Γ-r-subprocess of P if and only if r contains n, all the ambients that n can contain and all the ambients that can contain n. This amounts to say that at least all the subtree rooted in n and all the ancestors of n in the ambient hierarchy must be typed to ensure the safety of data occurring in n. Again, since the Γ-$\top$-subprocess of P is P itself, the following theorem also states the soundness result when P is fully typed.

Theorem 8 (Localised Type Soundness) *Let the Γ-r-subnet of P be defined and well-typed in Γ. If $A \triangleright P \longrightarrow^* A' \triangleright P'$, then the Γ'-r'-subnet of P' is defined and Γ'-safe, where $r' = r \cup (A' - A)$ and $\Gamma' = \Gamma$ **with** $A' - A$ **for** P'.*

References

[1] C. Braghin, A. Cortesi, and R. Focardi. Security Boundaries in Mobile Ambients. *Computer Languages*, 28(1):101–127, Nov 2002.

[2] M. Bugliesi, G. Castagna, and S. Crafa. Boxed ambients. In *Proc. of TACS 2001*, volume 2215 of *LNCS*, pages 38–63. Springer, 2001.

[3] L. Cardelli, G. Ghelli, and A. D. Gordon. Secrecy and group creation. In C. Palamidessi, editor, *Proc. of CONCUR 2000*, volume 1877 of *LNCS*, pages 365–379. Springer, 2000.

[4] L. Cardelli, G. Ghelli, and A. D. Gordon. Types for the ambient calculus. *Journal of Information and Computation*, 177(2):160–194, 2002.

[5] L. Cardelli and A. D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, 2000.

[6] M. Coppo, M. Dezani-Ciancaglini, E. Giovannetti, and I. Salvo. M^3: Mobility types for mobile processes in mobile ambients. In *Proc. of CATS'02*, volume 78 of *ENTCS*. Elsevier, 2002.

[7] S. Dal-Zilio and A. D. Gordon. Region analysis and a pi-calculus with groups. In M. Nielsen and B. Rovan, editors, *Proceedings of MFCS 2000*, volume 1893 of *LNCS*, pages 409–424. Springer, 2000.

[8] R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: a Kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.

[9] P. Degano, F. Levi, and C. Bodei. Safe ambients: Control flow analysis and security. In *Proc. of ASIAN'00*, volume 1961 of *LNCS*, pages 199–214. Springer, 2000.

[10] Ú. Erlingsson and F. B. Schneider. SASI enforcement of security policies: A retrospective. In *New Security Paradigms Workshop*, pages 87–95. ACM Press, 1999.

[11] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[12] E. Giovannetti. Type inference for mobile ambients in prolog. In *Proc. of CATS'04*, ENTCS. Elsevier, 2004.

[13] D. Gorla and R. Pugliese. Resource access and mobility control with dynamic privileges acquisition. In J. Baeten, J. Lenstra, J. Parrow, and G. Woeginger, editors, *Proc. of ICALP'03*, volume 2719 of *LNCS*, pages 119–132. Springer-Verlag, 2003.

[14] D. Gorla and R. Pugliese. A semantic theory for global computing system. Research report, Dipartimento di Sistemi e Informatica, Università di Firenze, 2003. Available at `http://www.dsi.unifi.it/~pugliese/DOWNLOAD/-bis4k-full.pdf`.

[15] D. Gorla and R. Pugliese. Controlling data movement in global computing applications. In *Proc. of 19th Annual ACM-SIGAPP Symposium on Applied Computing (SAC'04)*. ACM Press, 2004.

[16] R. R. Hansen, J. G. Jensen, F. Nielson, and H. R. Nielson. Abstract interpretation of mobile ambients. In *Proc. of SAS 1999*, volume 1694 of *LNCS*, pages 134–148. Springer, 1999.

[17] M. Hennessy. The security pi-calculus and non-interference. In *Proc. of MFPS XIX*, ENTCS. Elsevier, 2003. Full version to appear in *Journal of Logic and Algebraic Programming.*

[18] M. Hennessy and J. Riely. Information flow vs. resource access in the asynchronous pi-calculus. In U. Montanari, J. Rolim, and E. Welzl, editors, *Proc. of ICALP 2000*, volume 1853 of *LNCS*, pages 415–427. Springer, 2000.

[19] M. Hennessy and J. Riely. Resource Access Control in Systems of Mobile Agents. *Information and Computation*, 173:82–120, 2002.

[20] Z. D. Kirli. Confined mobile functions. In *Proc. of the 14th CSFW*, pages 283–294. IEEE Computer Society, 2001.

[21] C. Lhoussaine. Type inference for a distributed pi-calculus. In *Proc. of ESOP'03*, volume 2618 of *LNCS*, pages 253–268. Springer-Verlag, 2003. Full version to appear in *Science of Computer Programming.*

[22] C. Lhoussaine and V. Sassone. A dependently typed ambient calculus. In *Proceedings of ESOP'04*, volume 2986 of *LNCS*, pages 171–187. Springer-Verlag, 2004.

[23] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part I/II. *Journal of Information and Computation*, 100:1–77, Sept. 1992.

[24] G. Necula. Proof-Carrying Code. In *Proceedings of POPL '97*, pages 106–119. ACM Press, 1997.

[25] F. Nielson, H. R. Nielson, R. R. Hansen, and J. G. Jensen. Validating firewalls in mobile ambients. In J. C. Baeten and S. Mauw, editors, *Proc. of CONCUR '99*, volume 1664 of *LNCS*, pages 463–477. Springer, 1999.

[26] F. B. Schneider, G. Morrisett, and R. Harper. A language-based approach to security. In *Informatics: 10 Years Ahead, 10 Years Back*, volume 2000 of *LNCS*, pages 86–101. Springer, 2000.

[27] J. Vitek and B. Bokowski. Confined types in java. *Software - Practice and Experience*, 31(6):507–532, 2001.

[28] T. Zhao, J. Palsber, and J. Vitek. Lightweight confinement for featherweight java. In *Proc. of the 18th OOPSLA*, pages 135–148. ACM Press, 2003.

[29] P. Zimmer. Subtyping and typing algorithms for mobile ambients. In *Proc. of FoSSaCS'00*, volume 1784 of *LNCS*, pages 375–390. Springer, 2000.