

The M-calculus: A Higher-Order Distributed Process Calculus

Alan Schmitt, Jean-Bernard Stefani
INRIA
{alan.schmitt, jean-bernard.stefani}@inria.fr

December 2002

Abstract

This paper presents a new distributed process calculus, called the M-calculus, that can be understood as a higher-order version of the Distributed Join calculus with programmable localities. The calculus retains the implementable character of the Distributed Join calculus while overcoming several important limitations: insufficient control over communication and mobility, absence of dynamic binding, and limited locality semantics. The calculus is equipped with a polymorphic type system that guarantees the unicity of locality names, even in presence of higher-order communications – a crucial property for the determinacy of message routing in the calculus.

1 Introduction

Among the process calculi which have been introduced over the past decade to serve as a basis for a distributed and mobile programming model, the Join calculus [7, 6, 11] constitutes an interesting milestone. It provides a distributed programming model with hierarchical fail-stop localities, transparent mobility and communications, and it can be efficiently implemented. The Join calculus, however, has several limitations:

- It offers insufficient control over communication and process mobility, which is an issue in distributed environments where security is a primary concern. For instance, it is not possible to prevent a locality from migrating to another locality, except by forcing its failure. Also, once a resource (a Join calculus definition) has been defined and communicated, it is very difficult to prevent access to that resource or to define the equivalent of a firewall [4].
- It does not support dynamic binding. In a distributed programming model, it is important to provide both local and remote equivalent of libraries or services, because of the cost, safety, and security considerations that may apply. Thus, it should be possible to access identically named libraries or services (like a print service) at different sites. In the Join calculus such a choice is not directly available since each definition is uniquely defined: every resource is permanently bound to a single locality.
- It does not support the definition of localities with different semantics. For instance, localities in the Join calculus are defined to be fail-stop. While it would be possible to change the semantics of the calculus to accommodate different failure modes (such as omission or byzantine failures), the question remains as to how one can combine different failure modes

This work has been supported in part by the Mikado IST Global Computing Project (IST-2001-32222)

within the same calculus. Likewise, one could require a locality to be endowed with a particular form of access control (e.g. restricting access to resources within a locality to principals appearing in an access control list).

The M-calculus presented in this paper is designed to overcome the limitations of the Join calculus while preserving some of its key features, notably, its concept of hierarchical localities (which is crucial to deal with security, migration, and failures), its notion of multiway synchronization, and its implementable character.

Specifically, the main contributions of this paper are:

- the notion of *programmable locality*, that generalizes the different concepts of locality found in the Join calculus and other distributed process calculi;
- the conjunction of higher-order processes and hierarchical programmable localities to unify communication and process migration, combining the possibility of transparent routing as in the Join calculus with fine-grained ambient-like control over information exchange;
- the introduction of a passivation operator as a key primitive for programming different forms of control that can be exercised by localities;
- the definition of a type system that guarantees the determinacy of the routing mechanism by ensuring that every locality bears a unique name, even in presence of higher-order communication.

A programmable locality in the M-calculus (or locality, in brief) has a name and contains two processes: a *controller*, which filters incoming and outgoing messages, and a *content*. In order to apply the same control mechanisms to remote communication and process migration, the latter is just communication of a *thunk*, i.e. a frozen process. A running locality may be frozen by its controller using the *passivation* operator. This operator takes a function that defines the operations to apply on the controller and content processes of a locality, such as sending them in a remote message, discarding them, or modifying them.

The paper is organized as follows. Section 2 defines the syntax and operational semantics of the M-calculus. Section 3 introduces a type system that ensures the unicity of locality names. Section 4 gives an encoding of the Join calculus that illustrate the versatility of the calculus. Section 5 discusses related work. Section 6 concludes the main body of paper. Appendix A presents the proof of the safety properties of the type system of section 3.

2 The M-calculus

We present in this section the M-calculus, introducing the syntax and the semantics as we describe local communication, remote communication, control, and migration. The syntax is summarized in figures 1 and 2. In the following, we use bold fonts for identifiers that may stand for names or variables, as described in figure 2.

Communication takes the form of an asynchronous, channel-based, point-to-point exchange of messages, reflecting the dominant mode of communication in current large scale networks. Channels are called *resources* and we assume there is an infinite countable set of resource names. We let r range over this set. A local message is an application of a resource name to a tuple of values $r\tilde{V}$. Receivers in the M-calculus are reaction rules composed of a multiway synchronization pattern (similar to the one proposed informally by Milner for his “Polynomial π -calculus” and to Join patterns), and of a guarded process. Every reaction rule $\langle r_1\tilde{x}_1 \mid \dots \mid r_n\tilde{x}_n \triangleright P \rangle$ defines the resource names r_1, \dots, r_n . The formal parameters $\tilde{x}_1, \dots, \tilde{x}_n$ are tuples of *variables*, and we assume there is an infinite countable set of variables. The local communication rule is very similar to the JOIN rule of the Join calculus, substituting message arguments for formal arguments in the guarded process (rule R.RES of figure 5). We remark that the reaction rule is replicated: it does

$P ::=$	$\mathbf{0}$ V $\mathbf{a}(P)[P]$ $P \mid P$ PP (P, \dots, P) $\nu n.P$ $([\mu = V]P, P)$ $\langle J \triangleright P \rangle$ $\mathbf{pass} V$	process null process value locality parallel composition application tuple restriction conditional reaction rule passivation
$V ::=$	$()$ u (V, \dots, V) $\lambda x.P$	value null value name tuple abstraction
$J ::=$	$\mathbf{r}\tilde{x}$ $J \mid J$	Join pattern message synchronization

Figure 1: Syntax of the M-calculus

not disappear after reduction. We also remark that a reaction rule does not bind its defined names. New resource names are introduced and bound using a restriction operator $\nu r.P$.

In the reduction rules introduced in figure 5, rule R.CONTEXT uses evaluation contexts defined in figure 4, and rule R.EQUIV uses structural equivalence. Structural equivalence, \equiv , is the smallest equivalence relation that satisfies the rules given in Figure 6, where the parallel composition operator \mid for processes is taken to be commutative and associative, with $\mathbf{0}$ as its neutral element. The structural rules comprise scope extrusion rules for the restriction operator, standard rules for equivalence under α -conversion, and congruence for evaluation contexts. Equivalence of two processes P and Q up to α -conversion is noted $P =_\alpha Q$. We recall that in $\nu n.P$, $\lambda x.P$, and $\langle \mathbf{r}_1 \tilde{x}_1 \mid \dots \mid \mathbf{r}_n \tilde{x}_n \triangleright P \rangle$, the names and variables n , x , and \tilde{x}_i are bound in P . Free names of a process P are defined as usual and written $fn(P)$. This set is formally defined in figure 3. We recall that the defined names of a reaction rule $\langle J \triangleright P \rangle$ are free.

As a communication example, we may write a reference cell process as in the Join calculus:

$$\left(\begin{array}{l} \nu s. \quad \langle get(k) \mid s(st) \triangleright k(st) \mid s(st) \rangle \\ \quad \mid \quad \langle set(st') \mid s(st) \triangleright s(st') \rangle \\ \quad \mid \quad s(0) \end{array} \right) \mid get(print) \mid set(3)$$

which may reduce to $print(0)$ or $print(3)$.

We now describe remote communication of asynchronous messages. In many calculi, remote communication involves two steps: resolving where to send the message, and sending it. For instance, in the distributed Join calculus, every channel is defined in at most one location and definitions cannot move from one location to another. Thus a defined channel name is unambiguously associated to the location containing its definition. In the dynamic Join calculus [20], the destination for a dynamic message is resolved according to the channel name and the current position of the message. In the Ambient calculus, an ambient migrates according to the explicit capabilities that it expresses and its local environment. The destination of a message in the Box- π calculus [21] also depends on the immediate environment of the message. In order to avoid restricting the calculus to one particular semantics, we let the resolving step be a part of the calculus: a remote message has the form $a.r\tilde{V}$, where a is the explicit destination of the message, which can be thus

$n ::=$		resolved name
	r	resource name
	$ $	
	a	locality name
$\mathbf{r} ::=$		variable resource name
	r	resource name
	$ $	
	x	variable
$\mathbf{a} ::=$		variable locality name
	a	locality name
	$ $	
	x	variable
$u ::=$		name
	\mathbf{a}	variable locality name
	$ $	
	\mathbf{r}	variable resource name
	$ $	
	$\mathbf{a.r}$	located resource
$\mu ::=$		name pattern
	n	resolved name
	$ $	
	x	variable
	$ $	
	$-$	wildcard

Figure 2: Names

$fn(\mathbf{a}) = \{\mathbf{a}\}$	$fn(\mathbf{r}) = \{\mathbf{r}\}$
$fn(\mathbf{a.r}) = \{\mathbf{a}, \mathbf{r}\}$	$fn(_) = \emptyset$
$fn(_) = fn(\mathbf{0}) = \emptyset$	$fn(\lambda x.P) = fn(P) \setminus \{x\}$
$fn(\nu n.P) = fn(P) \setminus \{n\}$	$fn(PQ) = fn(P) \cup fn(Q)$
$fn(P_1, \dots, P_p) = fn(P_1) \cup \dots \cup fn(P_p)$	$fn(\langle J \triangleright P \rangle) = (fn(P) \setminus fn(J)) \cup dn(J)$
$fn(\mathbf{r}_1 \tilde{x}_1 \mid \dots \mid \mathbf{r}_q \tilde{x}_q) = fn(\mathbf{r}_1 \tilde{x}_1) \cup \dots \cup fn(\mathbf{r}_q \tilde{x}_q)$	$fn(\mathbf{r}(x_1, \dots, x_p)) = \{\mathbf{r}, x_1, \dots, x_p\}$
$fn(\mathbf{a}(P)[Q]) = \{\mathbf{a}\} \cup fn(P) \cup fn(Q)$	$fn(P \mid Q) = fn(P) \cup fn(Q)$
$fn([\mu = V]P, Q) = fn(\mu) \cup fn(V) \cup fn(P) \cup fn(Q)$	$fn(\text{pass } V) = fn(V)$
$fn(V_1, \dots, V_p) = fn(V_1) \cup \dots \cup fn(V_p)$	

Figure 3: Free names

chosen by the programmer. We see that an addressed resource $a.r$ is composed of an address (a locality name), which may correspond to an IP address, and a resource name, which may correspond to a port number. This construction is similar to the high-level $c@a$ construct of Nomadic Pict [24]. The second step of remote communication is the actual sending of the message to the remote locality. This communication step might be direct independently of the relative positions of the message and the destination, as in the Join calculus, or it might involve several local steps, following the structure of localities to reach the destination, as in the Ambient calculus. We remark that these two models may coincide when considering a flat model of localities.

Just as in the Join calculus and in Ambient calculi, we retain the idea of hierarchically organized localities, a crucial feature for capturing the spatial and logical partitioning of control in distributed systems. We assume there is an infinite countable set of locality names, and we let a, b range over

$$\mathbf{E} ::= (\cdot) \mid \mathbf{E}V \mid P\mathbf{E} \mid \nu n.\mathbf{E} \mid (\mathbf{E} \mid P) \mid (P \mid \mathbf{E}) \mid a(P)[\mathbf{E}] \mid a(\mathbf{E})[P] \mid (P_1, \dots, \mathbf{E}, \dots, P_n)$$

Figure 4: Evaluation Contexts

$$\begin{array}{c}
\frac{}{(\lambda x.P)V \rightarrow P\{V/x\}} \text{ [R.BETA]} \quad \frac{\text{match}(\mu, V)}{([\mu = V]P, Q) \rightarrow P} \text{ [R.IF.THEN]} \quad \frac{\neg \text{match}(\mu, V)}{([\mu = V]P, Q) \rightarrow Q} \text{ [R.IF.ELSE]} \\
\\
\frac{}{a(\text{pass } V \mid P)[Q] \rightarrow Va(\lambda.P)(\lambda.Q)} \text{ [R.PASSIV]} \quad \frac{\langle J \triangleright P \rangle = \langle r_1 \widetilde{x}_1 \mid \dots \mid r_n \widetilde{x}_n \triangleright P \rangle}{\langle J \triangleright P \rangle \mid r_1 \widetilde{V}_1 \mid \dots \mid r_n \widetilde{V}_n \rightarrow \langle J \triangleright P \rangle \mid P\{\widetilde{V}_i/\widetilde{x}_i\}} \text{ [R.RES]} \\
\\
\frac{P \rightarrow Q}{\mathbf{E}(P) \rightarrow \mathbf{E}(Q)} \text{ [R.CONTEXT]} \quad \frac{P \equiv P' \quad P' \rightarrow Q' \quad Q' \equiv Q}{P \rightarrow Q} \text{ [R.EQUIV]}
\end{array}$$

Figure 5: Reduction: Computing Rules

$$\begin{array}{c}
\frac{n \notin \text{fn}(Q)}{(\nu n.P) \mid Q \equiv \nu n.P \mid Q} \text{ [S.NU.PAR]} \quad \frac{n \notin \text{fn}(Q) \wedge n \neq a}{a(\nu n.P)[Q] \equiv \nu n.a(P)[Q]} \text{ [S.NU.CTRL]} \\
\\
\frac{n \notin \text{fn}(P) \wedge n \neq a}{a(P)[\nu n.Q] \equiv \nu n.a(P)[Q]} \text{ [S.NU.CONT]} \quad \frac{P =_\alpha Q}{P \equiv Q} \text{ [S.\alpha]} \quad \frac{P \equiv Q}{\mathbf{E}(P) \equiv \mathbf{E}(Q)} \text{ [S.CONTEXT]}
\end{array}$$

Figure 6: Structural Equivalence

this set. New locality names are introduced and bound by the restriction operator $\nu a.P$. To define the routing rules, we write $\text{locs}(P)$ for the multiset of unrestricted *active localities* of P . This multiset is formally defined in figure 8. In the rest of this paper, we say that a process P is *active* in Q if Q is structurally equivalent to a process of the form $\mathbf{E}(P)$ for some evaluation context \mathbf{E} . Localities in the M-calculus provide the means to enforce some control on incoming and outgoing messages. This control may be arbitrarily complex, may require maintaining some state, and should be kept separate from the program running in the locality. For this reason, localities take the form $a(P)[Q]$ where a is the name of the locality, P is a process controlling the locality and its interactions with the environment, and Q is the content of the locality. The first role of the controller is to filter incoming and outgoing messages. To this end, we introduce two special resource names i and o on which incoming and outgoing messages are intercepted (rules R.A.IN and R.A.OUT in figure 7). On interception, an incoming or outgoing message is split into three parts: the destination address, the targeted resource, and the message arguments. The controller should provide a reaction rule for these filtering channels, implementing the desired behavior. For instance, a locality that does not want to block any message could contain the process Fwd in its controller, where:

$$Fwd \stackrel{\text{def}}{=} \langle i(x, y, z) \triangleright x.y z \rangle \mid \langle o(x, y, z) \triangleright x.y z \rangle$$

We remark that this definition is stateless and relies on the other routing rules to send the message to its final destination automatically: even though the routing is step by step, it is not necessary to specify how to take each step. This is much different from the Ambient calculus where an explicit path to the target ambient needs to be given.

As localities form a tree, there is no notion of site as in Nomadic Pict (a site can be modeled by a locality at a given level in the tree) and the routing algorithm is part of our semantics. It is however possible to express different routing algorithms by forwarding messages from locality to locality using some specified resource. One example of this is the simulation of the dynamic Join calculus in the M-calculus described in section 4.3. It is also possible to intercept and reroute messages using the control mechanism, as shown below.

A message present in the controller is considered as having been controlled and may freely leave the controller (rules R.A.CTRL.TO.CONT and R.A.CTRL.TO.ENV). When a message has reached its final destination, it becomes a local message (rules R.A.CTRL.FINAL and R.A.CONT.FINAL).

$$\begin{array}{c}
\frac{}{a(P \mid a.r\tilde{V})[Q] \rightarrow a(P \mid r\tilde{V})[Q]} \text{[R.A.CTRL.FINAL]} \\
\\
\frac{}{a(P)[Q \mid a.r\tilde{V}] \rightarrow a(P)[Q \mid r\tilde{V}]} \text{[R.A.CONT.FINAL]} \quad \frac{b \in \text{locs}(P) \cup \text{locs}(Q) \cup \{a\}}{b.r\tilde{V} \mid a(P)[Q] \rightarrow a(P \mid i(b, r, \tilde{V}))[Q]} \text{[R.A.IN]} \\
\\
\frac{b \in \text{locs}(Q) \setminus \text{locs}(P) \quad b \neq a}{a(P \mid b.r\tilde{V})[Q] \rightarrow a(P)[Q \mid b.r\tilde{V}]} \text{[R.A.CTRL.TO.CONT]} \\
\\
\frac{b \notin \text{locs}(P) \cup \text{locs}(Q) \quad b \neq a}{a(P \mid b.r\tilde{V})[Q] \rightarrow a(P)[Q] \mid b.r\tilde{V}} \text{[R.A.CTRL.TO.ENV]} \\
\\
\frac{b \notin \text{locs}(Q) \quad b \neq a}{a(P)[Q \mid b.r\tilde{V}] \rightarrow a(P \mid o(b, r, \tilde{V}))[Q]} \text{[R.A.OUT]}
\end{array}$$

Figure 7: Reduction: Routing Rules (Addressed Messages)

$$\begin{array}{ll}
\text{locs}(\nu n.P) = \text{locs}(P) \setminus \{n\} & \text{locs}(PQ) = \emptyset \\
\text{locs}(\mathbf{a}(P)[Q]) = \mathbf{a}, \text{locs}(P), \text{locs}(Q) & \text{locs}(\langle J \triangleright Q \rangle) = \emptyset \\
\text{locs}(P \mid Q) = \text{locs}(P), \text{locs}(Q) & \text{locs}(\langle [\mu = V]P, Q \rangle) = \emptyset \\
\text{locs}(\mathbf{pass} V) = \emptyset & \text{locs}(V) = \emptyset \\
\text{locs}(\mathbf{0}) = \emptyset & \text{locs}(P_1, \dots, P_q) = \emptyset
\end{array}$$

Figure 8: Multiset of Active Localities

Local messages may move freely from controller to content and vice versa (figure 9), depending on where the resource is defined. To this end, we call *defined local names* the set of resources that are defined in a given process without inspecting sublocalities. This set is formally defined in figure 10. We give an example of transparent incoming message routing in figure 11.

One interesting feature when writing a reaction rule for the filtering channels i and o is to be able to test the target locality or resource. To this end, we introduce a simple name matching operator $([\mu = V]P, Q)$, whose semantics (rules R.IF.THEN and R.IF.ELSE in figure 5) rely on a *match()* predicate, which is true only in the following cases:

$$\text{match}(_, V) \quad \text{match}(n, n) \quad \text{match}(x, x)$$

For instance, a filter for incoming messages could be:

$$\langle i(d, r, v) \triangleright [a = d](b.r \ v, \mathbf{0}) \rangle$$

This filter throws away any message that is not for locality a , and reroutes messages for a to b .

As in the Ambient calculus and the Join calculus, we provide a way to modify the tree structure of localities. However, we want to be able to control incoming and outgoing localities at every

$$\begin{array}{c}
\frac{r \notin \text{dln}(P) \quad r \in \text{dln}(Q)}{a(P \mid r\tilde{V})[Q] \rightarrow a(P)[Q \mid r\tilde{V}]} \text{[R.L.CTRL.TO.CONT]} \\
\\
\frac{r \in \text{dln}(P) \quad r \notin \text{dln}(Q)}{a(P)[Q \mid r\tilde{V}] \rightarrow a(P \mid r\tilde{V})[Q]} \text{[R.L.CONT.TO.CTRL]}
\end{array}$$

Figure 9: Reduction: Routing Rules (Local Messages)

$$\begin{array}{ll}
dln(PQ) = \emptyset & dln(\nu n.P) = dln(P) \setminus \{n\} \\
dln((P_1, \dots, P_q)) = \emptyset & dln(\langle \mathbf{r}_1 \tilde{x}_1 \mid \dots \mid \mathbf{r}_q \tilde{x}_q \triangleright P \rangle) = \{\mathbf{r}_1, \dots, \mathbf{r}_q\} \\
dln(\mathbf{a}(P)[Q]) = \emptyset & dln(P \mid Q) = dln(P) \cup dln(Q) \\
dln([\mu = V]P, Q) = \emptyset & dln(\mathbf{pass} V) = \emptyset \\
dln(\mathbf{0}) = \emptyset & dln(V) = \emptyset
\end{array}$$

Figure 10: Defined local names

$$\begin{array}{l}
\underline{\mathbf{a.r}\tilde{V}} \mid b(\langle i(d, r, v) \triangleright d.rv \rangle) [\underline{\mathbf{a}}(\langle i(d, r, v) \triangleright d.rv \rangle)(\langle r = \dots \rangle)] \\
\rightarrow b(\langle i(d, r, v) \triangleright d.rv \rangle \mid \underline{\mathbf{i(a, r, \tilde{V})}}) [\underline{\mathbf{a}}(\langle i(d, r, v) \triangleright d.rv \rangle)(\langle r = \dots \rangle)] \\
\rightarrow b(\langle i(d, r, v) \triangleright d.rv \rangle \mid \underline{\mathbf{a.r}\tilde{V}}) [\underline{\mathbf{a}}(\langle i(d, r, v) \triangleright d.rv \rangle)(\langle r = \dots \rangle)] \\
\rightarrow b(\langle i(d, r, v) \triangleright d.rv \rangle) [\underline{\mathbf{a.r}\tilde{V}} \mid \underline{\mathbf{a}}(\langle i(d, r, v) \triangleright d.rv \rangle)(\langle r = \dots \rangle)] \\
\rightarrow b(\langle i(d, r, v) \triangleright d.rv \rangle) [\underline{\mathbf{a}}(\langle i(d, r, v) \triangleright d.rv \rangle \mid \underline{\mathbf{i(a, r, \tilde{V})}})(\langle r = \dots \rangle)] \\
\rightarrow b(\langle i(d, r, v) \triangleright d.rv \rangle) [\underline{\mathbf{a}}(\langle i(d, r, v) \triangleright d.rv \rangle \mid \underline{\mathbf{a.r}\tilde{V}})(\langle r = \dots \rangle)] \\
\rightarrow b(\langle i(d, r, v) \triangleright d.rv \rangle) [\underline{\mathbf{a}}(\langle i(d, r, v) \triangleright d.rv \rangle \mid \underline{\mathbf{r}\tilde{V}})(\langle r = \dots \rangle)] \\
\rightarrow b(\langle i(d, r, v) \triangleright d.rv \rangle) [\underline{\mathbf{a}}(\langle i(d, r, v) \triangleright d.rv \rangle)(\langle r = \dots \rangle \mid \underline{\mathbf{r}\tilde{V}})]
\end{array}$$

Figure 11: Remote communication example

locality boundary, as is possible with remote communication. We thus unify migration and remote communication by considering migration as the communication of a frozen process. A frozen process is of the form $\lambda.P$, and may be unfrozen by applying it to the null value $()$. We actually consider a generalization by embedding a call by value λ -calculus within our calculus (with the usual β reduction rule R.BETA of figure 5). We use standard notational conventions: in a term $\lambda x.P$ or $\nu n.P$, the scope extends as far to the right as possible; $PQ_1 \dots Q_n$ stands for $(\dots (PQ_1) \dots Q_n)$; $\lambda x_1 \dots x_q.P$ stands for $\lambda x_1 \dots \lambda x_q.P$; $\nu n_1 \dots n_q.P$ stands for $\nu n_1 \dots \nu n_q.P$. We also make use of the notation $\lambda.P$ to stand for a *thunk* $\lambda x.P$, with x not free in P .

The *passivation* primitive $\mathbf{pass} V$, where V is a function expecting a locality name and two frozen processes, is introduced to freeze running processes. Passivation is the second role of a controller: when evaluated in the controller of a locality $\mathbf{a}(\mathbf{pass} V \mid P)[Q]$, the locality is split into three parts: its name a , its frozen controller $\lambda.P$, and its frozen content $\lambda.Q$. These parts are given as arguments to the function V :

$$\mathbf{a}(\mathbf{pass} V \mid P)[Q] \rightarrow V \ a \ (\lambda.P) \ (\lambda.Q)$$

For instance, a function $V = \lambda x p q.x(p())[q()]$ simply recreates the passivated locality.

Locality mobility can be implemented using higher-order messages and passivation (cf the `go` construct in the Join calculus and ambients `in` and `out` capabilities). Locality $Q^m(a)$ below can be moved to a different locality:

$$\begin{array}{ll}
Q^m(a) & = \ \mathbf{a}(\mathbf{Fwd} \mid \langle \mathbf{go} \ u \triangleright \mathbf{Go}(u) \rangle)[Q] \\
\mathbf{Go}(u) & = \ \mathbf{pass} \ \lambda x p q.(u.\mathbf{enter} \ \lambda.x(p()))[q()]
\end{array}$$

A request `go b`, results in the passivation of the locality a and its sending as a thunk to the resource `enter` of the locality named b . If the request comes from the outside of the locality, the result is an objective form of move. If the request comes from the content of the locality, the result is a subjective form of move. The controller of locality b can contain the process *Enter* below to allow the insertion of a new locality in its content:

$$\mathbf{Enter} \ = \ \langle \mathbf{enter} \ f \triangleright \mathbf{pass} \ \lambda x p q.x(p())[q()] \mid f() \rangle$$

Passivation may also be used to implement various forms of control on a locality. Locality $Q^\circ(a)$ below can be suspended, resumed, dissolved (cf the `open` capability of ambients), and updated with a new controller (we note simply r a message of the form $r()$):

$$\begin{aligned}
Q^\circ(a) &= \nu s \text{ on}.L(a, s, \text{on}) \\
L(a, s, \text{on}) &= a \left(\begin{array}{l} \text{Fwd} \mid \langle \text{suspend} \mid \text{on} \triangleright S(s) \rangle \\ \mid \langle \text{resume} \mid s f \triangleright R(f, \text{on}) \rangle \\ \mid \langle \text{open} \triangleright O \rangle \mid \langle \text{update } f \triangleright U(f) \rangle \mid \text{on} \end{array} \right) [Q] \\
S(s) &= \text{pass } \lambda x p q. x(p()) \mid (s q) [0] \\
R(f, \text{on}) &= \text{pass } \lambda x p q. x(p()) \mid \text{on} [f()] \\
O &= \text{pass } \lambda x p q. q() \\
U(f) &= \text{pass } \lambda x p q. x(f()) [q()]
\end{aligned}$$

As usual, we take the reduction relation for the M-calculus, \rightarrow , as the smallest relation that satisfies the rules given in Figures 5, 7, and 9.

We present as an example an M-calculus variant of the taxi example presented by L. Cardelli in [3]. The basic idea is as follows: a passenger enters a taxi by specifying a route (list of localities to enter), a sitting behavior (to be executed while sitting in the taxi), and a continuation behavior (to be executed on arrival) – L. Cardelli’s version only comprises a route and a continuation behavior.

A route consists of a list of locality names a_1, \dots, a_n , which we represent by the frozen process

$$M = \lambda. \text{route}(a_1, \lambda. \text{route}(a_2, \dots, \lambda. \text{route}(a_n, \lambda. \text{route}(\text{nil}, \lambda. \mathbf{0})) \dots))$$

where `nil` is a special locality name reserved to signal the end of the list.

A passenger requests entrance in the taxi named `tx` by emitting a message

$$\text{tx.enter}(M, \lambda. S, \lambda. C)$$

where S is the sitting behavior, and C is the continuation behavior.

A taxi named `tx` is defined as follows:

$$\begin{aligned}
\text{Taxi} &= \nu f. \text{tx}(\langle \text{Enter} \rangle \mid \langle \text{Exit} \rangle \mid \langle \text{Route} \rangle \mid f) [0] \\
\text{Enter} &= \text{enter}(r, s, c) \mid f \triangleright \\
&\quad \text{pass } \lambda x p q. x(p()) \mid \text{onexit}(c) \mid r() [s()] \\
\text{Exit} &= \text{exit} \mid \text{onexit}(c) \triangleright \\
&\quad \text{pass } \lambda x p q. x(p()) \mid f [0] \mid q() \mid c() \\
\text{Route} &= \left(\begin{array}{l} \text{route}(x, y) \triangleright \\ \left[\begin{array}{l} \text{nil} = x \\ \text{exit}, \\ \text{pass } \lambda z p q. \\ x.\text{visit}(\lambda. z(y()) \mid p()) [q()] \end{array} \right] \end{array} \right)
\end{aligned}$$

Intuitively, each taxi contains a private lock (here the name `f`) indicating the taxi is free to hire. When a passenger enters the taxi, the lock is consumed, the frozen continuation is stored in the reference cell `onexit`, the route is unfrozen, and the sitting behaviour is spawned in the content of the taxi, using passivation. Since the route consists of a message containing a locality name and the rest of the route, the `Route` definition matches the locality name. If it is `nil`, then the

$\tau ::=$	Δ	type
	σ	value type
$\sigma ::=$	unit	value type
	α	unit type
	$\text{dom}(w)$	plain type variable
	$\sigma_1, \dots, \sigma_q$	name type
	$\sigma \rightarrow \tau$	tuple type
	$\langle \sigma \rangle_{\Delta}$	function type
	$\langle \sigma \rangle_{\Delta}^+$	plain resource
		sendable resource
$w ::=$		locality name variable
	a	locality name
	δ	name type variable
	\emptyset	no such locality
$\Delta ::=$		locality name multiset
	\emptyset	empty multiset
	ρ	multiset variable
	δ	name type variable
	a	locality name
	Δ, Δ	multiset union
$s ::=$	$\forall \tilde{\alpha} \tilde{\delta} \tilde{\rho}. \sigma$	type scheme

Figure 12: Types: Syntax

taxi has arrived, and the `exit` message is spawned. This message triggers the `(Exit)` definition, that consumes the `exit` message as well as the reference cell `onexit`, recreates the taxi as free to hire (spawning the lock `f`), and spawns the continuation as well as the content of the taxi outside. If the next destination is not `nil`, then the taxi migrates to this destination. We suppose every locality has a channel `visit` that simply spawns its argument. The frozen taxi is thus sent to the next stop, with the remaining of the route spawned in its controller.

3 The Type System

The routing rules for addressed messages in Figure 7 rely heavily on locality names and active localities. Locality names thus play the role of *addresses* in the M-calculus. However, to faithfully mirror the situation in current wide area networks, and to allow for an effective implementation of the calculus, one must ensure the determinacy of the final destination of a remote message by ensuring that no two active localities may bear the same name. Unfortunately, it is not possible to obtain this property by syntactic means only. In fact, even a simple type system will not do because of the higher-order features of the calculus. In presence of the passivation operator, one must indeed be careful of the effects of functions on locality names. For instance, if a resource `twice` is defined as `(twice $f \triangleright f() \mid f()$)`, then a passivation instruction of the form `pass $\lambda x p q. (\text{twice } \lambda. x(p())[q()])$` may lead to the illicit duplication of the passivated locality.

To enforce the unicity of active locality names, we introduce the following type system. The grammar for types is given in Figure 12. We remark that terms in the M-calculus are partitioned in two kinds: *processes* and *expressions*. This distinction is difficult to manifest by relying only on the syntax, mainly because of functional application (the result of an application may either be an expression or a process), but the intuition behind this partition is that processes may be put in

$$\begin{array}{lcl}
& \Delta \leq \Delta' & \Leftarrow \Delta \subseteq \Delta' \\
& \tilde{\sigma} \leq \tilde{\sigma}' & \Leftarrow (\sigma_i \leq \sigma'_i)^{i \in [1..n]} \\
\mathbf{unit} \leq \mathbf{unit} & \sigma \rightarrow \tau \leq \sigma' \rightarrow \tau' & \Leftarrow \sigma' \leq \sigma \text{ and } \tau \leq \tau' \\
\mathbf{dom}(w) \leq \mathbf{dom}(w) & \langle \sigma \rangle_{\Delta} \leq \langle \sigma' \rangle_{\Delta'} & \Leftarrow \sigma' \leq \sigma \text{ and } \Delta \leq \Delta' \\
\alpha \leq \alpha & \langle \sigma \rangle_{\Delta} \leq \sigma' \rightarrow \Delta' & \Leftarrow \sigma' \leq \sigma \text{ and } \Delta \leq \Delta' \\
\langle \sigma \rangle_{\Delta}^+ \leq \langle \sigma \rangle_{\Delta}^+ & \langle \sigma \rangle_{\Delta}^+ \leq \langle \sigma' \rangle_{\Delta'} & \Leftarrow \sigma' \leq \sigma \text{ and } \Delta \leq \Delta' \\
& \langle \sigma \rangle_{\Delta}^+ \leq \sigma' \rightarrow \Delta' & \Leftarrow \sigma' \leq \sigma \text{ and } \Delta \leq \Delta'
\end{array}$$

Figure 13: Subtyping relation

parallel, and include messages, localities, controller and content of localities, whereas expressions may reduce to values and include functions, tuples, and names. We formalize this partition by making a distinction among types τ between *process types* Δ and *value types* σ . Process types are multisets of locality names, representing an upper bound of the localities that may be or become active in the process. Value types represent the value the expression may eventually reduce to. They include function types $\sigma \rightarrow \tau$, tuple types $\tilde{\sigma}$, the \mathbf{unit} type, and types for names. Resource names have type $\langle \sigma \rangle_{\Delta}$ or $\langle \sigma \rangle_{\Delta}^+$ if they expect an argument of type σ (which may be a tuple) and if a message on this resource name leads to the creation of localities Δ . A resource name with the type $\langle \sigma \rangle_{\Delta}^+$ may be received and used for further input, as in the creation of a reaction rule $\langle \mathit{create} \ x \triangleright \langle x \rangle \triangleright a(\mathbf{0})[\mathbf{0}] \rangle$, where create may have the type $\langle \langle \mathbf{unit} \rangle_a^+ \rangle_{\emptyset}$. Locality names have type $\mathbf{dom}(w)$, where w may either be a locality name a , a name type variable δ , or the empty set \emptyset . Intuitively, a locality name a has singleton type $\mathbf{dom}(a)$, a variable x may have type $\mathbf{dom}(\delta)$, and no name may have type $\mathbf{dom}(\emptyset)$ (this last type is needed for technical reasons). Name type variables δ reflect term variables x that may be instantiated to locality names, as in $\langle \mathit{create}(x) \triangleright x(\mathbf{0})[\mathbf{0}] \rangle$ where create may be given the type $\forall \delta. \langle \mathbf{dom}(\delta) \rangle_{\delta}$.

Multisets Δ include locality names a , name type variables δ , multiset variables ρ , the empty multiset \emptyset , and unions of multisets Δ, Δ' . The basic intuition to guarantee the unicity of names of active localities in a process P is to type a process P with a multiset Δ . If this multiset happens to be a set, then we prove that every active locality bears a unique name.

We use $\forall \tilde{\alpha} \tilde{\delta} \tilde{\rho}. \sigma$ to denote a type scheme where plain type variables $\tilde{\alpha}$, name type variable $\tilde{\delta}$ and multiset variables $\tilde{\rho}$ are generalized. We use β to range indifferently over these different type variables.

In what follows, we consider an extended syntax for the M-calculus, where new resource names are required to be annotated by their type scheme, in order to specify whether the resource has a plain resource type (which may be polymorphic) or if it has a sendable resource type (which cannot be polymorphic for safety reasons).

The intersection operation between multisets, \cap , is the standard intersection on multisets (taking the smallest number of occurrences in both multisets). The inclusion relation \subseteq between multisets is also the standard one. By $\Delta - \Delta'$, we denote the multiset which is composed of the elements of Δ (locality names or multiset variables) after removing each element of Δ' . For instance $\rho, \rho, a, b, b - \rho, a, a, b = \rho, b$.

We define in figure 13 a subtyping relation \leq (where $\tilde{\sigma}$ and $\tilde{\sigma}'$ are tuples of the same size n).

The intuition behind the subtyping relation is that it is safe (with regard to the unicity of locality names) to replace a process that includes more active localities with a process that includes fewer active localities. It is also safe to replace a function by a resource name if the types agree (a resource name may be used to send a message, by functional application, but it can also be used to create an addressed resource name). It is finally also safe to replace a plain resource name by a sendable resource name (sendable resource names may be used for message sending, but they can also be used to instantiate variables that are defined names of a Join pattern).

We remark that sendable resource types have no subtype, except themselves. This is necessary for type safety since, unlike [25], we do not distinguish between input and output types.

$$\begin{aligned}
a, \Delta \sqcup a, \Delta' &= a, (\Delta \sqcup \Delta') \\
\rho, \Delta \sqcup \rho, \Delta' &= \rho, (\Delta \sqcup \Delta') \\
\delta, \Delta \sqcup \delta, \Delta' &= \delta, (\Delta \sqcup \Delta') \\
\Delta \sqcup \Delta' &= \Delta, \Delta' \text{ if } \Delta \cap \Delta' = \emptyset \\
\mathbf{unit} \sqcup \mathbf{unit} &= \mathbf{unit} \\
\mathbf{dom}(w) \sqcup \mathbf{dom}(w) &= \mathbf{dom}(w) \\
\alpha \sqcup \alpha &= \alpha \\
(\sigma_1, \dots, \sigma_n) \sqcup (\sigma'_1, \dots, \sigma'_n) &= (\sigma_1 \sqcup \sigma'_1, \dots, \sigma_n \sqcup \sigma'_n) \\
\sigma \rightarrow \tau \sqcup \sigma' \rightarrow \tau' &= (\sigma \sqcap \sigma') \rightarrow (\tau \sqcup \tau') \\
\langle \sigma \rangle_{\Delta} \sqcup \langle \sigma' \rangle_{\Delta'} &= \langle \sigma \sqcap \sigma' \rangle_{\Delta \sqcup \Delta'} \\
\langle \sigma \rangle_{\Delta} \sqcup \sigma' \rightarrow \Delta' &= (\sigma \sqcap \sigma') \rightarrow (\Delta \sqcup \Delta') \\
\langle \sigma \rangle_{\Delta}^+ \sqcup \langle \sigma' \rangle_{\Delta'}^+ &= \langle \sigma \rangle_{\Delta}^+ \\
\langle \sigma \rangle_{\Delta}^+ \sqcup \langle \sigma' \rangle_{\Delta'}^+ &= \langle \sigma \sqcap \sigma' \rangle_{\Delta \sqcup \Delta'}^+ \text{ if } \sigma \neq \sigma' \text{ or } \Delta \neq \Delta' \\
\langle \sigma \rangle_{\Delta}^+ \sqcup \langle \sigma' \rangle_{\Delta'}^+ &= \langle \sigma \sqcap \sigma' \rangle_{\Delta \sqcup \Delta'}^+ \\
\langle \sigma \rangle_{\Delta}^+ \sqcup \sigma' \rightarrow \Delta' &= (\sigma \sqcap \sigma') \rightarrow (\Delta \sqcup \Delta')
\end{aligned}$$

Figure 14: Definition of \sqcup

$$\begin{aligned}
a, \Delta \sqcap a, \Delta' &= a, (\Delta \sqcap \Delta') \\
\rho, \Delta \sqcap \rho, \Delta' &= \rho, (\Delta \sqcap \Delta') \\
\delta, \Delta \sqcap \delta, \Delta' &= \delta, (\Delta \sqcap \Delta') \\
\Delta \sqcap \Delta' &= \emptyset \text{ if } \Delta \cap \Delta' = \emptyset \\
\mathbf{unit} \sqcap \mathbf{unit} &= \mathbf{unit} \\
\mathbf{dom}(w) \sqcap \mathbf{dom}(w) &= \mathbf{dom}(w) \\
\alpha \sqcap \alpha &= \alpha \\
(\sigma_1, \dots, \sigma_n) \sqcap (\sigma'_1, \dots, \sigma'_n) &= (\sigma_1 \sqcap \sigma'_1, \dots, \sigma_n \sqcap \sigma'_n) \\
\sigma \rightarrow \tau \sqcap \sigma' \rightarrow \tau' &= (\sigma \sqcup \sigma') \rightarrow (\tau \sqcap \tau') \\
\langle \sigma \rangle_{\Delta} \sqcap \langle \sigma' \rangle_{\Delta'} &= \langle \sigma \sqcup \sigma' \rangle_{\Delta \sqcap \Delta'} \\
\langle \sigma \rangle_{\Delta} \sqcap \sigma' \rightarrow \Delta' &= \langle \sigma \sqcup \sigma' \rangle_{\Delta \sqcap \Delta'} \\
\langle \sigma \rangle_{\Delta}^+ \sqcap \sigma' &= \langle \sigma \rangle_{\Delta}^+ \text{ if } \langle \sigma \rangle_{\Delta}^+ \leq \sigma'
\end{aligned}$$

Figure 15: Definition of \sqcap

$$\begin{array}{ll}
fsv(\emptyset) = \emptyset & fvw(\emptyset) = \emptyset \\
fsv(\rho) = \{\rho\} & fvw(\rho) = \emptyset \\
fsv(\delta) = \emptyset & fvw(\delta) = \{\delta\} \\
fsv(a) = \emptyset & fvw(a) = \emptyset \\
fsv(\Delta, \Delta') = fsv(\Delta) \cup fsv(\Delta') & fvw(\Delta, \Delta') = fvw(\Delta) \cup fvw(\Delta')
\end{array}$$

Figure 16: Definition of $fsv()$ and $fwv()$

$$\begin{array}{l}
C ::= \cdot : \tau \mid \nu r : s.C \mid \nu a.C \mid \lambda x.C \mid \mathbf{a}(C)[Q] \\
\mid \mathbf{a}(P)[C] \mid (C \mid P) \mid (P \mid C) \mid \mathbf{pass} C \mid (CQ) \\
\mid (PC) \mid ([\mu = C]P, Q) \mid ([\mu = V]C, P) \\
\mid ([\mu = V]P, C) \mid \langle J \triangleright C \rangle \mid (P_1, \dots, C, \dots, P_q)
\end{array}$$

Figure 17: Typed Contexts

We define the symmetric \sqcup and \sqcap operators on types in figures 14 and 15 (other possible cases not listed are undefined). $\Delta_1 \sqcup \Delta_2$ is defined as the multiset where the multiplicity of each name a is taken to be the *max* of the multiplicities in Δ_1 and Δ_2 , and $\Delta_1 \sqcap \Delta_2$ is defined as the multiset where the multiplicity of each name a is taken to be the *min* of the multiplicities in Δ_1 and Δ_2 (it is the usual intersection on multisets).

We use Γ and its decorated variants to denote type environments, i.e. finite mappings between names and type schemes. We define the set of free plain type variables $ftv()$ as usual. We define the set of free multiset variables $fsv()$ and the set of free name type variables $fwv()$ in figure 16.

We let the set of free variables $fv()$ be the union of $ftv()$, $fsv()$, and $fwv()$. Type judgments take the following form, where C is an M-context extended with a typed hole, as defined in Figure 17: $\Gamma \vdash C : \tau$ and $\Gamma \vdash P : \tau$.

We write $fun(\sigma; \tau)$ for the types of all values that may be used as functions, that is for either $\sigma \rightarrow \tau$, $\langle \sigma \rangle_\tau$, or $\langle \sigma \rangle_\tau^+$ (in the latter two cases, τ is necessarily a process type Δ). We write $chan(\sigma; \Delta)$ for channel types (either $\langle \sigma \rangle_\Delta$ or $\langle \sigma \rangle_\Delta^+$).

In order to correctly type the *i* and *o* resources, we consider only type environments with the following associations:

$$\begin{array}{l}
i : \forall \alpha \delta \rho. \langle \text{dom}(\delta), \langle \alpha \rangle_\rho, \alpha \rangle_\rho \\
o : \forall \alpha \delta \rho. \langle \text{dom}(\delta), \langle \alpha \rangle_\rho, \alpha \rangle_\rho
\end{array}$$

Both the *i* and *o* resources expect a locality name (the destination of the message), the targeted resource expecting an argument of type α and creating localities ρ , and an argument of type α . A message on such a channel potentially creates localities ρ .

The type system is defined by the rules in Figure 18. They make use of the *Inst* operator, that takes a type scheme and returns a type where the generalized plain type variables, multiset variables, and name type variables have been instantiated to types, multisets, and locality names or name type variables respectively.

Typing rule JOIN may seem complex but is the usual typing rule for Join patterns: the guarded process is typed in an environment extended with the formal parameters, and the result is checked to create fewer localities than advertised by the resource types. Every defined resource name that is a variable is checked to have a sendable resource type in the environment. The additional hypotheses check that the type schemes associated with the resources are consistent with the typing environment, following the usual rules of the Join calculus: no generalized variable may occur free in the environment nor be shared by two resources in a Join pattern.

$$\begin{array}{c}
\frac{u : \forall \tilde{\beta}. \sigma \in \Gamma \quad \sigma\theta = \text{Inst}(\forall \tilde{\beta}. \sigma) \quad \text{fn}(\text{ran}(\theta)) \subseteq \text{dom}(\Gamma)}{\Gamma \vdash u : \sigma\theta} \text{ [NAME]} \qquad \frac{}{\Gamma \vdash \mathbf{0} : \emptyset} \text{ [NIL]} \\
\\
\frac{}{\Gamma \vdash () : \text{unit}} \text{ [VOID]} \qquad \frac{}{\Gamma \vdash (\cdot : \tau) : \tau} \text{ [PROC.HOLE]} \qquad \frac{\Gamma \vdash \mathbf{a} : \text{dom}(w) \quad \Gamma \vdash \mathbf{r} : \text{chan}(\sigma; \Delta)}{\Gamma \vdash \mathbf{a.r} : \sigma \rightarrow \Delta} \text{ [ADDR]} \\
\\
\frac{\Gamma + x : \sigma \vdash P : \tau \quad x \notin \text{dom}(\Gamma) \quad \text{fn}(\sigma) \subseteq \text{dom}(\Gamma)}{\Gamma \vdash \lambda x. P : \sigma \rightarrow \tau} \text{ [FUN]} \qquad \frac{(\Gamma \vdash P_i : \sigma_i)^{i \in [1..q]}}{\Gamma \vdash P_1, \dots, P_q : (\sigma_1, \dots, \sigma_q)} \text{ [TUPLE]} \\
\\
\frac{\Gamma \vdash \mathbf{a} : \text{dom}(w) \quad \Gamma \vdash P : \Delta_1 \quad \Gamma \vdash Q : \Delta_2}{\Gamma \vdash \mathbf{a}(P)[Q] : w, \Delta_1, \Delta_2} \text{ [DOM]} \qquad \frac{\Gamma \vdash P : \Delta_1 \quad \Gamma \vdash Q : \Delta_2}{\Gamma \vdash P \mid Q : \Delta_1, \Delta_2} \text{ [PAR]} \\
\\
\frac{\Gamma + r : \forall \tilde{\beta}. \langle \sigma \rangle_{\Delta} \vdash P : \Delta_1 \quad r \notin \text{dom}(\Gamma) \quad \text{fv}(\forall \tilde{\beta}. \langle \sigma \rangle_{\Delta}) = \emptyset \quad \text{fn}(\forall \tilde{\beta}. \langle \sigma \rangle_{\Delta}) \subseteq \text{dom}(\Gamma)}{\Gamma \vdash \nu r : \forall \tilde{\beta}. \langle \sigma \rangle_{\Delta}. P : \Delta_1} \text{ [NU.RES.1]} \\
\\
\frac{\Gamma + r : \langle \sigma \rangle_{\Delta}^+ \vdash P : \Delta_1 \quad r \notin \text{dom}(\Gamma) \quad \text{fv}(\langle \sigma \rangle_{\Delta}^+) = \emptyset \quad \text{fn}(\langle \sigma \rangle_{\Delta}^+) \subseteq \text{dom}(\Gamma)}{\Gamma \vdash \nu r : \langle \sigma \rangle_{\Delta}^+. P : \Delta_1} \text{ [NU.RES.2]} \\
\\
\frac{\Gamma + a : \text{dom}(a) \vdash P : \Delta \quad a \notin \text{fn}(\Gamma) \quad a \notin (\Delta - a)}{\Gamma \vdash \nu a. P : \Delta - a} \text{ [NU.DOM]} \\
\\
\frac{\Gamma \vdash V : \text{dom}(\delta) \rightarrow (\text{unit} \rightarrow \Delta_1) \rightarrow \text{fun}(\text{unit} \rightarrow \Delta_2; \Delta) \quad \rho_1 \neq \rho_2 \quad \rho_1 \in \Delta_1 \quad \rho_2 \in \Delta_2 \quad \delta, \rho_1, \rho_2 \notin \text{fv}(\Gamma) \cup (\Delta - (\delta, \rho_1, \rho_2))}{\Gamma \vdash \text{pass } V : \Delta - (\delta, \rho_1, \rho_2)} \text{ [PASS]} \\
\\
\frac{\Gamma \vdash P : \text{fun}(\sigma; \tau) \quad \sigma' \leq \sigma \quad \Gamma \vdash Q : \sigma'}{\Gamma \vdash PQ : \tau} \text{ [APP]} \qquad \frac{}{\Gamma \vdash _ : \text{dom}(\emptyset)} \text{ [WILD]} \\
\\
\frac{\Gamma \vdash \mu : \sigma \quad \sigma = \text{dom}(w) \text{ or } \sigma = \text{chan}(\sigma_{\mu}; \Delta) \quad \Gamma \vdash V : \tau \quad \Gamma \vdash P : \tau_1 \quad \Gamma \vdash Q : \tau_2}{\Gamma \vdash ([\mu = V]P, Q) : \tau_1 \sqcup \tau_2} \text{ [TEST]} \\
\\
\frac{(\mathbf{r}_i \text{ is not a variable and } r_i : s_i = \forall \tilde{\beta}_i. \langle \tilde{\sigma}_i \rangle_{\Delta_i} \in \Gamma, \text{ or } \mathbf{r}_i : s_i = \langle \tilde{\sigma}_i \rangle_{\Delta_i}^+ \in \Gamma)^{i \in [1..n]} \quad \Delta' \leq \Delta_1, \dots, \Delta_n \quad (\tilde{x}_i)^i \cap \text{dom}(\Gamma) = \emptyset \quad \Gamma + \tilde{x}_1 : \tilde{\sigma}_1 + \dots + \tilde{x}_n : \tilde{\sigma}_n \vdash P : \Delta' \quad \forall i \in [1..n], \tilde{\beta}_i \cap \text{fv}(\Gamma) = \emptyset \quad \forall i, j \in [1..n]^2, i \neq j \implies \tilde{\beta}_i \cap \tilde{\beta}_j = \emptyset}{\Gamma \vdash \langle \mathbf{r}_1 \tilde{x}_1 \mid \dots \mid \mathbf{r}_n \tilde{x}_n \triangleright P \rangle : \emptyset} \text{ [JOIN]}
\end{array}$$

Figure 18: Typing rules

In rule PASS, the passivation function is checked to have a type that is a subtype of a function expecting a locality name and two thunks. The name type variable δ in the locality name type represents the name of the passivated locality. Each thunk type includes a multiset variable ρ_1 and ρ_2 , respectively, representing the active localities of each thunk. These three variables are intuitively generalized by checking that they do not occur in the typing environment (they will respectively be substituted by the name of the passivated locality and by the active localities of the controller and content, that are unknown when typing the primitive). The type of the whole passivation construct is simply the type of the result of the passivation function, removing the name of the passivated locality and the multiset variables since the locality is passivated.

Rule TEST checks that if the pattern uses variables, these have either resource or locality type, insuring they shall only be instantiated with resource or locality names. This rule relies on rule WILD that only says that a wildcard is well-typed, using the ad-hoc type $\text{dom}(\emptyset)$.

The soundness of our type system is characterized by the following definitions and theorems.

Definition 3.1 (Well-formed typing environment) *A typing environment Γ is well-formed if and only if:*

1. Γ only contains associations of the form $r : \forall \tilde{\beta}. \langle \sigma \rangle_{\Delta}$, $r : \langle \sigma \rangle_{\Delta}^+$, and $a : \text{dom}(a)$;
2. we have $\text{fn}(\Gamma) = \text{dom}(\Gamma)$.

Lemma 3.2 (Subject reduction for \equiv) *Let $\Gamma \vdash P : \tau$ be a typing derivation with Γ well-formed. If $P \equiv P'$, then there exists a typing derivation $\Gamma \vdash P' : \tau$.*

Theorem 1 (Subject reduction) *Let $\Gamma \vdash P : \tau$ be a typing derivation with Γ well-formed. If $P \rightarrow P'$, then there exists a type τ' such that $\tau' \leq \tau$ and $\Gamma \vdash P' : \tau'$.*

Definition 3.3 (Failure) *A locality $a(P')[Q']$ is said to be free and active in P if it is active in P and if it is not under a scope restriction for a .*

A process P has failed if and only if there is some process P' active in P containing at least two free and active localities bearing the same name.

Theorem 2 (Progress) *Let $\Gamma \vdash P : \Delta$ be a typing derivation with Γ well-formed. If Δ is a set, then the process P has not failed.*

Most of the complexity of the subject reduction proof is alleviated by distinguishing name variables x from name type variables δ , greatly simplifying the substitution lemma. Complete proofs are developed in appendix A.

Our notion of progress deals only with the unicity of active locality names. Since our type system is very close to type systems for the λ -calculus and the Join calculus, this progress property can be easily extended to more usual guarantees.

We now discuss the power and limits of our type system.

Linearity Since our type system aims at enforcing the linearity of names of active localities, we describe how this feature interacts with functions (or resources) that are not linear. First of all, our system guarantees the linearity of names of localities that are *active* (as described in theorem 2). Relaxing the linearity constraint for localities that are not active yields a much simpler type system (e.g. our system allows the typing of $\lambda x.a(\mathbf{0})[\mathbf{0}] \mid a(\mathbf{0})[\mathbf{0}]$ as long as this function is not applied). We also require linearity of a newly introduced locality name in the process under the scope restriction (rule NU.DOM, since the introduced name should not occur in the type). We recall that the type of a process is a conservative approximation of the active localities it may contain. In the following, we only consider processes that are active.

Consider for instance the function $\lambda x.x() \mid x()$. This function takes a frozen process and runs it twice. It is perfectly reasonable to apply it to a frozen process that does not release localities with

free names, as in: $(\lambda x.x() \mid x())\lambda.\mathbf{0}$ or $(\lambda x.x() \mid x())\lambda.\nu a.a(\mathbf{0})[\mathbf{0}]$. In these cases, the function has type $(\mathbf{unit} \rightarrow \emptyset) \rightarrow \emptyset$. The type system would reject applying this function to the thunk $\lambda.a(\mathbf{0})[\mathbf{0}]$.

A function $\lambda x.x()$, that only runs a process, can be given the type $(\mathbf{unit} \rightarrow \Delta) \rightarrow \Delta$. Since functions never have polymorphic types, one can instead consider a resource run defined as $\langle \text{run}(x) \triangleright x() \rangle$. Such a resource can be given the type $\forall \rho. \langle \mathbf{unit} \rightarrow \rho \rangle_\rho$: it simply recreates the active localities frozen in its argument. The check that these localities do not interfere with currently active localities is done when typing a message on `run`. We remark that our use of multiset variables is very similar to row variables described in [16].

Subtyping Subtyping is explicitly used in typing rule `APP`. Returning to the example of the `run` resource above, this resource can receive a frozen process as argument, as in $(\text{run } \lambda.a.(\mathbf{0})[\mathbf{0}])$. It can also receive a resource name, as in $(\text{run } \text{create}_a)$ with $\langle \text{create}_a() \triangleright a(\mathbf{0})[\mathbf{0}] \rangle$, since $\langle \mathbf{unit} \rangle_a$ is a subtype of $\mathbf{unit} \rightarrow a$.

The subtyping used in typing rule `JOIN` gives some flexibility when defining a resource, since the type of the resource gives an upper bound on the localities that may be created. For instance, the special channel `i` has type $\forall \alpha \delta \rho. \langle \text{dom}(\delta), \langle \alpha \rangle_\rho, \alpha \rangle_\rho$. Typical definitions for this channel are $\langle i(d, r, \text{args}) \triangleright d.r \text{ args} \rangle$, which simply sends the message, and $\langle i(d, r, \text{args}) \triangleright \mathbf{0} \rangle$, which discards it. The typing of the second definition relies on the fact that this definition of `i` does not create all the localities it is allowed to.

Dependent Types and Polymorphism Our type system does not allow dependent types, but simulates them using polymorphism and *name type variables* (type variables that represent locality names), since locality names may occur in types. For instance the resource `new` in the definition $\langle \text{new}(x) \triangleright x(\mathbf{0})[\mathbf{0}] \rangle$ may be given the polymorphic type $\forall \delta. \langle \text{dom}(\delta) \rangle_\delta$: it expects any locality name and creates a locality with this name. We remark that our solution is less powerful than dependent types, since it does not allow the typing of $(\lambda f.f a \mid f b)(\lambda x.x(\mathbf{0})[\mathbf{0}])$. However, polymorphism is powerful enough to let us type definitions of the form: $\langle \text{create}(x, p, q) \triangleright x(p())[q()] \rangle$, with `create`: $\forall \delta \rho_1 \rho_2. \langle \text{dom}(\delta), \mathbf{unit} \rightarrow \rho_1, \mathbf{unit} \rightarrow \rho_2 \rangle_{\delta, \rho_1, \rho_2}$.

Passivation One limitation of our type system comes from the passivation operator. Consider for instance the process: $a(\text{pass } \lambda xpq.x(\mathbf{0})[b(\mathbf{0})[\mathbf{0}]]) [b(\mathbf{0})[\mathbf{0}]]$. The `pass` operator freezes locality `a` and respawns it, discarding its controller and content, but adds a locality `b` in its content. Since the locality `b` that is added is first discarded, one would expect this process to be well typed. This is not the case because locality `b` is active, and may be sent somewhere else (see for instance the migration example in section 2). This is why the process `pass` $\lambda xpq.x(\mathbf{0})[b(\mathbf{0})[\mathbf{0}]]$ has type `b`: this process creates an active locality `b`, independently of the controller and content that are passivated. On the other hand, the process `pass` $\lambda xpq.x(p())[q()]$ has type \emptyset (it only recreates a locality that has been passivated) and the following process is well typed: $a(\text{pass } \lambda xpq.x(p())[q()]) [b(\mathbf{0})[\mathbf{0}]]$.

Example As a small example, we remark that the taxi example we describe at the end of section 2 is well typed with the following bindings:

$$\begin{aligned}
\text{enter} & : \forall \rho_S, \rho_C. \langle \mathbf{unit} \rightarrow \emptyset, \mathbf{unit} \rightarrow \rho_S, \mathbf{unit} \rightarrow \rho_C \rangle_{\rho_S, \rho_C} \\
\text{exit} & : \langle \mathbf{unit} \rangle_\emptyset \\
\text{onexit} & : \forall \rho_C. \langle \mathbf{unit} \rightarrow \rho_C \rangle_{\rho_C} \\
\text{route} & : \forall \delta. \langle \text{dom}(\delta), \mathbf{unit} \rightarrow \emptyset \rangle_\emptyset \\
\text{visit} & : \forall \rho. \langle \mathbf{unit} \rightarrow \rho \rangle_\rho
\end{aligned}$$

4 Simulating the Join calculus

In this section, we show that we can simulate the Distributed Join Calculus in a straightforward manner, thus retaining all its expressivity. However, we first check that the M-Calculus addresses the limitation of the Join calculus described in the introduction.

The lack of control over communication and mobility is directly addressed by the possibility to program *controllers* for each locality to filter incoming and outgoing messages. These controllers are normal M-Calculus processes, they can be arbitrarily complex.

Dynamic binding, or more precisely locality based binding, is provided by *addressed messages*, which explicitly include the name of the locality hosting the targeted resource, and by the possibility to define *resources* at several localities. A simulation of the Dynamic Join calculus [20] in the M-Calculus presented in section 4.3 illustrates how to use addressed messages to get transparent locality based binding.

Finally, programmable controllers and the *passivation* operator let the programmer define localities with different semantics within the same program.

4.1 Simulating the distributed Join calculus

Since the M-calculus is a direct offspring of the Join calculus, a translation of a Join calculus process into an M-calculus process is relatively straightforward. Such a translation is interesting to present, however, because it illustrates the versatility of programmable localities. In the following, we only consider Join processes with no free names. We proceed in three steps, in order to account for the fact that resource names are introduced with their types in the M-calculus, and that message routing requires the messages to be annotated with address information.

The first step consists in typing the Join calculus process, and annotating every local Join calculus definition with the channel names (as well as their type schemes) and location names it defines. We write $\text{def } (D; n_1 : s_1, n_q : s_q, a_1, \dots, a_r) \text{ in } \dots$ the result of the first translation step of $\text{def } D \text{ in } P$ if $dn(D) = \{n_1, \dots, n_q\} \cup \{a_1, \dots, a_r\}$ and if the channel names n_i have type schemes s_i after generalization. By definition of the types of the Join calculus, the s_i are of the form $\forall \tilde{\alpha}. \langle \tilde{\tau} \rangle$. We suppose that these type schemes have no free type variables, and we transform them immediately in M-calculus type schemes, writing $\langle \rangle_\emptyset$ instead of $\langle \rangle$.

The second translation step consists in annotating every channel name that is not a variable with the locality where its definitions reside. To do this, we rely on the fact that in the Join calculus, the only locality in which a definition may eventually be dissolved is the syntactically enclosing locality. This property is a direct consequence of the definition of migration: the only way to have processes migrate in the Join calculus is by locality migration. Thus definitions cannot be separated from the locality where they syntactically occur. We do not formally present the algorithm used in this second step, which is roughly of the following form: for every channel name that is not in a Join pattern, find the enclosing def binder for this name, then find the name of the syntactically enclosing locality, and prepend the locality name to the channel name.

The third step is the translation to the M-calculus itself. We represent a location of the Join calculus $a[\cdot]$ by a locality $a(PJ)[\cdot]$ with:

$$\begin{aligned} PJ &= Fwd \mid \langle \text{add } f \triangleright Add(f) \rangle \mid \langle \text{go } (b, \kappa) \triangleright Send(b, \kappa) \rangle \\ Add(f) &= \text{pass } \lambda x p q. x(p())[q()] \mid f() \\ Send(b, \kappa) &= \text{pass } \lambda x p q. (b.\text{add } \lambda x. x(p())[q()] \mid \kappa()) \end{aligned}$$

We may type the resource add with the type $\forall \rho. \langle \text{unit} \rightarrow \rho \rangle_\rho$ that expects a thunk and frees it, and the resource go with the type $\forall \delta \rho. \langle \text{dom}(\delta), \text{unit} \rightarrow \rho \rangle_\rho$, that expects a locality name and a frozen continuation that it will eventually spawn.

Writing $\llbracket \cdot \rrbracket$ for the translation operator, we have:

$$\begin{aligned}
\llbracket b.n \rrbracket &= b.n & \llbracket b \rrbracket &= b \\
\llbracket x \rrbracket &= x & \llbracket \mathbf{0} \rrbracket &= \mathbf{0} \\
\llbracket P \mid Q \rrbracket &= \llbracket P \rrbracket \mid \llbracket Q \rrbracket & \llbracket \top \rrbracket &= \mathbf{0} \\
\llbracket \mathbf{go} \ b; P \rrbracket &= \mathbf{go}(b, \lambda. \llbracket P \rrbracket) & \llbracket D, D' \rrbracket &= \llbracket D \rrbracket \mid \llbracket D' \rrbracket \\
\llbracket b[D : P] \rrbracket &= b(PJ)(\llbracket D \rrbracket \mid \llbracket P \rrbracket) \\
\llbracket m\langle n_1, \dots, n_q \rangle \rrbracket &= \llbracket m \rrbracket(\llbracket n_1 \rrbracket, \dots, \llbracket n_q \rrbracket) \\
\llbracket n_1\langle \widehat{x}_1 \rangle \mid \dots \mid n_q\langle \widehat{x}_q \rangle \triangleright P \rrbracket &= \langle n_1\widehat{x}_1 \mid \dots \mid n_q\widehat{x}_q \triangleright \llbracket P \rrbracket \rangle \\
\llbracket \mathbf{def} \ (D; n_1 : s_1, \dots, n_q : s_q, a_1, \dots, a_r) \ \mathbf{in} \ P \rrbracket &= \\
&\nu n_1 : s_1, \dots, n_q : s_q. \nu a_1, \dots, a_r. \llbracket D \rrbracket \mid \llbracket P \rrbracket
\end{aligned}$$

Note that our translation is not entirely faithful with respect to migration, since in the distributed Join calculus migration does not occur if the target locality is a sublocality of the moving one. In order to detect these cases, we would need a more complex translation.

4.2 Simulating the distributed Join calculus with failures

We now extend the previous translation to the Join calculus with failures. To do this, we only need to change the translation of a locality, replacing the PJ controller by the following $PJF(a)$ controller:

$$\begin{aligned}
PJF &= Fwd \\
&\mid \langle \mathbf{add} \ f \triangleright Add(f) \rangle \\
&\mid \langle \mathbf{go} \ (b, \kappa) \triangleright Send(b, \kappa) \rangle \\
&\mid \langle \mathbf{halt} \triangleright Halt \rangle \\
&\mid \langle \mathbf{ping} \ (y, n) \triangleright y() \rangle \\
Add(f) &= \mathbf{pass} \ \lambda x p q. x(p())[q() \mid f()] \\
Send(b, \kappa) &= \mathbf{pass} \ \lambda x p q. (b.\mathbf{add} \ \lambda x(p())[q() \mid \kappa()]) \\
Halt &= \mathbf{pass} \ \lambda x p q. x \left(\begin{array}{l} \langle \mathbf{ping} \ (y, n) \triangleright n() \rangle \\ \mid \langle \mathbf{add} \ f \triangleright Add(f) \rangle \\ \mid \langle i \ (d, m, v) \triangleright P(m, v) \rangle \\ \mid \langle o \ (d, m, v) = \mathbf{0} \rangle \end{array} \right) [q()] \\
P(m, v) &= ([\mathbf{ping} = m]m v, ([\mathbf{add} = m]m v, \mathbf{0}))
\end{aligned}$$

Thus, when a locality has failed, it prevents all outgoing messages from leaving the failed locality: no sublocality may send a message outside. Similarly, with respect to incoming messages, a failed locality only accepts messages sent on \mathbf{ping} or on \mathbf{add} for itself or its sublocalities. Messages on \mathbf{ping} are emitted locally, and will be subsequently reduced by the new definition for \mathbf{ping} , that will answer saying the sublocality has failed. Messages on \mathbf{add} will add the migrating location to the current locality, thereby cutting it from the rest of the world.

We remark that the sublocalities of the failing one are still active, but they cannot communicate with the outside world. This translation strongly relies on the interception of routed messages by controller processes.

4.3 Simulating the dynamic Join calculus

We now extend the translation of section 4.1 with a form of locality based dynamic binding inspired by [20]. However, instead of sending a dynamic message directly to the closest enclosing definition, we choose to propagate it in a more local, step by step manner. More precisely, a message on a dynamic channel that is not defined locally is simply sent to the enclosing locality. As in section 4.1, we translate a static channel name into an addressed resource name. A dynamic channel name is translated to a local resource name. Every imported dynamic name is associated to a definition sending a message on this name to the enclosing locality. We suppose that static and dynamic names are disjoint, and we suppose as well that imported and locally defined dynamic names are disjoint.

As in the previous translations, we proceed in three steps: annotating the term using typing information, resolution of static channel names to addressed names, and translation. The typing annotation phase is identical to the previous translation, it annotates every definition $\text{def } D \text{ in } P$ with the set of static channel names it defines and their type schemes, and the set of location names it defines. The second step is also identical, addressing every static name with the location where it is defined.

As concerns the third step, the translation step, we first remark that the translation is parameterized by a location name: the name of the current enclosing location. We detail only the rules that are modified or added:

$$\begin{aligned} \llbracket \mathbf{d} \rrbracket^a &= \mathbf{d} \\ \llbracket b[D : P]^{\Delta, I} \rrbracket^a &= b(PJD(b, a, I)) \llbracket [D]^b \mid [P]^b \rrbracket \\ \llbracket \nu \mathbf{d} : \langle \tilde{\tau} \rangle_{\Delta}^+ . P \rrbracket^a &= \nu \mathbf{d} : \langle \tilde{\tau} \rangle_{\emptyset}^+ . \llbracket P \rrbracket^a \end{aligned}$$

In order to forward dynamic messages on imported names to the enclosing locality, we define the process:

$$Dyn(\{\mathbf{d}_1, \dots, \mathbf{d}_q\}) = \langle \mathbf{d}_1 x \triangleright \text{up}(\mathbf{d}_1, x) \rangle, \dots \langle \mathbf{d}_q x \triangleright \text{up}(\mathbf{d}_q, x) \rangle$$

We translate a location $a[\cdot]^{\Delta, I}$ by the locality $a(PJD(a, b, I))[\cdot]$ if b is the initial enclosing locality, with $PJD(a, b, I)$ being:

$$\begin{aligned} PJD(a, b, I) &= Fwd \mid \langle \text{add } f \triangleright Add(a, f) \rangle \mid \langle \text{father}(c) \mid \text{go}(b, \kappa) \triangleright Send(a, b, \kappa) \rangle \\ &\quad \mid \langle \text{dyn}(m, args) \triangleright m \ args \rangle \\ &\quad \mid \langle \text{father}(c) \mid \text{up}(m, args) \triangleright \text{father}(c) \mid c.\text{dyn}(m, args) \rangle \\ &\quad \mid Dyn(I) \mid \text{father}(b) \\ Add(a, f) &= \text{pass } \lambda x p q. x(p()) [q()] \mid f() \\ Send(a, b, \kappa) &= \text{pass } \lambda x p q. (b.\text{add } \lambda x. x(p()) \mid \text{father}(b)) [q()] \mid \kappa() \end{aligned}$$

Let us now study how a message $\mathbf{d}(V)$ on dynamic channel \mathbf{d} is handled. If this dynamic channel is defined locally (the name is not in I), then there is a local definition for this channel, and the message may trigger it (if all other messages required by the pattern are there, of course). Otherwise, if the message is imported, then there is a definition in Dyn for this channel name that sends a message on up containing the dynamic channel name \mathbf{d} and the arguments V of the message. This message on up synchronizes with the message on father that contains the name of the current father, for instance b , and a message on $b.\text{dyn}(\mathbf{d}, V)$ is spawned. When this message reaches b , the initial message $\mathbf{d}(V)$ is spawned, and a new cycle may start. Thus a dynamic channel name is simply a local name whose destination is resolved step by step, using explicit user-controlled routing.

We may type the channels up and dyn with the type scheme $\forall \alpha \rho. \langle \langle \alpha \rangle_{\rho}, \alpha \rangle_{\rho}$.

We remark that this translation is not faithful for two reasons. First of all, as in the other translations of the distributed Join calculus, we do not detect circular migrations. Second of all, we propagate dynamic messages in a step by step manner, and not directly to their closest enclosing definition. This is because the dynamic Join calculus relies on the notion of *frozen location* and their unfolding to correctly compute the lookup function (we remark that the circular migration detection also relies on the notion of frozen location). We could simulate such a behavior if the spawning of frozen localities in the M-calculus could be done in a step by step manner (in the current calculus, the spawning of a frozen process is atomic).

5 Related work

Several distributed process calculi have been proposed in the recent years, but they all have shortcomings as distributed *programming* models:

- Ambient calculi, such as the original Mobile Ambients [4] and the subsequent variants (Safe Ambients [10], Safe Ambients with Passwords [13], Boxed Ambients [2], Controlled Ambients [22]), provide a simple model of hierarchical localities with fine-grained control over locality moves and communications, but their basic mobility primitives (the `in` and `out` capabilities) require a 3-party atomic handshake which makes them costly to implement in a distributed setting as illustrated by the implementation of Mobile Ambients in the Join calculus [8].¹
- Higher-order process calculi such as as Facile/CHOCS and $D\pi\lambda$ [25] model process mobility via higher-order communication and remote process execution, but lack an explicit notion of locality to account for potential failures or to provide a basis for access control. Furthermore, they do not allow for a running process to be migrated to a different locality, unless the process has been explicitly defined to allow for such a migration.
- Variants of the first-order asynchronous π -calculus with explicit localities such as the Join calculus [6, 11], Nomadic Pict [24], DiTyCo [12], or the π_{1l} -calculus [1], feature process migration primitives (`go` in the Join calculus, `spawn` in the π_{1l} -calculus, `migrate` in Nomadic Pict) but lack sufficient control over resource access and process migration.

Compared to these works, the M-calculus has several distinguishing features.

Its notion of programmable locality allows the definition of different forms of locality within the same calculus. In contrast, the calculi above, or even calculi such as Klaim [14, 15] (that uses generative communication as its basic form of communication), or ATF [5] (that considers distributed 2-phase transactions as processes), consider only a single form of locality.

In distributed calculi, many alternatives exist when it comes to combine communication and localities. At one extreme lies the fully transparent communication of the Join calculus, where messages are routed directly to the target locality. At the other extreme lies Mobile Ambients, where communication is purely local to an ambient and remote communication must use migration primitives and explicitly encoded routes to deliver an ambient message. The Seal calculus [23] and Boxed Ambients lie between these two extremes by providing the ability to communicate across one locality boundary. In the M-calculus, we still provide transparent routing but allow messages to be intercepted each time they cross a locality boundary.

Finally, one may remark that there are striking similarities between controlling migration on one hand, and controlling communication on the other hand. For this reason, we take the further step to merge the two aspects, by considering a higher-order calculus. In our setting, migration becomes communication of a *think* or passivated process.

With respect to the type system, Yoshida and Hennessy’s work on $D\pi\lambda$, a higher-order distributed process calculus [26], is closest to our own. $D\pi\lambda$ allows the communication of processes as *thunks*. In this regard, it is similar to the M-calculus. There are however several important differences that we now detail. The two calculi take a different approach to the determinacy of communication. As in the Join calculus, the $D\pi\lambda$ calculus adopts a “channel locality” invariant, which ensures a channel or resource is only present in one locality. In the M-calculus, resources of the same name can be present in different localities in order to allow for dynamic binding. The M-calculus therefore relies on addressed resources, where a resource name is annotated with a locality

¹Recent work on a distributed abstract machine for Safe Ambients [17], just reinforces this point. The distributed implementation of ambients proposed there does away with the problem by implementing the `in` and `out` capabilities locally (by using cocapabilities and single-threadedness), and interpreting the `open` capability as a move to the implicit location of the parent ambient. In this interpretation, ambients no longer characterize the physical distribution of a computation, which defeats the original intent of the calculus. Furthermore, work on Boxed Ambients successfully argues against the `open` capability.

name, to ensure the determinacy of message routing. The “unicity of locality names” invariant in the M-calculus is more complex to ensure than the “channel locality” invariant in the $D\pi\lambda$ calculus. This is due to the passivation operator in the M-calculus. This operator has no counterpart in the $D\pi\lambda$ calculus, where the only way to obtain a thunk (frozen process) is by either specifying it, or by receiving it. The ability to passivate an active process, as in the M-calculus, is powerful but it makes the type system more complex, as the type of the passivation operator depends on some type information of the passivated process. As in the $D\pi\lambda$ calculus, a process in the M-calculus has a type that gives information on its behavior (its interface for the $D\pi\lambda$, its active localities for the M-calculus). More precisely, the type of a term is a conservative approximation of the value or process it may become. Thus locality errors in $D\pi\lambda$ and multiple locality definitions in the M-calculus only lead to type errors if the faulty process may eventually become active. Both systems also have a form of dependent types, but they are dealt with differently. In the $D\pi\lambda$ -calculus, name variables can occur in types and can be bound in types, yielding dependent types. In the M-calculus, the dependency between input and output of a resource (channel) is represented as a type variable (which may stand for a locality name) that is generalized. According to the type of the argument, the type variables of the type scheme of a resource are instantiated to the correct values, and the type of a message is thus dependent on the type of the argument. In order to simplify the type system of the M-calculus, we allow locality names in the types but not name variables. We use instead name type variables, which provide us with a cleaner distinction between types and names. The advantage of dependent types appears when considering partial application: if the result of an application is a function that has a dependent type, this function may still be applied to several frozen processes having different types, which is impossible when relying on first order polymorphism. However, previous work on the typing of the Join calculus and our experience with programming in JoCaml showed us that first order polymorphism is useful, well-understood, and powerful enough when using Join patterns. Finally, both the $D\pi\lambda$ calculus and the M-calculus allow input on channels that were received previously, but the $D\pi\lambda$ uses finer input-output types for channels that could be adapted to our setting.

6 Conclusion and future work

We have presented in this paper the M-calculus, a higher-order distributed process calculus, and an associated type system that statically enforces the unicity of locality names, a crucial invariant for the determinacy of final destination for remote messages. The M-calculus constitutes a non-trivial and powerful extension of the Join calculus [6, 11] with call-by-value higher-order functions, programmable localities, and dynamic binding.

The possibility to define, within the same calculus, different kinds of localities, with non-trivial behavior, is an important requirement for a realistic foundation of distributed mobile programming. While the M-calculus still falls short of meeting key requirements for advanced distributed programming (e.g. support for transactional behavior), we believe it constitutes an important step.

To validate the implementable character of the M-calculus, we have defined and implemented a distributed abstract machine for it [9], and, in order to prove the correctness of our abstract machine, we are working on an annotated lower-level calculus that makes the routing and passivation reduction rules more local. In parallel, we are defining notions of *observables* in order to compare, using operational equivalences, the annotated calculus to the original calculus, and to study the correctness of the translations from distributed calculi to the M-calculus.

References

- [1] R. Amadio. An asynchronous model of locality, failure, and process mobility. Technical report, INRIA Research Report RR-3109, INRIA Sophia-Antipolis, France, 1997.

- [2] M. Bugliesi, G. Castagna, and S. Crafa. Boxed ambients. In *4th International Symposium on Theoretical Aspects of Computer Software (TACS)*, 2001.
- [3] L. Cardelli. Types for mobile ambients. In *Proceedings 26th Annual ACM Symposium on Principles of Programming Languages (POPL)*, 1999.
- [4] L. Cardelli and A. Gordon. Mobile ambients. In *Foundations of Software Science and Computational Structures, M. Nivat (Ed.), Lecture Notes in Computer Science, Vol. 1378*. Springer Verlag, 1998.
- [5] D. Duggan. Atomic failure in wide-area computation. In *Proceedings 4th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, S. Smith and C. Talcott eds. Kluwer, 2000.
- [6] C. Fournet. *The Join-Calculus*. PhD thesis, Ecole Polytechnique, 1998.
- [7] C. Fournet, G. Gonthier, J.J. Levy, L. Maranget, and D. Remy. A calculus of mobile agents. In *In Proceedings 7th International Conference on Concurrency Theory (CONCUR '96), Lecture Notes in Computer Science 1119*. Springer Verlag, 1996.
- [8] C. Fournet, J.J. Levy, and A. Schmitt. An asynchronous distributed implementation of mobile ambients. In *Proceedings of the International IFIP Conference TCS 2000, Sendai, Japan, Lecture Notes in Computer Science 1872*. Springer, 2000.
- [9] F. Germain, M. Lacoste, and J.B. Stefani. An abstract machine for a higher-order distributed process calculus. In *Proceedings of the EACTS Workshop on Foundations of Wide Area Network Computing (F-WAN)*, July 2002.
- [10] F. Levi and D. Sangiorgi. Controlling interference in ambients. In *Proceedings 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2000)*, 2000.
- [11] J.J. Levy. Some results on the join-calculus. In *3rd International Symposium on Theoretical aspects of Computer Software (TACS), Lecture Notes in Computer Science no 1281*. Springer, 1997.
- [12] L. Lopes, F. Silva, A. Figueira, and V. Vasconcelos. DiTyCO: An Experiment in Code Mobility from the Realm of Process Calculi. In *Proceedings 5th Mobile Object Systems Workshop (MOS'99)*, 1999.
- [13] M. Merro and M. Hennessy. Bisimulation congruences in safe ambients. In *29th ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon, 16-18 January, 2002*.
- [14] R. De Nicola, G.L. Ferrari, and R. Pugliese. Klaim: a Kernel Language for Agents Interaction and Mobility. *IEEE Trans. on Software Engineering, Vol. 24, no 5*, 1998.
- [15] R. De Nicola, G.L. Ferrari, R. Pugliese, and B. Venneri. Types for Access Control. *Theoretical Computer Science, Vol. 240, no 1*, 2000.
- [16] Didier Rémy. Type inference for records in a natural extension of ML. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press, 1993.
- [17] D. Sangiorgi and A. Valente. A Distributed Abstract Machine for Safe Ambients. In *Proceedings of the 28th International Colloquium on Automata, Languages and Programming*, volume 2076 of *Lect. Notes in Comp. Sci.* Springer-Verlag, 2001.

- [18] A. Schmitt and J.B. Stefani. The M-calculus: A Higher Order Distributed Process Calculus. Draft of the long version, available at <http://pauillac.inria.fr/~aschmitt/publications.html>, 2002.
- [19] Alan Schmitt. *Conception et Implmentation de Calculs d'Agents Mobiles*. PhD thesis, Ecole Polytechnique, 2002.
- [20] Alan Schmitt. Safe Dynamic Binding in the Join Calculus. In *IFIP TCS'02*, Montreal, Canada, 2002.
- [21] Peter Sewell and Jan Vitek. Secure composition of untrusted code: Box- π , wrappers and causality types. *Journal of Computer Security*, 2000. Invited submission for a CSFW00 special issue. To appear.
- [22] D. Teller, P. Zimmer, and D. Hirschhoff. Using Ambients to Control Resources. In *Proceedings CONCUR 02*, 2002.
- [23] J. Vitek and G. Castagna. Towards a calculus of secure mobile computations. In *Proceedings Workshop on Internet Programming Languages, Chicago, Illinois, USA*, 1998.
- [24] P. Wojciechowski and P. Sewell. Nomadic Pict: Language and Infrastructure. *IEEE Concurrency*, vol. 8, no 2, 2000.
- [25] N. Yoshida and M. Hennessy. Subtyping and locality in distributed higher-order processes. In *Proceedings CONCUR 99, Lecture Notes in Computer Science no 1664*. Springer, 1999.
- [26] N. Yoshida and M. Hennessy. Assigning types to processes. In *15th Annual IEEE Symposium on Logic in Computer Science (LICS)*, 2000.

A Proofs

We prove in this section that every locality that is active in a process that has a type that is a set bears a unique name, and that our type system has the subject reduction property.

A.1 Preliminary lemmas

Proposition A.1 (Subtyping properties) 1. *The subtyping relation \leq is a partial order on types.*

2. *Let θ be a substitution from plain type variables to types, from multiset variables to multisets, and from name type variables to name type variables or locality names. For every type τ_1 and τ_2 , if we have $\tau_1 \leq \tau_2$, then we have $\tau_1\theta \leq \tau_2\theta$.*

3. *Let τ'_1, τ'_2, τ_1 , and τ_2 be types such that $\tau'_1 \leq \tau_1, \tau'_2 \leq \tau_2$. If $\tau_1 \sqcup \tau_2$ is defined, then $\tau'_1 \sqcup \tau'_2$ is defined and we have $\tau'_1 \sqcup \tau'_2 \leq \tau_1 \sqcup \tau_2$. If $\tau'_1 \sqcap \tau'_2$ is defined, then $\tau_1 \sqcap \tau_2$ is defined and we have $\tau'_1 \sqcap \tau'_2 \leq \tau_1 \sqcap \tau_2$.*

4. *Let τ_1 and τ_2 be types. If $\tau_1 \sqcup \tau_2$ is defined, then we have $\tau_1 \leq \tau_1 \sqcup \tau_2$. If $\tau_1 \sqcap \tau_2$ is defined, then we have $\tau_1 \sqcap \tau_2 \leq \tau_1$.*

5. *Let τ_1 and τ_2 be types. If $\tau_1 \sqcup \tau_2$ is defined, then we have $\text{fn}(\tau_1 \sqcup \tau_2) \subseteq \text{fn}(\tau_1) \cup \text{fn}(\tau_2)$. If $\tau_1 \sqcap \tau_2$ is defined, then we have $\text{fn}(\tau_1 \sqcap \tau_2) \subseteq \text{fn}(\tau_1) \cup \text{fn}(\tau_2)$.*

Proof: We first prove property (1). To this end, we show that the \leq relation is reflexive, anti-symmetric and transitive. We start with reflexivity, by induction on the structure of the type. We immediately have $\Delta \leq \Delta, \alpha \leq \alpha, \text{unit} \leq \text{unit}, \text{dom}(w) \leq \text{dom}(w)$, and $\langle \sigma \rangle_{\Delta}^+ \leq \langle \sigma \rangle_{\Delta}^+$. We conclude by induction for function types, tuple types, and plain resource types.

We now prove that \leq is antisymmetric. Let τ_1 and τ_2 be types such that $\tau_1 \leq \tau_2$ and $\tau_2 \leq \tau_1$. We prove by induction on the structure of τ_1 that $\tau_1 = \tau_2$. The result immediately holds if τ_1 is a plain type variable α , the unit type, or a $\text{dom}(w)$ type, since in all this cases τ_2 is necessarily the same type by definition of \leq . If τ_1 is a multiset Δ_1 , then τ_2 is necessarily a multiset Δ_2 by definition of \leq , thus we have $\Delta_1 \subseteq \Delta_2$ and $\Delta_2 \subseteq \Delta_1$. By definition of the multiset inclusion relation, we thus have $\Delta_1 = \Delta_2$. If τ_1 is a tuple type, we immediately conclude by induction. If τ_1 is a function type, we also immediately conclude by induction. We now consider the case when τ_1 is a plain resource type $\langle \sigma \rangle_{\Delta}$. We first prove that τ_2 is also a plain resource type. By definition of \leq , τ_2 is either a function type or a plain resource type. If τ_2 is a function type, we deduce from $\tau_2 \leq \tau_1$ and the definition of \leq that τ_1 is a function type, which is impossible. Thus τ_2 is a plain resource type, and we conclude by induction. We now consider the case when τ_1 is a sendable resource type $\langle \sigma \rangle_{\Delta}^+$. As in the previous case, we first prove that τ_2 is also a sendable resource type. By definition of \leq , τ_2 may either be a function type, a plain resource type, or a sendable resource type. If it is a function type, then by $\tau_2 \leq \tau_1$ the type τ_1 is also a function type, which is impossible. If τ_2 is a plain resource type, then by $\tau_2 \leq \tau_1$ the type τ_1 is either a plain resource type or a function type, which is impossible. Thus τ_2 is necessarily a sendable resource type and we conclude by the only case that deals with two sendable resource types in the definition of \leq .

We now show the transitivity of \leq . Let τ_1, τ_2 , and τ_3 be types such that $\tau_1 \leq \tau_2$ and $\tau_2 \leq \tau_3$. We prove by induction on the structure of τ_1 that $\tau_1 \leq \tau_3$. If τ_1 is either unit , $\text{dom}(w)$, or a plain type variable α , then the result immediately holds since we necessarily have $\tau_1 = \tau_2 = \tau_3$. If τ_1 is a multiset Δ_1 , then τ_2 is also a multiset Δ_2 (since $\tau_1 \leq \tau_2$), thus τ_3 is also a multiset Δ_3 (since $\tau_2 \leq \tau_3$). We thus have $\Delta_1 \subseteq \Delta_2 \subseteq \Delta_3$, thus $\Delta_1 \subseteq \Delta_3$. If τ_1 is a tuple type, then τ_2 and τ_3 are necessarily tuple types and we conclude by induction. If τ_1 is a function type, then necessarily τ_2 and τ_3 are also function types and we conclude by induction. If τ_1 is a resource type (either plain or sendable), we proceed by case on the shape of τ_2 and τ_3 . The cases to study are:

$$\begin{array}{rcl}
\tau_1 & \leq & \tau_2 \leq \tau_3 \\
\text{plain} & \leq & \text{plain} \leq \text{plain} \\
\text{plain} & \leq & \text{plain} \leq \text{function} \\
\text{plain} & \leq & \text{function} \leq \text{function} \\
\text{sendable} & \leq & \text{sendable} \leq \text{sendable} \\
\text{sendable} & \leq & \text{sendable} \leq \text{plain} \\
\text{sendable} & \leq & \text{sendable} \leq \text{function} \\
\text{sendable} & \leq & \text{plain} \leq \text{plain} \\
\text{sendable} & \leq & \text{plain} \leq \text{function} \\
\text{sendable} & \leq & \text{function} \leq \text{function}
\end{array}$$

We do not detail these cases, they all hold by induction.

We now prove property (2). Let τ_1 and τ_2 be types such that $\tau_1 \leq \tau_2$, and let θ be a substitution as described in the property. We proceed by induction on the structure of τ_1 . The property immediately holds if τ_1 is either **unit**, or $\text{dom}(w)$ (by definition of θ the resulting type is syntactically correct). The property holds for plain type variables since \leq is reflexive. If τ_1 is a multiset Δ_1 , then τ_2 is also a multiset Δ_2 . Since we have $\Delta_1 \subseteq \Delta_2$, we necessarily have $\Delta_2 = \Delta_1, \Delta'_2$. Thus we have $\Delta_1\theta \subseteq \Delta_1\theta, \Delta'_2\theta = \Delta_2\theta$ (the resulting type is a multiset by definition of θ). If τ_1 is a tuple, function or resource type, we conclude by induction.

We now prove property (3) by induction on the structure of τ_1 . If τ_1 is either **unit**, $\text{dom}(w)$, or a type variable α , we necessarily have $\tau_1 = \tau_2 = \tau'_1 = \tau'_2 = \tau_1 \sqcup \tau_2 = \tau'_1 \sqcup \tau'_2 = \tau_1 \sqcap \tau_2 = \tau'_1 \sqcap \tau'_2$ (we have $\tau_1 = \tau_2$ since $\tau_1 \sqcup \tau_2$ or $\tau'_1 \sqcap \tau'_2$ is defined). If τ_1 is a multiset Δ_1 , then τ_2 is necessarily a multiset Δ_2 (since $\tau_1 \sqcup \tau_2$ is defined). Thus by definition of \leq , τ'_1 and τ'_2 are also multisets Δ'_1 and Δ'_2 , and $\tau'_1 \sqcup \tau'_2$ is necessarily defined. We now prove that we have $\tau'_1 \sqcup \tau'_2 \leq \tau_1 \sqcup \tau_2$. We recall that for any name, name variable, or multiset variable, the number of occurrences of a name in $\Delta_1 \sqcup \Delta_2$ is the max of the number of occurrences of the name in Δ_1 and Δ_2 . Let a be a name. Since we have $\Delta'_1 \subseteq \Delta_1$, the number of occurrences of a in Δ'_1 is smaller than in Δ_1 . Similarly, the number of occurrences of a in Δ'_2 is smaller than in Δ_2 . Thus the number of occurrences of a in $\Delta'_1 \sqcup \Delta'_2$ is smaller than in $\Delta_1 \sqcup \Delta_2$. Thus we have $\Delta'_1 \sqcup \Delta'_2 \subseteq \Delta_1 \sqcup \Delta_2$. We now consider $\tau'_1 \sqcap \tau'_2$. Since all types are multisets, $\tau_1 \sqcap \tau_2$ is defined. Moreover, the number of occurrences of a name a in $\Delta'_1 \sqcap \Delta'_2$ is the min of its occurrences in Δ'_1 and Δ'_2 . Thus the number of occurrences of a in $\Delta'_1 \sqcap \Delta'_2$ is smaller than in $\Delta_1 \sqcap \Delta_2$, thus $\tau'_1 \sqcap \tau'_2 \leq \tau_1 \sqcap \tau_2$.

If τ_1 is a tuple, then τ_2 , τ'_1 , and τ'_2 are all tuples of the same size (by definition of \leq , \sqcup , and \sqcap). We immediately conclude by induction.

We now consider the case where the type is a function type. There are many cases depending on the shape of the other types. We detail some of them.

We now prove the case where $\sigma'_1 \rightarrow \tau'_1 \leq \sigma_1 \rightarrow \tau_1$ and $\sigma'_2 \rightarrow \tau'_2 \leq \sigma_2 \rightarrow \tau_2$. If $\sigma_1 \rightarrow \tau_1 \sqcup \sigma_2 \rightarrow \tau_2$ is defined, then necessarily $\sigma_1 \sqcap \sigma_2$ and $\tau_1 \sqcup \tau_2$ are defined. Since we have $\sigma_1 \leq \sigma'_1$, $\sigma_2 \leq \sigma'_2$, $\tau'_1 \leq \tau_1$, and $\tau'_2 \leq \tau_2$, we apply the induction hypothesis to prove that $\sigma'_1 \sqcap \sigma'_2$ is defined and that $\sigma_1 \sqcap \sigma_2 \leq \sigma'_1 \sqcap \sigma'_2$. We also use the induction hypothesis to prove that $\tau'_1 \sqcup \tau'_2$ is defined, and that $\tau'_1 \sqcup \tau'_2 \leq \tau_1 \sqcup \tau_2$. Thus $\sigma'_1 \rightarrow \tau'_1 \sqcup \sigma'_2 \rightarrow \tau'_2$ is defined and $\sigma'_1 \rightarrow \tau'_1 \sqcup \sigma'_2 \rightarrow \tau'_2 = (\sigma'_1 \sqcap \sigma'_2) \rightarrow (\tau'_1 \sqcup \tau'_2) \leq (\sigma_1 \sqcap \sigma_2) \rightarrow (\tau_1 \sqcup \tau_2) = \sigma_1 \rightarrow \tau_1 \sqcup \sigma_2 \rightarrow \tau_2$. The symmetric case where $\sigma'_1 \rightarrow \tau'_1 \sqcap \sigma'_2 \rightarrow \tau'_2$ is defined is proved similarly.

We now prove the case $\langle \sigma'_1 \rangle_{\Delta'_1} \leq \sigma_1 \rightarrow \Delta_1$ and $\langle \sigma'_2 \rangle_{\Delta'_2} \leq \langle \sigma_2 \rangle_{\Delta_2}$. If $\sigma'_1 \rightarrow \Delta'_1 \sqcap \langle \sigma'_2 \rangle_{\Delta'_2}$ is defined, then $\sigma'_1 \sqcup \sigma'_2$ is defined, as well as $\Delta'_1 \sqcap \Delta'_2$. By definition of \leq , we have $\sigma_1 \leq \sigma'_1$, $\sigma_2 \leq \sigma'_2$, $\Delta'_1 \leq \Delta_1$, and $\Delta'_2 \leq \Delta_2$. By induction, $\sigma_1 \sqcup \sigma_2$ is defined and $\sigma_1 \sqcup \sigma_2 \leq \sigma'_1 \sqcup \sigma'_2$, and $\Delta_1 \sqcap \Delta_2$ is defined, and $\Delta'_1 \sqcap \Delta'_2 \leq \Delta_1 \sqcap \Delta_2$. Thus $\sigma_1 \rightarrow \Delta_1 \sqcap \langle \sigma_2 \rangle_{\Delta_2}$ is defined, and:

$$\sigma'_1 \rightarrow \Delta'_1 \sqcap \langle \sigma'_2 \rangle_{\Delta'_2} = \langle \sigma'_1 \sqcup \sigma'_2 \rangle_{\Delta'_1 \sqcap \Delta'_2} \leq \langle \sigma_1 \sqcup \sigma_2 \rangle_{\Delta_1 \sqcap \Delta_2} = \sigma_1 \rightarrow \Delta_1 \sqcap \langle \sigma_2 \rangle_{\Delta_2}$$

We do not detail the many other cases. We remark that the cases that involve sendable resource types are easier since the shape of the other types are more constrained (for instance if τ_1 is a sendable resource types, then necessarily $\tau'_1 = \tau_1$ by definition of \leq).

We now prove property (4). We proceed by induction on the structure of τ_1 . The property is immediate if τ_1 is a type variable α , or one of the types `unit`, or $\text{dom}(w)$, since necessarily $\tau_1 = \tau_2 = \tau_1 \sqcup \tau_2 = \tau_1 \sqcap \tau_2$. If τ_1 is a multiset, then τ_2 is also a multiset and the property is immediately true by the definition of \sqcap and \sqcup .

If both τ_1 and τ_2 are function types or plain resource types, then the property holds by induction. If τ_1 and τ_2 are both sendable resource types, then if $\tau_1 \sqcup \tau_2$ is defined (or if $\tau_1 \sqcap \tau_2$ is defined), then we necessarily have $\tau_1 = \tau_2 = \tau_1 \sqcup \tau_2 = \tau_1 \sqcap \tau_2$.

We now consider the case where the types are $\langle \sigma_1 \rangle_{\Delta_1}$ and $\sigma_2 \rightarrow \Delta_2$. If $\langle \sigma_1 \rangle_{\Delta_1} \sqcup \sigma_2 \rightarrow \Delta_2$ is defined, then $\sigma_1 \sqcap \sigma_2$ and $\Delta_1 \sqcup \Delta_2$ are defined. By induction, we have $\sigma_1 \sqcap \sigma_2 \leq \sigma_1$ and $\Delta_1 \leq \Delta_1 \sqcup \Delta_2$. Thus we have $\langle \sigma_1 \rangle_{\Delta_1} \leq (\sigma_1 \sqcap \sigma_2) \rightarrow (\Delta_1 \sqcup \Delta_2) = \langle \sigma_1 \rangle_{\Delta_1} \sqcup \sigma_2 \rightarrow \Delta_2$. The symmetrical case is identical. If $\langle \sigma_1 \rangle_{\Delta_1} \sqcap \sigma_2 \rightarrow \Delta_2$ is defined, then $\sigma_1 \sqcup \sigma_2$ and $\Delta_1 \sqcap \Delta_2$ are defined. By induction, we have $\sigma_1 \leq \sigma_1 \sqcup \sigma_2$ and $\Delta_1 \sqcap \Delta_2 \leq \Delta_1$. Thus we have $\langle \sigma_1 \rangle_{\Delta_1} \sqcap \sigma_2 \rightarrow \Delta_2 = \langle \sigma_1 \sqcup \sigma_2 \rangle_{\Delta_1 \sqcap \Delta_2} \leq \langle \sigma_1 \rangle_{\Delta_1}$. The symmetrical case is identical.

We do not detail the cases with sendable resource types, they are similar, but simpler since more constrained.

We finally prove property (5). We once again proceed by induction on the structure of τ_1 . The property is immediate if τ_1 is a plain type variable α , or one of the types `unit`, or $\text{dom}(w)$, since necessarily $\tau_1 = \tau_2 = \tau_1 \sqcup \tau_2 = \tau_1 \sqcap \tau_2$. If τ_1 is a multiset Δ_1 , then by definition of \sqcap and \sqcup , τ_2 is a multiset Δ_2 , and we have $\text{fn}(\Delta_1 \sqcup \Delta_2) = \text{fn}(\Delta_1) \cup \text{fn}(\Delta_2)$ and $\text{fn}(\Delta_1 \sqcap \Delta_2) \subseteq \text{fn}(\Delta_1) \cup \text{fn}(\Delta_2)$. All the other cases hold using the induction hypothesis. \square

Lemma A.2 *If $\Gamma \vdash P : \Delta$, then we have $\text{fsv}(P) = \text{ftv}(P) = \text{fww}(P) = \emptyset$.*

Proof: We proceed by induction on the typing derivation. The result is immediate since the only syntactic construction that may include type variables are scope restriction for resource names, and the typing rules `NU.RES.1` and `NU.RES.2` explicitly check that no type variable is free. \square

Lemma A.3 *Let $\Gamma \vdash P : \tau$ be the conclusion of a typing derivation. We have $\text{fn}(\tau) \subseteq \text{fn}(\Gamma)$.*

Proof: We proceed by induction on the typing derivation.

`NIL`, `VOID`, `WILD` Immediate.

`NAME` The free name that may occur in $\sigma\theta$ are either in the range of θ , thus they are free in Γ by the typing rule hypothesis, or they already occur in σ and are necessarily free in Γ , since these names occur in the type scheme and locality names cannot be generalized.

`ADDR` By induction, we have $\text{fn}(\text{chan}(\sigma; \Delta)) \subseteq \text{fn}(\Gamma)$. Thus we have $\text{fn}(\sigma \rightarrow \Delta) = \text{fn}(\text{chan}(\sigma; \Delta)) \subseteq \text{fn}(\Gamma)$.

`FUN` By induction, we have $\text{fn}(\tau) \subseteq \text{fn}(\Gamma + x : \sigma)$. Since a variable x cannot occur in types, and by hypothesis of the typing rule $\text{fn}(\sigma) \subseteq \text{dom}(\Gamma)$, we thus have $\text{fn}(\sigma \rightarrow \tau) = \text{fn}(\sigma) \cup \text{fn}(\tau) \subseteq \text{fn}(\Gamma)$.

`DOM`, `PAR`, `TUPLE` Immediate by induction.

`NU.RES.1` We have by induction $\text{fn}(\Delta_1) \subseteq \text{fn}(\Gamma + r : \forall \tilde{\beta}. \langle \sigma \rangle_{\Delta})$. Since r may not occur in types, and since $\text{fn}(\forall \tilde{\beta}. \langle \sigma \rangle_{\Delta}) \subseteq \text{dom}(\Gamma)$, we thus have $\text{fn}(\Delta_1) \subseteq \text{fn}(\Gamma)$.

`NU.RES.1` We have by induction $\text{fn}(\Delta_1) \subseteq \text{fn}(\Gamma + r : \langle \sigma \rangle_{\Delta}^{\dagger})$. Since r may not occur in types, and since $\text{fn}(\langle \sigma \rangle_{\Delta}^{\dagger}) \subseteq \text{dom}(\Gamma)$, we thus have $\text{fn}(\Delta_1) \subseteq \text{fn}(\Gamma)$.

`NU.DOM` By induction hypothesis, we have $\text{fn}(\Delta) \subseteq \text{fn}(\Gamma) \cup \{a\}$. Since we have $a \notin \Delta - a$ by hypothesis of the typing rule, we thus have $\text{fn}(\Delta - a) \subseteq \text{fn}(\Gamma)$.

`PASS` Immediate by induction since $\text{fn}(\Delta) \cup \{\delta\} \cup \text{fn}(\Delta_1) \cup \text{fn}(\Delta_2) \subseteq \text{fn}(\Gamma)$.

APP Immediate by induction since $fn(\tau) \subseteq fn(fun(\sigma; \tau)) \subseteq fn(\Gamma)$.

TEST By property A.1(5), we have $fn(\tau_1 \sqcup \tau_2) \subseteq fn(\tau_1) \cup fn(\tau_2)$. We conclude by induction.

JOIN Immediate. \square

Lemma A.4 *Let $\Gamma \vdash P : \tau$ be the concluding judgment of a typing derivation. Let n be a name of a variable. Let n' be a name of the same kind as n (a locality name, a resource name or a variable) such that n' does not occur in the typing derivation. We have:*

$$\Gamma\{n'/n\} \vdash P\{n'/n\} : \tau\{n'/n\}$$

Proof: We proceed by induction on the typing derivation.

NIL, VOID, WILD Immediate.

NAME Since neither names nor variables may be generalized, we immediately have $u\{n'/n\} : \forall \tilde{\beta}. \sigma\{n'/n\} \in \Gamma\{n'/n\}$. We consider the instantiation $\theta\{n'/n\}$ restricted to the domain of θ and we prove that $\sigma\{n'/n\}\theta\{n'/n\} = \sigma\theta\{n'/n\}$. This is immediate since the domain of θ only include type variables, and since n' is fresh, thus $\{n'/n\}\{n'/n\} = \{n'/n\}$.

We still need to prove that $fn(ran(\theta\{n'/n\})) \subseteq dom(\Gamma\{n'/n\})$. If n is not free in $ran(\theta)$, then we have $fn(ran(\theta\{n'/n\})) = fn(ran(\theta)) \subseteq dom(\Gamma)$, and we conclude by remarking that if n is not free in $fn(ran(\theta))$, we also have $fn(ran(\theta)) \subseteq dom(\Gamma\{n'/n\})$. If n occurs in $fn(ran(\theta))$ then the result is immediate, since by hypothesis of the initial typing rule, we have $n \in dom(\Gamma)$.

In all cases, we conclude by rule NAME.

ADDR Immediate by induction, since names are replaced by names of the same kind.

FUN If n is x , then the substitution has no effect, since variables do not occur in types, and since x is not in the domain of Γ (thus it is not free in Γ). Otherwise the result is immediate by induction, since n' does not occur in the initial typing derivation, thus the hypothesis $x \notin dom(\Gamma\{n'/n\})$ is satisfied. The hypothesis on the free names of σ is also preserved by the renaming.

DOM Immediate by induction since the renaming preserves the name kind.

PAR, TUPLE Immediate by induction.

NU.RES.1 If n is r , then the substitution has no effect since resource names do not occur in types, and since $r \notin dom(\Gamma)$. Otherwise the result is immediate by induction, since n' does not occur in the initial typing derivation, thus $r \notin dom(\Gamma\{n'/n\})$. Moreover, since we have $fn(\forall \tilde{\beta}. \langle \sigma \rangle_{\Delta}) \subseteq dom(\Gamma)$, we then have $fn(\forall \tilde{\beta}. \langle \sigma\{n'/n\} \rangle_{\Delta\{n'/n\}}) \subseteq dom(\Gamma\{n'/n\})$ as neither n nor n' may be generalized.

NU.RES.2 If n is r , then the substitution has no effect since resource names do not occur in types, and since $r \notin dom(\Gamma)$. Otherwise the result is immediate by induction, since n' does not occur in the initial typing derivation, thus $r \notin dom(\Gamma\{n'/n\})$. Moreover, since we have $fn(\langle \sigma \rangle_{\Delta}^+) \subseteq dom(\Gamma)$, we then have $fn(\langle \sigma\{n'/n\} \rangle_{\Delta\{n'/n\}}^+) \subseteq dom(\Gamma\{n'/n\})$.

NU.DOM If n is a , then the substitution has no effect since a does not occur free in Γ nor in the type $\Delta - a$ by the typing rule hypotheses. Otherwise, the result is immediate by induction since n' is fresh thus cannot be a .

PASS We remark that neither n' nor n may be δ (which is a name type variable) nor ρ_1 nor ρ_2 (which are multiset variables). We have $(\Delta - \delta, \rho_1, \rho_2)\{n'/n\} = \Delta\{n'/n\} - (\delta, \rho_1, \rho_2)$, and we conclude by induction.

APP Immediate by induction, since the type structure and subtyping relation are preserved by the renaming of locality names.

TEST Immediate by induction, since the \sqcup and \sqcap operators commute with the renaming of a locality name to a fresh one, and since the type structure of the type of the pattern is unchanged by the renaming.

JOIN If n is one of the \tilde{x}_i , then the substitution has no effect, since no x_i is in the domain of Γ (thus every x_i is different from every \mathbf{r}_i) and no variable may occur in types. Otherwise the result is immediate by induction since the renaming is to a fresh name (thus the condition on the x_i still holds) and does not involve type variables. We also remark that the renaming preserves the name kind, which is required for the typing of the resource names. \square

In the following lemma, we write $\nu n.P$ for either $\nu r : s.P$ or $\nu a.P$. As before, we say that two names have the same kind if both are either resource names, locality names, or variables.

Lemma A.5 (α -conversion) *Let $\Gamma \vdash \nu n.P : \tau$ be the concluding judgment of a typing derivation, and n' be a fresh name of same kind than n . We then have $\Gamma \vdash \nu n'.(P\{n'/n\}) : \tau$.*

Let $\Gamma \vdash \lambda x.P : \tau$ be the concluding judgment of a typing derivation, and y be a fresh variable. We then have $\Gamma \vdash \lambda y.P\{y/x\} : \tau$.

Let $\Gamma \vdash \langle \mathbf{r}_1 \tilde{x}_1 \mid \dots \mid \mathbf{r}_n \tilde{x}_n \triangleright P \rangle : \emptyset$ be the concluding judgment of a typing derivation, and $(\tilde{y}_i)_{i \in [1..n]}$ be tuples of fresh and pairwise distinct variables (of same size as the \tilde{x}_i). We then have $\Gamma \vdash \langle \mathbf{r}_1 \tilde{y}_1 \mid \dots \mid \mathbf{r}_n \tilde{y}_n \triangleright P\{\tilde{y}_i/\tilde{x}_i\} \rangle : \emptyset$.

Proof: In each case, the last typing rule applied is necessarily either NU.RES.1 or NU.RES.2 for a scope restriction of a resource name, NU.DOM for a scope restriction of a locality name, FUN for a λ -abstraction, and JOIN for a reaction rule. In each case we use lemma A.4 with the typing of the guarded process (several times for the reaction rule case). As concerns the resource restriction or the λ -abstraction cases, the resulting type is the same (since variables and resource names do not occur in types), and the typing environment Γ is not modified (since the renamed name does not occur in the typing environment by the hypothesis of the typing rule). We conclude by rules NU.RES.1, NU.RES.2, or FUN since the condition on types is not modified by the renaming (only locality names may both occur in types and in the domain of Γ) and since the new name is fresh. As concern the reaction rule case, the type and the typing environment Γ are not modified by the renaming. We may conclude by typing rule JOIN since the new names are fresh, the resource names are not modified, and the other conditions on the types are not modified. As concern the case of the restriction of a locality names, we know that a is not free in Γ by hypothesis of rule NU.DOM, thus Γ is not modified by the substitution. Moreover, a occurs at most once in Δ , thus $\Delta\{a'/a\} - a' = \Delta - a$ and the final type is identical. We may conclude by rule NU.DOM since a' is fresh (thus may not be free in Γ). \square

Proposition A.6 (Subtyping and passivation) *Let σ be a type of the following shape:*

$$\text{dom}(\delta) \rightarrow (\text{unit} \rightarrow \Delta_1) \rightarrow \text{fun}(\text{unit} \rightarrow \Delta_2; \Delta)$$

For any type σ' such that $\sigma' \leq \sigma$, the shape of σ' is necessarily:

$$\text{dom}(\delta) \rightarrow (\text{unit} \rightarrow \Delta'_1) \rightarrow \text{fun}(\text{unit} \rightarrow \Delta'_2; \Delta')$$

with $\Delta' \subseteq \Delta$, $\Delta_1 \subseteq \Delta'_1$, and $\Delta_2 \subseteq \Delta'_2$.

Proof: There are three cases depending on the shape of $\text{fun}(\text{unit} \rightarrow \Delta_2; \Delta)$. We only detail the case where σ is $\text{dom}(\delta) \rightarrow (\text{unit} \rightarrow \Delta_1) \rightarrow (\text{unit} \rightarrow \Delta_2) \rightarrow \Delta$ (the other cases are simpler).

Since we have $\sigma' \leq \sigma$, we necessarily have $\sigma' = \text{dom}(\delta) \rightarrow \tau_1$ with $\tau_1 \leq (\text{unit} \rightarrow \Delta_1) \rightarrow (\text{unit} \rightarrow \Delta_2) \rightarrow \Delta$. This is the case because the result type of the first arrow in σ is not a process

type, thus the only possible subtyping rule that may apply is between functional types. Moreover, the first argument of σ' has type $\text{dom}(\delta)$ since it is the only supertype of $\text{dom}(\delta)$.

We now detail τ_1 , which necessarily has the shape $\tau_1 = \sigma_1 \rightarrow \tau_2$ with $\text{unit} \rightarrow \Delta_1 \leq \sigma_1$ and $\tau_2 \leq (\text{unit} \rightarrow \Delta_2) \rightarrow \Delta$. As before, the only subtyping rule that applies is the one between functional types (the result type is not a process type). As the only supertype of a functional type is a functional type, we have $\sigma_1 = \sigma'_1 \rightarrow \tau'_1$ with $\sigma'_1 \leq \text{unit}$ (thus $\sigma'_1 = \text{unit}$) and $\Delta_1 \leq \tau'_1$ (thus τ'_1 is a multiset Δ'_1 with $\Delta_1 \subseteq \Delta'_1$).

We now detail τ_2 . Since we have $\tau_2 \leq (\text{unit} \rightarrow \Delta_2) \rightarrow \Delta$, τ_2 has one of the following shapes:

1. $\sigma_2 \rightarrow \Delta'$ or
2. $\langle \sigma_2 \rangle_{\Delta'}$ or
3. $\langle \sigma_2 \rangle_{\Delta'}^+$.

with $\text{unit} \rightarrow \Delta_2 \subseteq \sigma_2$ and $\Delta' \subseteq \Delta$. We remark that we may more concisely describe τ_2 as $\text{fun}(\sigma_2; \Delta')$. With an identical argument than for σ_1 , we deduce that σ_2 has the shape $\text{unit} \rightarrow \Delta'_2$ with $\Delta_2 \subseteq \Delta'_2$.

We thus have $\sigma' = \text{dom}(\delta) \rightarrow (\text{unit} \rightarrow \Delta'_1) \rightarrow \text{fun}(\text{unit} \rightarrow \Delta'_2; \Delta')$ with $\Delta_1 \subseteq \Delta'_1$, $\Delta_2 \subseteq \Delta'_2$, and $\Delta' \subseteq \Delta$. \square

Lemma A.7 (Typing and contexts) *Let $C(\cdot : \tau)$ a typing context and P a process such that $\Gamma \vdash C(\cdot : \tau) : \tau_1$ and $\Gamma' \vdash P : \tau'$ with $\tau' \leq \tau$, where Γ' is the typing environment used to type the context hole. We require $C(P)$ to be a syntactically correct process.*

We then have $\Gamma \vdash C(P) : \tau'_1$ with $\tau'_1 \leq \tau_1$.

Moreover, if $\tau' = \tau$, we then have $\tau'_1 = \tau_1$.

Proof: We proceed by induction on the context structure. The additional property that the resulting type is unchanged if the type of the process filling the hole is the same is immediate.

$(\cdot : \tau)$ We necessarily have $\Gamma' = \Gamma$, and by hypothesis $\Gamma' \vdash P : \tau' \leq \tau$.

$\nu r : s.C$ The last typing rule used is necessarily **NU.RES.1** or **NU.RES.2**, depending on the shape of s . The result is thus immediate by induction.

$\nu a.C$ The last typing rule used is necessarily **NU.DOM**. By induction, we have a typing derivation for $\Gamma + a : \text{dom}(a) \vdash C(P) : \Delta'$ with $\Delta' \subseteq \Delta$. Since we have $a \notin \Delta - a$, we also necessarily have $a \notin \Delta' - a$ and $\Delta' - a \subseteq \Delta - a$. We conclude using rule **NU.DOM**.

$\lambda x.C$ The last typing rule used is necessarily **FUN**. By induction, we have a typing derivation for $\Gamma + x : \sigma \vdash C(P) : \tau'$ with $\tau' \leq \tau$. We may apply rule **FUN** and conclude by remarking that $\sigma \rightarrow \tau' \leq \sigma \rightarrow \tau$.

a(C)[Q] and a(P)[C] Immediate by induction, since the last typing rule used is necessarily **DOM**.

$C \mid Q$ **and** $P \mid C$ Immediate by induction, since the last typing rule used is necessarily **PAR**.

pass C The last typing rule used is necessarily **PASS**. Thus by induction we have a typing derivation for $\Gamma \vdash C(P) : \sigma'_V$ with $\sigma'_V \leq \sigma_V$. By proposition A.6, σ'_V has the expected shape. Moreover, ρ_1 still occurs in Δ'_1 since $\rho_1 \in \Delta_1 \subseteq \Delta'_1$, and similarly we also have $\rho_2 \in \Delta'_2$. Finally, we have $\delta, \rho_1, \rho_2 \notin \Delta' - \delta, \rho_1, \rho_2$ since $\Delta' \subseteq \Delta$. We may conclude by rule **PASS**, and we have $\Delta' - (\delta, \rho_1, \rho_2) \subseteq \Delta - (\delta, \rho_1, \rho_2)$.

CQ and PC In both cases, the last typing rule used is necessarily **APP**. In the first case, by induction we have a typing $\Gamma \vdash C(P) : \sigma'_P$ with $\sigma'_P \leq \sigma_P$. By definition of the subtyping relation, we have $\sigma'_P = \text{fun}(\sigma''; \tau')$ with $\sigma \leq \sigma''$ and $\tau' \leq \tau$ (we do not prove this property, the proof is identical to the one of proposition A.6). Since the subtyping relation is transitive

(property A.1(1)), we have $\sigma' \leq \sigma''$, and we may conclude by rule APP with $\tau' \leq \tau$ as concluding type.

The second case is immediate by induction, relying once more on the transitivity of \leq .

$([\mu = C]P, Q)$, $([\mu = V]C, Q)$ **and** $([\mu = V]P, C)$ In all cases, the last typing rule used is necessarily TEST. The result is immediate by induction using property A.1(3), that indicate that $\tau'_1 \sqcup \tau'_2$ exists and is a subtype of $\tau_1 \sqcup \tau_2$, and the property that any subtype of $\text{dom}(w)$ is $\text{dom}(w)$ and any subtype of $\text{chan}(\sigma_\mu; \Delta)$ is of the form $\text{chan}(\sigma'_\mu; \Delta')$.

$\langle J \triangleright C \rangle$ The last typing rule used is necessarily JOIN. The result is immediate by induction using the transitivity of \leq to compare the type of the guarded process $C(P)$ to the types of the resources.

(P_1, \dots, P_q) The last typing rule used is necessarily TUPLE. We conclude by induction. \square

From now on, we rely on lemmas A.5 and A.7 to work up-to α -renaming of names or variables to fresh names or variables.

Definition A.8 (Well formed type variable substitution) *A substitution θ is said to be well formed if and only if it is from plain type variables to types, from name type variables to locality names or name type variables, and from multiset variables to multisets, and such that $\theta\theta = \theta$.*

Lemma A.9 (Type variables instantiation) *Let $\Gamma \vdash P : \tau$ be the concluding judgment of a type derivation, and θ be a well formed substitution such that $\text{fn}(\text{ran}(\theta)) \subseteq \text{dom}(\Gamma\theta) = \text{dom}(\Gamma)$. We then have $\Gamma\theta \vdash P : \tau' \leq \tau\theta$.*

Moreover, if θ renames injectively name type variables to name type variables and multiset variables to multiset variables that do not occur in the initial typing derivation (there is no condition on plain type variables), we then have $\tau' = \tau\theta$.

Proof: We first remark that no type variable may be free in P , by lemma A.2. We proceed by induction on the typing derivation, for any well formed substitution θ . In most cases, the additional property leading to $\tau' = \tau\theta$ is immediate, we only detail the cases when it is not (rules PASS and TEST).

NIL, VOID, WILD Immediate.

NAME We assume that the generalized variables of $s = \forall \tilde{\beta}. \sigma$ are disjoint from the domain and variables in the range of θ . We then have $u : \forall \tilde{\beta}. \sigma\theta \in \Gamma\theta$. We let φ be the instantiation from s to $\sigma\varphi$. We remark that the substitution $\varphi\theta$ restricted to the domain of φ is an instantiation of s to $\sigma\varphi\theta$. We show that we have $\sigma\theta\varphi\theta = \sigma\varphi\theta$. We first consider a generalized variable of s , for instance β . By hypothesis, β is not in the domain of θ thus we have $\beta\theta = \beta$, and we conclude. Otherwise, we consider a type variable β that is not generalized (thus $\beta\varphi = \beta$). The generalized variables being disjoint from the variables occurring in the range of θ , we thus have $\beta\theta\varphi = \beta\theta$, thus $\beta\theta\varphi\theta = \beta\theta\theta = \beta\theta = \beta\varphi\theta$. Thus $\varphi\theta$ is an instantiation from $\forall \tilde{\beta}. \sigma\theta$ to $\sigma\varphi\theta$.

We now show that $\text{fn}(\text{ran}(\varphi\theta)) \subseteq \text{dom}(\Gamma\theta)$. This is immediately the case since we have $\text{fn}(\text{ran}(\varphi\theta)) \subseteq \text{fn}(\text{ran}(\varphi)) \cup \text{fn}(\text{ran}(\theta)) \subseteq \text{dom}(\Gamma) \cup \text{dom}(\Gamma\theta) = \text{dom}(\Gamma\theta)$.

ADDR By induction we have $\Gamma\theta \vdash \mathbf{r} : \tau'$ with $\tau' \leq \text{chan}(\sigma\theta; \Delta\theta)$. Thus necessarily τ' is of the form $\text{chan}(\sigma'; \Delta')$ with $\sigma\theta \leq \sigma'$ and $\Delta' \leq \Delta\theta$. By induction we also have $\Gamma\theta \vdash \mathbf{a} : \text{dom}(w\theta)$ (since the only subtype of $\text{dom}(w\theta)$ is $\text{dom}(w\theta)$, and since $w\theta$ is either a name type variable or a locality name), thus by rule ADDR we have $\Gamma\theta \vdash \mathbf{a.r} : \sigma' \rightarrow \Delta'$, with $\sigma' \rightarrow \Delta' \leq \sigma\theta \rightarrow \Delta\theta$.

FUN By induction, we have a typing $\Gamma\theta + x : \sigma\theta \vdash P : \tau'$ with $\tau' \leq \tau\theta$. Since the domain of θ only include type variables, we have $\text{dom}(\Gamma\theta) = \text{dom}(\Gamma)$. We now consider a name of $\sigma\theta$. Either the name is not in the range of θ , then it is necessarily present in σ , thus it is in $\text{dom}(\Gamma)$ by the hypothesis of the initial **FUN** typing rule. Otherwise, by hypothesis of the lemma it is also in $\text{dom}(\Gamma)$. Thus we have $\text{fn}(\sigma\theta) \subseteq \text{dom}(\Gamma\theta)$. We may apply rule **FUN**, and we conclude by remarking that $\sigma\theta \rightarrow \tau' \leq \sigma\theta \rightarrow \tau\theta$.

DOM, PAR, TUPLE Immediate by induction.

NU.RES.1, NU.RES.2 Immediate by induction, the type associated to r having no free type variable (thus staying unchanged by the substitution), and since $\text{dom}(\Gamma\theta) = \text{dom}(\Gamma)$.

NU.DOM If a occurs in the range of θ , it is first α -renamed to a fresh name. The result is then immediate by induction, since $a\theta = a$ and a is not in the range of θ , thus $(\Delta - a)\theta = \Delta\theta - a$.

PASS If the type variables δ , ρ_1 , or ρ_2 occur in the domain or in the range of θ , we first rename them injectively to fresh variables δ' , ρ'_1 , and ρ'_2 using the induction hypothesis. We remark that $\text{fn}(\text{ran}(\{\delta', \rho'_1, \rho'_2 / \delta, \rho_1, \rho_2\})) = \emptyset$. The type variables δ , ρ_1 , and ρ_2 are not free in Γ by hypothesis of the initial **PASS** typing rule. We thus have (since the renaming is injective to fresh variables, thus we are in the case where the result type is equal to the substituted type):

$$\Gamma \vdash V : \sigma_V \{\delta', \rho'_1, \rho'_2 / \delta, \rho_1, \rho_2\}$$

As δ , ρ_1 , and ρ_2 occur each at most once in Δ , and as δ' , ρ'_1 , and ρ'_2 are fresh, those variables occur at most once in $\Delta \{\delta', \rho'_1, \rho'_2 / \delta, \rho_1, \rho_2\}$ and we have $\Delta - (\delta, \rho_1, \rho_2) = \Delta \{\delta', \rho'_1, \rho'_2 / \delta, \rho_1, \rho_2\} - (\delta', \rho'_1, \rho'_2)$.

We now apply the induction hypothesis on the new derivation for V , and we obtain:

$$\Gamma\theta \vdash V : \sigma'_V$$

with $\sigma'_V \leq \sigma_V \{\delta', \rho'_1, \rho'_2 / \delta, \rho_1, \rho_2\}\theta$. By proposition A.6, σ'_V is of the form $\text{dom}(\delta') \rightarrow (\text{unit} \rightarrow \Delta'_1) \rightarrow \text{fun}(\text{unit} \rightarrow \Delta'_2; \Delta')$ with $\Delta_1 \{\delta', \rho'_1, \rho'_2 / \delta, \rho_1, \rho_2\}\theta \subseteq \Delta'_1$, $\Delta_2 \{\delta', \rho'_1, \rho'_2 / \delta, \rho_1, \rho_2\}\theta \subseteq \Delta'_2$, and $\Delta' \subseteq \Delta \{\delta', \rho'_1, \rho'_2 / \delta, \rho_1, \rho_2\}\theta$.

Since we initially have $\rho_1 \in \Delta_1$, we have $\rho'_1 \in \Delta_1 \{\delta', \rho'_1, \rho'_2 / \delta, \rho_1, \rho_2\}$. As ρ'_1 does not occur in the domain of θ , we have $\rho'_1 \in \Delta_1 \{\delta', \rho'_1, \rho'_2 / \delta, \rho_1, \rho_2\}\theta$. Thus we have $\rho'_1 \in \Delta'_1$. Similarly, we have $\rho'_2 \in \Delta'_2$.

We also have $\delta', \rho'_1, \rho'_2 \notin \text{fv}(\Gamma\theta) \cup (\Delta' - \delta', \rho'_1, \rho'_2)$, since neither δ' , ρ'_1 , nor ρ'_2 occur in the range of θ , and since they all occur at most once in $\Delta \{\delta', \rho'_1, \rho'_2 / \delta, \rho_1, \rho_2\}$.

We may apply the **PASS** typing rule. We now study the resulting type. We have (since the variables δ' , ρ'_1 , and ρ'_2 occur neither in the domain nor the range of θ):

$$\begin{aligned} \Delta' - (\delta', \rho'_1, \rho'_2) &\subseteq \Delta \{\delta', \rho'_1, \rho'_2 / \delta, \rho_1, \rho_2\}\theta - (\delta', \rho'_1, \rho'_2) \\ &= (\Delta \{\delta', \rho'_1, \rho'_2 / \delta, \rho_1, \rho_2\} - (\delta', \rho'_1, \rho'_2))\theta = (\Delta - (\delta, \rho_1, \rho_2))\theta \end{aligned}$$

In the equality case, we have $\Delta' = \Delta \{\delta', \rho'_1, \rho'_2 / \delta, \rho_1, \rho_2\}\theta$ and we deduce that $\Delta' - (\delta', \rho'_1, \rho'_2) = (\Delta - (\delta, \rho_1, \rho_2))\theta$.

APP By induction, we have $\Gamma\theta \vdash P : \sigma'_P$ with $\sigma'_P \leq \text{fun}(\sigma\theta; \tau\theta)$. Thus σ'_P is $\text{fun}(\sigma''; \tau'')$ with $\sigma\theta \leq \sigma''$ and $\tau'' \leq \tau\theta$.

We also have, by induction, $\Gamma\theta \vdash Q : \sigma'_Q \leq \sigma'\theta$. Since we have $\sigma' \leq \sigma$, by property A.1(2) we have $\sigma'\theta \leq \sigma\theta$, thus by transitivity $\sigma'_Q \leq \sigma''$. We may apply typing rule **APP** to yield:

$$\Gamma\theta \vdash PQ : \tau'' \leq \tau\theta$$

TEST We first show that if $\tau_1 \sqcup \tau_2$ is defined (respectively if $\tau_1 \sqcap \tau_2$ is defined), then $\tau_1\theta \sqcup \tau_2\theta$ is defined (respectively $\tau_1\theta \sqcap \tau_2\theta$ is defined) and we have $\tau_1\theta \sqcup \tau_2\theta \leq (\tau_1 \sqcup \tau_2)\theta$ (respectively $(\tau_1 \sqcap \tau_2)\theta \leq \tau_1\theta \sqcap \tau_2\theta$). This property is immediate by induction on the type structure, and is based on the facts that $(\Delta, \Delta')\theta = \Delta\theta, \Delta'\theta$, that $(\Delta \cap \Delta')\theta \subseteq \Delta\theta \cap \Delta'\theta$ (this last inclusion is an equality if name type variables and multiset variables are renamed injectively to fresh variables), and that $\tau \sqcup \tau = \tau \sqcap \tau = \tau$.

By induction, we have $\Gamma\theta \vdash \mu : \sigma'$ with $\sigma' \leq \sigma\theta$. If σ is $\text{dom}(w)$, then σ' is necessarily $\text{dom}(w\theta)$. If σ is $\text{chan}(\sigma_\mu; \Delta)$, then σ' is necessarily of the form $\text{chan}(\sigma'_\mu; \Delta')$.

By induction we then have $\Gamma\theta \vdash P : \tau'_1 \leq \tau_1\theta$ and $\Gamma\theta \vdash Q : \tau'_2 \leq \tau_2\theta$ (we do not detail the typing of the tested value). We have by A.1(3) that $\tau'_1 \sqcup \tau'_2$ is defined and $\tau'_1 \sqcup \tau'_2 \leq \tau_1\theta \sqcup \tau_2\theta \leq (\tau_1 \sqcup \tau_2)\theta$.

We conclude by rule **TEST**.

JOIN We first rename the generalized variables so that they occur neither in the domain nor the range of θ . Thus the generalization conditions are unchanged, as well as the condition on the formal parameters of the Join pattern. We also remark that the status of the resource names is not modified by the substitution, and that the shape of their types is also not modified. We may conclude by induction on the typing of the guarded process yielding type $\Delta'' \subseteq \Delta'\theta$, and by rule **JOIN** since $\Delta'' \subseteq \Delta'\theta \subseteq \Delta_1\theta, \dots, \Delta_n\theta$. \square

Lemma A.10 (Weakening) *Let $\Gamma \vdash P : \tau$ be the concluding judgment of a typing derivation, and u be a name such that $u \notin \text{dom}(\Gamma)$. We then have, for any type scheme s , $\Gamma + u : s \vdash P : \tau$.*

Proof: We proceed by induction on the typing derivation. Most cases are immediate, we omit them.

FUN If the variable x is u , we first α -rename it to a fresh variable (we remark that x cannot occur in s). Otherwise this case is immediate.

NU.RES.1, NU.RES.2 If the resource name r is u , we first α -rename it to a fresh resource name (we remark that r cannot occur in s). Otherwise this case is immediate.

NU.DOM If the locality name a is u , or if a is free in s , we first α -rename it to a fresh locality name. Then this case is immediate.

PASS If δ , ρ_1 , or ρ_2 occur free in s , we first rename them injectively in fresh type variables using lemma A.9. We remark, as in the **PASS** case of the proof of this lemma, that the final typing judgment is unchanged. We then conclude by induction.

JOIN We first remark that u cannot be one of the r_i since we have $u \notin \text{dom}(\Gamma)$. If u occurs in one \tilde{x}_i , we first α -rename this variable in the Join pattern. Moreover, if a generalized typed variable of a resource type scheme occurs free in s , we also rename it. We immediately conclude by induction on the typing of the guarded process. \square

Lemma A.11 (Strengthening) *Let $\Gamma + a : \text{dom}(a) \vdash P : \tau$ be the concluding judgment of a typing derivation, such that the locality name a does not occur in $\text{fn}(P) \cup \text{dom}(\Gamma)$. We then have $\Gamma\{\emptyset/a\} \vdash P : \tau\{\emptyset/a\}$.*

Let $\Gamma + r : s \vdash P : \tau$ be the concluding judgment of a typing derivation, such that the resource name r does not occur in $\text{fn}(P)$. We then have $\Gamma \vdash P : \tau$.

Proof: We first remark that $\Gamma\{\emptyset/a\}$ is a typing environment since a does not occur in the domain of Γ . We thus have $\text{dom}(\Gamma) = \text{dom}(\Gamma\{\emptyset/a\})$.

We proceed by induction on the typing derivation for both properties.

NIL, VOID, WILD Immediate.

NAME In the case of removing a resource name r from the typing environment, we first remark that the name being typed u is different from r since r does not occur free in P . Moreover, the only names that may occur free in the range of θ are domain names, thus the condition $fn(ran(\theta)) \subseteq dom(\Gamma)$ is satisfied. We conclude by rule NAME.

We now consider the case of removing a locality name a . As in the previous case, the name u being typed is necessarily different from a . By hypothesis of the typing rule, we have $u : \forall \tilde{\beta}. \sigma \in \Gamma$, thus we have $u : \forall \tilde{\beta}. \sigma\{\tilde{\beta}/a\} \in \Gamma\{\tilde{\beta}/a\}$. We assume that the initial substitution instantiating the type of u is θ . We now consider the substitution $\theta\{\tilde{\beta}/a\}$ restricted to the domain of θ . This substitution is an instantiation, and we have $\sigma\{\tilde{\beta}/a\}(\theta\{\tilde{\beta}/a\}) = (\sigma\theta)\{\tilde{\beta}/a\}$ (we remark that in the left-hand side of this equality, the substitution $\theta\{\tilde{\beta}/a\}$ has $dom(\theta)$ as domain, whereas in the right-hand side of the equality, the substitution $\{\tilde{\beta}/a\}$, which is not an instantiation since its domain include locality names, is applied to the type $\sigma\theta$). Moreover, since we have $fn(ran(\theta)) \subseteq dom(\Gamma) \cup \{a\}$, we necessarily have $fn(ran(\theta\{\tilde{\beta}/a\})) \subseteq dom(\Gamma\{\tilde{\beta}/a\}) = dom(\Gamma)$.

We conclude by rule NAME.

ADDR Immediate by induction.

FUN In the case of a resource name, we remark that it cannot be a variable, thus we may apply the induction hypothesis, since $r \in fn(P) \iff r \in fn(\lambda x.P)$. As resource names do not occur in types, the condition on the free names of σ stay unchanged, and we conclude by rule FUN.

In the case of a locality name, we remark that it cannot be a variable, thus we may apply the induction hypothesis, since $a \notin dom(\Gamma + x : \sigma)$ implies $a \in fn(P) \iff a \in fn(\lambda x.P)$. We then have:

$$\Gamma\{\tilde{\beta}/a\} + x : \sigma\{\tilde{\beta}/a\} \vdash P : \tau\{\tilde{\beta}/a\}$$

We now check that $fn(\sigma\{\tilde{\beta}/a\}) \subseteq dom(\Gamma\{\tilde{\beta}/a\}) = dom(\Gamma)$. This is the case since we have $fn(\sigma) \subseteq dom(\Gamma) \cup \{a\}$. We conclude by rule FUN.

DOM, PAR, TUPLE Immediate by induction.

NU.RES.1, NU.RES.2 In the case of a resource name r , we first remark that r is necessarily different from the restricted name since r occurs in the domain of the initial typing environment. We thus have $r \notin fn(P)$. The result is then immediate by induction, since resource names do not occur in types.

In the case of a locality name, if the restricted name is r , we remark that we have, by hypothesis of the lemma, $a \notin fn(\nu r : s.P)$. Thus we have $a \notin fn(P)$ (locality names are distinct from resource names), and $a \notin fn(s)$. We apply the induction hypothesis to yield:

$$\Gamma\{\tilde{\beta}/a\} + r : s \vdash P : \Delta_1\{\tilde{\beta}/a\}$$

and we conclude by rule NU.RES.1 or NU.RES.2, since a is not free in s .

NU.DOM The resource name case is immediate by induction.

We now consider the case of a locality name. Let b the locality name of the restriction. We necessarily have $b \neq a$ since a occurs in the domain of the initial typing environment. Thus we have $a \notin fn(P)$ and $a \notin dom(\Gamma + b : dom(b))$. We may apply the induction hypothesis, to yield:

$$\Gamma\{\tilde{\beta}/a\} + b : dom(b) \vdash P : \Delta\{\tilde{\beta}/a\}$$

The locality name b being different from a , we have $(\Delta - b)\{\tilde{\beta}/a\} = \Delta\{\tilde{\beta}/a\} - b$. We then conclude by rule NU.DOM.

PASS The resource name case is immediate by induction.

We now consider the case of a locality name. By induction, we have:

$$\Gamma\{\emptyset/a\} \vdash V : \sigma_V\{\emptyset/a\}$$

with $\sigma_v = \text{dom}(\delta) \rightarrow (\text{unit} \rightarrow \Delta_1\{\emptyset/a\}) \rightarrow \text{fun}(\text{unit} \rightarrow \Delta_2\{\emptyset/a\}; \Delta\{\emptyset/a\})$.

Since a and \emptyset are neither name type variables nor multiset variables, the other conditions holds and we may apply rule PASS to yield:

$$\Gamma\{\emptyset/a\} \vdash \text{pass } V : \Delta\{\emptyset/a\} - (\delta, \rho_1, \rho_2)$$

We conclude by remarking that $(\Delta - (\delta, \rho_1, \rho_2))\{\emptyset/a\} = \Delta\{\emptyset/a\} - (\delta, \rho_1, \rho_2)$.

APP The resource name case is immediate by induction.

We now consider the case of a locality name. By induction, we have:

$$\Gamma\{\emptyset/a\} \vdash P : \text{fun}(\sigma\{\emptyset/a\}; \tau\{\emptyset/a\})$$

and:

$$\Gamma\{\emptyset/a\} \vdash Q : \sigma'\{\emptyset/a\}$$

Since we have $\sigma' \leq \sigma$, we have $\sigma'\{\emptyset/a\} \leq \sigma\{\emptyset/a\}$. We conclude by rule APP.

TEST The resource name case is immediate by induction.

We now consider the case of a locality name. The result is also immediate by induction since the type of the pattern has correct shape after substitution, and since $\tau_1\{\emptyset/a\} \sqcup \tau_2\{\emptyset/a\} = (\tau_1 \sqcup \tau_2)\{\emptyset/a\}$.

JOIN The resource name case is immediate by induction, since the resource being removed cannot be one the resource names of the Join pattern. The generalization conditions immediately hold as there are fewer free variables in the typing environment.

We now consider the case of a locality name. We may apply the induction hypothesis (variables cannot be locality names). We have:

$$\Gamma\{\emptyset/a\} + \widetilde{x}_1 : \widetilde{\sigma}_1\{\emptyset/a\} + \dots + \widetilde{x}_n : \widetilde{\sigma}_n\{\emptyset/a\} \vdash P : \Delta\{\emptyset/a\}$$

We also have, since locality names cannot be generalized:

$$r_i : \forall \widetilde{\beta}_i. \langle \widetilde{\sigma}_i\{\emptyset/a\} \rangle_{\Delta_i\{\emptyset/a\}} \in \Gamma\{\emptyset/a\}$$

or

$$\mathbf{r}_i : \langle \sigma_i\{\emptyset/a\} \rangle_{\Delta_i\{\emptyset/a\}}^+ \in \Gamma\{\emptyset/a\}$$

We remark that we still have $\Delta'\{\emptyset/a\} \subseteq \Delta_1\{\emptyset/a\}, \dots, \Delta_n\{\emptyset/a\}$ and that the generalized type variables of the resource type schemes of the Join pattern are not modified (neither a nor \emptyset are type variables), as well as the free type variables of the typing environment. The generalization conditions thus hold, and we may conclude by rule JOIN. \square

A.2 Subject reduction and progress

Lemma A.12 (Well-formedness of typing environments) *Let $\Gamma \vdash \mathbf{E}(\cdot : \tau') : \tau$ be the concluding judgment of the typing derivation for an evaluation context, with Γ well-formed. Let Γ' the type environment used in the judgment of the typing derivation for typing the hole of $\mathbf{E}(\cdot : \tau')$. Then Γ' is well-formed.*

Proof: We proceed by induction on the context structure, and rely on the context shape to deduce the last typing rule used. We remark that the induction proceed in an unusual direction: we suppose that the typing environment in the last typing rule used is well formed, we show that the environment used in the hypotheses of this last rule are also well formed, and since the contexts types in these hypotheses are smaller, we deduce using the induction hypothesis that the hole is typed with a well-formed environment.

As concerns the well-formedness requirement on the free names of Γ , we only show that we have $fn(\Gamma) \subseteq dom(\Gamma)$ since we always have $dom(\Gamma) \subseteq fn(\Gamma)$.

(\cdot) Immediate, since $\Gamma = \Gamma'$.

EV, PE The last typing rule used is necessarily APP. The result is immediate by induction since the typing environment is not modified in the hypotheses of the typing rule.

$\nu n.E$ We first consider the restriction of a locality name. In this case, the last typing rule used is necessarily NU.DOM. We show that $\Gamma + a : dom(a)$ is well-formed. This is the case since the binding added has the correct form, and since $fn(\Gamma + a : dom(a)) = fn(\Gamma) \cup \{a\} \subseteq dom(\Gamma) \cup \{a\} = dom(\Gamma + a : dom(a))$. We conclude by induction.

We now consider the restriction of a resource name. According to the type explicitly given to the resource, the last typing rule used is either NU.RES.1 or NU.RES.2. In both cases, the binding added to the type environment has the correct form. Moreover, we have by rule hypothesis $fn(s) \subseteq dom(\Gamma)$, thus $fn(\Gamma + r : s) = fn(\Gamma) \cup \{r\} \cup fn(s) \subseteq dom(\Gamma) \cup \{r\} = dom(\Gamma + r : s)$. We conclude by induction.

E | P, P | E, a(P)[E], a(E)[P], (P₁, ..., E, ..., P_q) Immediate by induction, the typing environment being the same in the hypotheses of the last typing rule (PAR, DOM, and TUPLE) in all these cases. \square

Proof of lemma 3.2: We proceed by induction on the derivation of $P \equiv P'$, and by case on the rule used. Reflexivity and transitivity are immediate. Symmetry is proved for each case as we consider the two directions of the structural equivalence.

S.NU.PAR We consider the structural equivalence step: $(\nu n.P) | Q \equiv \nu n.P | Q$

There are four different cases to consider: scope extrusion or scope intrusion, resource name or locality name. In all cases, we have $n \notin fn(Q)$.

We first consider scope extrusion.

If n is a resource name r , we have the following initial typing derivation:

$$[\text{NU.RES.X}] \frac{\frac{\Gamma + r : s \vdash P : \Delta_1 \quad r \notin dom(\Gamma)}{\Gamma \vdash \nu r : s.P : \Delta_1} \quad \Gamma \vdash Q : \Delta_2}{\Gamma \vdash (\nu r : s.P) | Q : \Delta_1, \Delta_2} [\text{PAR}]$$

where NU.RES.X is either NU.RES.1 or NU.RES.2, depending on s , and with additional conditions on s that are unchanged by the reduction.

Since we have $r \notin dom(\Gamma)$, we may apply lemma A.10 to the subderivation for Q and build a type derivation whose conclusion is the judgment:

$$\Gamma + r : s \vdash Q : \Delta_2$$

We conclude by typing rules PAR and NU.RES.X.

We now consider the case where n is a locality name a . We have the initial typing derivation:

$$[\text{NU.DOM}] \frac{\frac{\Gamma + a : \text{dom}(a) \vdash P : \Delta_1 \quad a \notin \text{fn}(\Gamma) \quad a \notin (\Delta_1 - a)}{\Gamma \vdash \nu a.P : \Delta_1 - a}}{\Gamma \vdash (\nu a.P) \mid Q : (\Delta_1 - a), \Delta_2} \quad \Gamma \vdash Q : \Delta_2}{\Gamma \vdash (\nu a.P) \mid Q : (\Delta_1 - a), \Delta_2} [\text{PAR}]$$

Since a is not free in Γ , by lemma A.3 and the derivation $\Gamma \vdash Q : \Delta_2$, we have $a \notin \Delta_2$.

Since a is not free in Γ , it is not in its domain and we may apply lemma A.10 to the subderivation for Q to obtain a derivation ending with:

$$\Gamma + a : \text{dom}(a) \vdash Q : \Delta_2$$

We thus immediately build, using rule PAR, a derivation:

$$\Gamma + a : \text{dom}(a) \vdash P \mid Q : \Delta_1, \Delta_2$$

We remark that since $a \notin \Delta_2$ and $a \notin \Delta_1 - a$, we have $a \notin (\Delta_1, \Delta_2) - a$. We may thus apply rule NU.DOM to conclude.

We now consider scope intrusion.

In the case of a resource name r , we have the initial typing derivation:

$$[\text{PAR}] \frac{\frac{\Gamma + r : s \vdash P : \Delta_1 \quad \Gamma + r : s \vdash Q : \Delta_2}{\Gamma + r : s \vdash P \mid Q : \Delta_1, \Delta_2} \quad r \notin \text{dom}(\Gamma)}{\Gamma \vdash \nu r : s.P \mid Q : \Delta_1, \Delta_2} [\text{NU.RES.X}]$$

where, as in the scope extrusion case, NU.RES.X is either NU.RES.1 or NU.RES.2 depending on the shape of s , and with additional conditions on s that are unchanged in this case.

Since we have $r \notin \text{fn}(Q)$, we may apply lemma A.11 to the typing derivation for Q , to yield:

$$\Gamma \vdash Q : \Delta_2$$

We conclude by typing rules NU.RES.X for P and PAR.

We now consider the case of a locality name a . We have the initial typing derivation:

$$[\text{PAR}] \frac{\frac{\Gamma + a : \text{dom}(a) \vdash P : \Delta_1 \quad \Gamma + a : \text{dom}(a) \vdash Q : \Delta_2}{\Gamma + a : \text{dom}(a) \vdash P \mid Q : \Delta_1, \Delta_2}}{\vdots} \quad \frac{\vdots \quad a \notin \text{fn}(\Gamma) \quad a \notin (\Delta_1, \Delta_2) - a}{\Gamma \vdash \nu a.P \mid Q : (\Delta_1, \Delta_2) - a} [\text{NU.DOM}]$$

Since we have $a \notin \text{fn}(Q)$ and $a \notin \text{fn}(\Gamma)$ (thus $a \notin \text{dom}(\Gamma)$), we may apply lemma A.11 to the typing derivation for Q and obtain:

$$\Gamma \vdash Q : \Delta_2 \{\emptyset/a\}$$

since $\Gamma \{\emptyset/a\} = \Gamma$, as $a \notin \text{fn}(\Gamma)$.

By hypothesis, we have $a \notin (\Delta_1, \Delta_2) - a$, thus a occurs at most once in Δ_1, Δ_2 . If a does not occur in Δ_2 , we then have $(\Delta_1, \Delta_2) - a = (\Delta_1 - a), \Delta_2 = (\Delta_1 - a), \Delta_2 \{\emptyset/a\}$ and $a \notin \Delta_1 - a$. Otherwise a occurs at most once in Δ_2 and does not occur in Δ_1 , and we have $(\Delta_1, \Delta_2) - a = (\Delta_1 - a), (\Delta_2 - a) = (\Delta_1 - a), \Delta_2 \{\emptyset/a\}$ (since the substitution removes the single a of Δ_2), and we still have $a \notin \Delta_1 - a$.

In all cases, we may apply rule NU.DOM to P (since $a \notin \Delta_1 - a$), then apply rule PAR to get the type $(\Delta_1 - a), \Delta_2 \{\emptyset/a\} = (\Delta_1, \Delta_2) - a$.

S.NU.CTRL We consider the structural equivalence step: $a(\nu n.P)[Q] \equiv \nu n.a(P)[Q]$, with $n \neq a$ and $n \notin \text{fn}(Q)$.

As in the previous case, we need to consider four cases.

We first consider scope extrusion of a resource name r . We have the initial typing derivation:

$$[\text{NU.RES.X}] \frac{\frac{\Gamma + r : s \vdash P : \Delta_1 \quad r \notin \text{dom}(\Gamma)}{\Gamma \vdash \nu r : s.P : \Delta_1} \quad \Gamma \vdash a : \text{dom}(a) \quad \Gamma \vdash Q : \Delta_2}{\Gamma \vdash a(\nu r : s.P)[Q] : a, \Delta_1, \Delta_2} [\text{DOM}]$$

where NU.RES.X is either NU.RES.1 or NU.RES.2, depending on the shape of s .

The type of a is $\text{dom}(a)$ in Γ since Γ is well-formed.

Since $r \notin \text{dom}(\Gamma)$, we may apply lemma A.10 to the typing derivations for a and Q to yield:

$$\Gamma + r : s \vdash a : \text{dom}(a)$$

and:

$$\Gamma + r : s \vdash Q : \Delta_2$$

We now apply rule DOM to obtain the derivation:

$$\Gamma + r : s \vdash a(P)[Q] : a, \Delta_1, \Delta_2$$

and we conclude by rule NU.RES.X.

We do not detail the proof for the scope intrusion of a resource name r , since this proof is very similar to the previous case, except that it relies on the hypotheses $r \neq a$ and $r \notin \text{fn}(Q)$ to use lemma A.11 to strengthen the type derivations for a and Q .

We now consider the scope extrusion of a locality name b . We have the initial typing derivation:

$$\frac{\frac{\Gamma + b : \text{dom}(b) \vdash P : \Delta_1 \quad b \notin \text{fn}(\Gamma) \quad b \notin \Delta_1 - b}{\Gamma \vdash \nu b.P : \Delta_1 - b} [\text{NU.DOM}] \quad \begin{array}{c} \vdots \\ \Gamma \vdash a : \text{dom}(a) \quad \Gamma \vdash Q : \Delta_2 \end{array}}{\Gamma \vdash a(\nu b.P)[Q] : a, (\Delta_1 - b), \Delta_2} [\text{DOM}]$$

Since we have $b \notin \text{fn}(\Gamma)$, we may apply lemma A.3 to the subderivation $\Gamma \vdash Q : \Delta_2$ to deduce that $b \notin \Delta_2$. Moreover, since we have $b \notin \text{dom}(\Gamma)$, we may apply lemma A.10 to yield:

$$\Gamma + b : \text{dom}(b) \vdash a : \text{dom}(a)$$

and:

$$\Gamma + b : \text{dom}(b) \vdash Q : \Delta_2$$

We may now use rule DOM to obtain:

$$\Gamma + b : \text{dom}(b) \vdash a(P)[Q] : a, \Delta_1, \Delta_2$$

Since we have $b \neq a$ and $b \notin \Delta_2$, we have $(a, \Delta_1, \Delta_2) - b = a, (\Delta_1 - b), \Delta_2$. Moreover, since we have $b \notin \Delta_1 - b$, we immediately have $b \notin (a, \Delta_1, \Delta_2) - b$. We conclude by rule NU.DOM.

We now consider scope intrusion of a locality name b . Since we have $b \notin \text{fn}(Q)$ and $b \notin \text{dom}(\Gamma)$, we apply lemma A.11 to the type derivation $\Gamma + b : \text{dom}(b) \vdash Q : \Delta_2$ to yield:

$$\Gamma \vdash Q : \Delta_2\{\emptyset/b\}$$

Similarly, since $b \neq a$ and $b \notin \text{dom}(\Gamma)$, we have by the same lemma:

$$\Gamma \vdash a : \text{dom}(a)$$

In order to conclude, we need to show that $b \notin \Delta_1 - b$ (to apply rule NU.DOM to P), and that $a, (\Delta_1 - b), \Delta_2\{\emptyset/b\} = (a, \Delta_1, \Delta_2) - b$ (to make sure the final type is identical after rule DOM).

We recall that we initially have $b \notin (a, \Delta_1, \Delta_2) - b$ and $b \neq a$. Thus b occurs at most once in Δ_1, Δ_2 , thus at most once in Δ_1 . We always have $b \notin \Delta_1 - b$.

If b does not occur in Δ_2 , then we have $(a, \Delta_1, \Delta_2) - b = a, (\Delta_1 - b), \Delta_2 = a, (\Delta_1 - b), \Delta_2\{\emptyset/b\}$.

If b occurs in Δ_2 , then it occurs only once and cannot occur in Δ_1 . We then have $\Delta_2 - b = \Delta_2\{\emptyset/b\}$ and $\Delta_1 - b = \Delta_1$. Thus we have $(a, \Delta_1, \Delta_2) - b = a, \Delta_1, (\Delta_2 - b) = a, (\Delta_1 - b), \Delta_2\{\emptyset/b\}$.

In all cases we may conclude by rule NU.DOM for P , then by rule DOM .

S.NU.CONT This case is identical to **S.NU.CTRL**.

S. α This case is immediate, since by lemma A.5 and A.7, typing is done up to α -renaming to fresh names.

S.CONTEXT This case is immediate by induction, since by lemma A.12 the typing environment used to type the hole of the evaluation context is well-formed since Γ is well-formed, by induction hypothesis the structural equivalence step has the subject reduction property, and by lemma A.7 (in the case where the type of the process filling the hole is identical to the type of the hole), we build the final typing derivation yielding the same final type. \square

Lemma A.13 (Substitution lemma) *Let $\Gamma + x : \sigma \vdash P : \tau$ and $\Gamma \vdash V : \sigma'$ two typing derivations such that $\sigma' \leq \sigma$, such that every resource name in Γ is bound to a type of the form $\forall \tilde{\beta}. \langle \sigma_r \rangle_{\Delta_r}$ or of the form $\langle \sigma_r \rangle_{\Delta_r}^+$, such that every domain name is bound to a type of the form $\text{dom}(w)$, and such that every variable is bound to a monomorphic type.*

We then have a typing derivation of the form $\Gamma \vdash P\{V/x\} : \tau'$ with $\tau' \leq \tau$.

Proof: We proceed by induction on the typing derivation for P .

We first remark that if $\sigma = \text{dom}(w)$, then V is either a locality name or a variable, and if $\sigma = \text{chan}(\sigma_r; \Delta_r)$, then V is either a resource name or a variable. To prove this, we first remark that in the first case, σ' is necessarily $\text{dom}(w)$ (which is the only subtype of $\text{dom}(w)$), and in the second case, σ' is some $\text{chan}(\sigma'_r; \Delta'_r)$, by definition of \leq . Then we proceed by case on V . If V is $()$, then the only typing rule that applies is VOID and V has type unit . If V is a λ -abstraction, then by FUN its type is a functional type, if it is a tuple then by TUPLE its type is a tuple type. If V is an addressed resource name, then by rule ADDR its type is a functional type. The only cases remaining are variables, that may have any value type, resource names (then by rule NAME its type is some $\text{chan}(\sigma'_r; \Delta'_r)$, by hypothesis on Γ) and locality names (then by rule NAME , its type is necessarily $\text{dom}(w)$, by hypothesis on Γ).

We also remark that for any value V' such that $\Gamma + x : \sigma \vdash V' : \tau''$, the process $V'\{V/x\}$ is a value. We proceed by case on V . If it is $()$, a resource name r , a domain name a , or a variable name y different from x , then the result is immediate since the substitution does not change V' . If it is a function, it is of the form $\lambda y. P'$ with y different from x (since this function is necessarily typed using rule FUN that checks that the bound name is not present in the typing environment $\Gamma + x : \sigma$), thus we have $(\lambda y. P')\{V/x\} = \lambda y. (P'\{V/x\})$, which is a value. If V' is x , then $V'\{V/x\}$ is V , which is a value.

NIL, VOID, WILD Immediate.

NAME If x is u , then the result is immediately the hypothesis $\Gamma \vdash V : \sigma'$ since $\sigma' \leq \sigma$. Otherwise the result is immediate since removing the binding for $x : \sigma$ leaves unchanged the binding for u , and since we still have $fn(ran(\theta)) \subseteq dom(\Gamma)$, as names in the range of θ may only be domain names, and not variables.

ADDR We first remark that if \mathbf{r} is x , then $\sigma = chan(\sigma_r; \Delta_r)$ (since x is bound to a monomorphic σ and we have $\Gamma \vdash x : chan(\sigma_r; \Delta_r)$), thus V is either a resource name or a variable and the process obtained after substitution is syntactically correct. Similarly, we remark that if \mathbf{a} is x , then $\sigma = \mathbf{dom}(w)$ (since x is bound to a monomorphic σ and we have $\Gamma \vdash x : \mathbf{dom}(w)$), thus V is either a locality name or a variable name. We obtain a syntactically correct process after substitution.

By induction, we have $\Gamma \vdash \mathbf{a}\{V/x\} : \mathbf{dom}(w)$ (since $\mathbf{dom}(w)$ has only itself as subtype) as well as $\Gamma \vdash \mathbf{r}\{V/x\} : chan(\sigma'_r; \Delta'_r)$, since a subtype of $chan(\sigma_r; \Delta_r)$ is necessarily of the form $chan(\sigma'_r; \Delta'_r)$ with $\sigma_r \leq \sigma'_r$ and $\Delta'_r \leq \Delta_r$. We conclude by rule ADDR and we remark that $\sigma'_r \rightarrow \Delta'_r \leq \sigma_r \rightarrow \Delta_r$.

FUN By hypothesis of the FUN typing rule, we know that the substituted variable x is different from the λ -abstracted variable y , since x is in the domain of the initial typing environment. In order to apply the induction hypothesis, we first weaken the typing $\Gamma \vdash V : \sigma'$ to $\Gamma + y : \sigma_y \vdash V : \sigma'$, using lemma A.10 (which we may use since $y \notin dom(\Gamma)$). Moreover, we remark that we add a binding of a variable to a monomorphic type to Γ , thus the other condition to apply the induction hypothesis is satisfied. We thus have by induction the typing:

$$\Gamma + y : \sigma_y \vdash P\{V/x\} : \tau'$$

with $\tau' \leq \tau$. Since we had $fn(\sigma_y) \subseteq dom(\Gamma + x : \sigma)$, we necessarily have $fn(\sigma_y) \subseteq dom(\Gamma)$ as only locality names may occur in types. We apply rule FUN which yields a type $\sigma_y \rightarrow \tau'$ which is a subtype of $\sigma_y \rightarrow \tau$, and we conclude by remarking that $(\lambda y.P)\{V/x\} = \lambda y.(P\{V/x\})$.

DOM As in the previous cases, we rely on the typing of \mathbf{a} to prove that $\mathbf{a}\{V/x\}$ is syntactically correct. The result is then immediate by induction, as the only subtype of $\mathbf{dom}(w)$ is $\mathbf{dom}(w)$.

PAR, TUPLE Immediate by induction.

NU.RES.1, NU.RES.2 In this case we write s for the type of the restricted resource name r . We first remark that r cannot be x since r does not occur in the domain of the initial typing environment $\Gamma + x : \sigma$. We also remark that variables do not occur in types, so we have $s\{V/x\} = s$. We first weaken the derivation $\Gamma \vdash V : \sigma'$ to $\Gamma + r : s \vdash V : \sigma'$ by lemma A.10 (since $r \notin dom(\Gamma)$). We remark that in both cases (restriction of a plain or sendable resource), we extend the environment in a way that is compatible with the hypothesis of the lemma. We may apply the induction hypothesis to yield:

$$\Gamma + r : s \vdash P\{V/x\} : \Delta'_1$$

with $\Delta'_1 \leq \Delta_1$. To conclude by rule NU.RES.X, we remark that the name removed from the domain of $\Gamma + x : \sigma$ is a variable, thus cannot occur in types and $fn(s) \subseteq dom(\Gamma)$ is satisfied, and that $(\nu r : s.P)\{V/x\} = \nu r : s.(P\{V/x\})$.

NU.DOM As in the previous case, a cannot be x . We also weaken the derivation $\Gamma \vdash V : \sigma'$ to $\Gamma + a : \mathbf{dom}(a) \vdash V : \sigma'$ by lemma A.10 (since $a \notin fn(\Gamma)$, thus $a \notin dom(\Gamma)$). Since we extend Γ with a binding satisfying the lemma condition, we apply the induction hypothesis to yield:

$$\Gamma + a : \mathbf{dom}(a) \vdash P\{V/x\} : \Delta'$$

with $\Delta' \leq \Delta$. Since we have $a \notin \Delta - a$, we necessarily have $a \notin \Delta' - a$, and we may apply rule NU.DOM. We conclude by remarking that $(\nu a.P)\{V/x\} = \nu a.(P\{V/x\})$, and that $\Delta' - a \leq \Delta - a$.

PASS By induction, we have $\Gamma \vdash V'\{V/x\} : \sigma'_{V'}$, with $\sigma'_{V'} \leq \sigma_{V'}$. By a remark at the beginning of the proof, we know that $V'\{V/x\}$ is a value. By proposition A.6, we know that $\sigma'_{V'}$ has the expected form $\text{dom}(\delta) \rightarrow (\text{unit} \rightarrow \Delta'_1) \rightarrow \text{fun}(\text{unit} \rightarrow \Delta'_2; \Delta')$. Since we have $\Delta_1 \subseteq \Delta'_1$, we necessarily have $\rho_1 \in \Delta'_1$, and similarly $\rho_2 \in \Delta'_2$. Since the typing environment is smaller, we immediately have $\delta, \rho_1, \rho_2 \notin \text{fv}(\Gamma)$. Since $\Delta' \leq \Delta$, we also have $\delta, \rho_1, \rho_2 \notin \Delta' - \delta, \rho_1, \rho_2$ and $\Delta' - \delta, \rho_1, \rho_2 \leq \Delta - \delta, \rho_1, \rho_2$. We conclude by rule PASS.

APP By induction, we have $\Gamma \vdash P\{V/x\} : \sigma'_P$ with $\sigma'_P \leq \text{fun}(\sigma_P; \tau_P)$. By definition of \leq , σ'_P is necessarily of the form $\text{fun}(\sigma''_P; \tau''_P)$ with $\sigma_P \leq \sigma''_P$ and $\tau_P \leq \tau''_P$.

By induction, we also have $\Gamma \vdash Q\{V/x\} : \sigma''_Q$ with $\sigma''_Q \leq \sigma_Q$. We have by hypothesis of the initial typing rule $\sigma_Q \leq \sigma_P$, we thus have:

$$\sigma''_Q \leq \sigma_Q \leq \sigma_P \leq \sigma''_P$$

We conclude by rule APP.

TEST By induction, we have $\Gamma \vdash \mu\{V/x\} : \sigma'_\mu$ with $\sigma'_\mu \leq \sigma_\mu$. We first check that $\mu\{V/x\}$ is a pattern. If μ is $_$, a resource name r , a domain name a , or a variable y different from x , then the substitution has no effect. If μ is x , then σ' —the type of V —is necessarily a subtype of σ_μ , which has the form $\text{dom}(w)$ or $\text{chan}(\sigma''_\mu; \Delta''_\mu)$ by typing rule hypothesis. If σ_μ has the form $\text{dom}(w)$, then σ' is $\text{dom}(w)$, and by the remark at the beginning of the proof, V is either a domain name or a variable, which is a pattern. If σ_μ is of the form $\text{chan}(\sigma''_\mu; \Delta''_\mu)$, then σ' is of the form $\text{chan}(\sigma''_\mu; \Delta''_\mu)$ by definition of subtyping, then by the same remark V is either a resource name or a variable, which are both patterns.

We remark as well that σ'_μ has the correct form by definition of subtyping.

As concerns the typing of the matched value V' , we have by induction $\Gamma \vdash V'\{V/x\} : \tau'_{V'}$. By the second initial remark, $V'\{V/x\}$ is a value.

The result is then immediate by induction, relying on property A.1(3) to prove that the final type $\tau'_1 \sqcup \tau'_2$ exists and is smaller than $\tau_1 \sqcup \tau_2$.

JOIN If x is one the \mathbf{r}_i , then necessarily we have $\sigma = \langle \tilde{\sigma}_i \rangle_{\Delta_i}^+$, as x cannot be a resource name. Thus we have $\sigma' = \sigma$, since a sendable resource type only has itself as subtype. Moreover, V is necessarily a resource name or a variable \mathbf{r}' . Since we have $\Gamma \vdash \mathbf{r}' : \langle \tilde{\sigma}_i \rangle_{\Delta_i}^+$, then necessarily we have $\mathbf{r}' : \langle \tilde{\sigma}_i \rangle_{\Delta_i}^+ \in \Gamma$, since either \mathbf{r}' is a variable, which is necessarily bound to a monomorphic type, or \mathbf{r}' is a resource name with a sendable resource type, which is monomorphic by the lemma hypotheses.

We also remark that none of the \tilde{x}_i may be x , since the former variables do not occur in the domain of the initial typing environment $\Gamma + x : \sigma$.

We apply lemma A.10 to weaken the derivation $\Gamma \vdash V : \sigma'$ to $\Gamma + \tilde{x}_1 : \tilde{\sigma}_1 + \dots + \tilde{x}_n : \tilde{\sigma}_n \vdash V : \sigma'$. We remark that we only add variables bound to monomorphic types to Γ , satisfying the hypothesis of the lemma. We may apply the induction hypothesis to yield:

$$\Gamma + \tilde{x}_1 : \tilde{\sigma}_1 + \dots + \tilde{x}_n : \tilde{\sigma}_n \vdash P\{V/x\} : \Delta''$$

with $\Delta'' \leq \Delta'$.

We now check that all the conditions of rule JOIN are satisfied to obtain the judgment:

$$\Gamma \vdash \langle \mathbf{r}_1\{V/x\}\tilde{x}_1 \mid \dots \mid \mathbf{r}_n\{V/x\}\tilde{x}_n \triangleright P\{V/x\} \rangle : \emptyset$$

First of all, if \mathbf{r}_i is not a variable, then it is not modified by the substitution, and the condition on its presence in Γ is a direct consequence of its presence in $\Gamma + x : \sigma$. If \mathbf{r}_i is x , then we have $\mathbf{r}_i : \langle \tilde{\sigma}_i \rangle_{\Delta_i}^+ \in \Gamma + x : \sigma$, $V = \mathbf{r}'$, and $\mathbf{r}' : \langle \tilde{\sigma}_i \rangle_{\Delta_i}^+ \in \Gamma$. Thus the presence hypothesis is still

satisfied with an identical type. By transitivity, we have $\Delta'' \leq \Delta_1, \dots, \Delta_n$. We immediately have $(\tilde{x}_i)^i \cap \text{dom}(\Gamma) = \emptyset$ since we had initially $(\tilde{x}_i)^i \cap \text{dom}(\Gamma + x : \sigma) = \emptyset$. As the resource types are identical, and as the typing environment is smaller, the generalization conditions still hold.

We thus may apply rule JOIN and conclude.

We remark that it is necessary for a sendable resource type to have no subtype, otherwise this proof would not hold in this case (a generalization condition might break). \square

Proof of theorem 1: We proceed by induction on the reduction step (the induction is only needed for reduction under context and reduction between structural equivalence steps).

R.BETA We consider the reduction step $(\lambda x.P)V \rightarrow P\{V/x\}$

The initial typing derivation is:

$$[\text{FUN}] \frac{\frac{\Gamma + x : \sigma \vdash P : \tau \quad x \notin \text{dom}(\Gamma) \quad \text{fn}(\sigma) \subseteq \text{dom}(\Gamma)}{\Gamma \vdash \lambda x.P : \sigma \rightarrow \tau}}{\Gamma \vdash (\lambda x.P)V : \tau} \quad \Gamma \vdash V : \sigma' \quad \sigma' \leq \sigma \quad [\text{APP}]$$

As the typing environment Γ is well-formed (we recall that a well-formed environment does not bind variables), we may use the substitution lemma A.13 with the typing derivation for P , to yield:

$$\Gamma \vdash P\{V/x\} : \tau'$$

with $\tau' \leq \tau$.

R.IF.THEN We consider the reduction step $([\mu = V]P, Q) \rightarrow P$ with $\text{match}(\mu, V)$

The initial typing derivation is (we omit the typing of μ and V which are not necessary for this proof):

$$\frac{\Gamma \vdash P : \tau_1 \quad \Gamma \vdash Q : \tau_2}{\Gamma \vdash ([\mu = V]P, Q) : \tau_1 \sqcup \tau_2} \quad [\text{TEST}]$$

Since we have $\tau_1 \leq \tau_1 \sqcup \tau_2$ by A.1(4), we conclude by the subderivation $\Gamma \vdash P : \tau_1$.

R.IF.ELSE We consider the reduction step $([\mu = V]P, Q) \rightarrow Q$ with $\neg \text{match}(\mu, V)$.

The initial typing derivation is (omitting the typing of μ and V):

$$\frac{\Gamma \vdash P : \tau_1 \quad \Gamma \vdash Q : \tau_2}{\Gamma \vdash ([\mu = V]P, Q) : \tau_1 \sqcup \tau_2} \quad [\text{TEST}]$$

Since we have $\tau_2 \leq \tau_1 \sqcup \tau_2$ by A.1(4), we conclude by the subderivation $\Gamma \vdash Q : \tau_2$.

R.PASSIV We consider the reduction:

$$a(\text{pass } V \mid P)[Q] \rightarrow Va(\lambda.P)(\lambda.Q)$$

The initial typing derivation is:

$$[\text{PAR}] \frac{\frac{\Gamma \vdash V : \sigma_V}{\Gamma \vdash \text{pass } V : \Delta_3} \quad [\text{PASS}] \quad \Gamma \vdash P : \Delta_P}{\Gamma \vdash (\text{pass } V \mid P) : \Delta_3, \Delta_P} \quad \vdots \quad \frac{\Gamma \vdash a : \text{dom}(a) \quad \Gamma \vdash Q : \Delta_Q}{\Gamma \vdash a(\text{pass } V \mid P)[Q] : a, \Delta_3, \Delta_P, \Delta_Q} \quad [\text{DOM}]$$

with

$$\begin{aligned}\sigma_V &= \text{dom}(\delta) \rightarrow (\text{unit} \rightarrow \Delta_1) \rightarrow \text{fun}(\text{unit} \rightarrow \Delta_2; \Delta) \\ \Delta_3 &= \Delta - \delta, \rho_1, \rho_2\end{aligned}$$

and $\rho_1 \neq \rho_2$, $\rho_1 \in \Delta_1$, $\rho_2 \in \Delta_2$, $\delta, \rho_1, \rho_2 \notin \text{fv}(\Gamma) \cup \Delta_3$.

The type for a is deduced from the fact that Γ is well-formed.

We first apply lemma A.9 to the subderivation for V to rename injectively δ , ρ_1 , and ρ_2 to variables that do not appear in Δ_P nor Δ_Q using substitution θ_V . As neither δ , ρ_1 , nor ρ_2 are free in Γ , and as we are in the case where the resulting type is the same than the initial substituted type, we have:

$$\Gamma \vdash V : \sigma_V \theta_V$$

We remark that $\sigma_V \theta_V$ has still the same correct form, and all conditions for applying rule PASS are satisfied. As δ , ρ_1 , and ρ_2 respectively occur at most once in Δ , we have $\Delta - \delta, \rho_1, \rho_2 = \Delta \theta_V - (\delta, \rho_1, \rho_2) \theta_V$, thus the result after applying rule PASS is the same.

We suppose, to simplify the notation, that the renaming has been taken care of, and that the final type variable names are still δ , ρ_1 , and ρ_2 .

Thus we have $\{^{a; \Delta_P; \Delta_Q / \delta; \rho_1; \rho_2}\} \{^{a; \Delta_P; \Delta_Q / \delta; \rho_1; \rho_2}\} = \{^{a; \Delta_P; \Delta_Q / \delta; \rho_1; \rho_2}\}$ and the substitution $\theta = \{^{a; \Delta_P; \Delta_Q / \delta; \rho_1; \rho_2}\}$ is well-formed.

By lemma A.3 and since Γ is well-formed, we have $\text{fn}(\Delta_P) \subseteq \text{fn}(\Gamma) = \text{dom}(\Gamma)$. Similarly, we have $\text{fn}(\Delta_Q) \subseteq \text{fn}(\Gamma)$. We also have $a \in \text{dom}(\Gamma)$. Thus we have $\text{fn}(\text{ran}(\theta)) \subseteq \text{dom}(\Gamma)$. We may apply lemma A.9 with the substitution θ and the derivation $\Gamma \vdash V : \sigma_V$. We obtain $\Gamma \vdash V : \sigma'_V$ with $\sigma'_V \leq \sigma_V \theta$, since δ , ρ_1 , and ρ_2 are not free in Γ .

As $\rho_1 \in \Delta_1$ and $\rho_2 \in \Delta_2$, we have $\Delta_1 = \rho_1, \Delta'_1$ and $\Delta_2 = \rho_2, \Delta'_2$. Thus we have $\sigma_V \theta = \text{dom}(a) \rightarrow (\text{unit} \rightarrow \Delta_P, (\Delta'_1 \theta)) \rightarrow \text{fun}(\text{unit} \rightarrow \Delta_Q, (\Delta'_2 \theta)); \Delta \theta$.

By proposition A.6, we have $\sigma'_V = \text{dom}(a) \rightarrow (\text{unit} \rightarrow \Delta''_1) \rightarrow \text{fun}(\text{unit} \rightarrow \Delta''_2; \Delta'')$ with $\Delta_P, (\Delta'_1 \theta) \subseteq \Delta''_1$, $\Delta_Q, (\Delta'_2 \theta) \subseteq \Delta''_2$, and $\Delta'' \subseteq \Delta \theta$.

We immediately have $\Gamma \vdash \lambda.P : \text{unit} \rightarrow \Delta_P$ and $\Gamma \vdash \lambda.Q : \text{unit} \rightarrow \Delta_Q$ by rule FUN. Moreover, since $\Delta_P \subseteq \Delta''_1$ and $\Delta_Q \subseteq \Delta''_2$, we have $\text{unit} \rightarrow \Delta_P \leq \text{unit} \rightarrow \Delta''_1$, as well as $\text{unit} \rightarrow \Delta_Q \leq \text{unit} \rightarrow \Delta''_2$. We apply rule APP three times (for a , $\lambda.P$, and $\lambda.Q$) to obtain:

$$\Gamma \vdash Va(\lambda.P)(\lambda.Q) : \Delta''$$

We now remark that $\Delta_3 = \Delta - \delta, \rho_1, \rho_2$. Thus we have $\Delta \subseteq \Delta_3, \delta, \rho_1, \rho_2$. Thus we have $\Delta \theta \subseteq \Delta_3, a, \Delta_P, \Delta_Q$ (since $\delta, \rho_1, \rho_2 \notin \Delta_3$). Since we have $\Delta'' \subseteq \Delta \theta$, the final type is a subtype of the initial type $\Delta_3, a, \Delta_P, \Delta_Q$.

R.RES We have the following reduction step:

$$\langle J \triangleright P \rangle | r_1 \widetilde{V}_1 | \dots | r_n \widetilde{V}_n \rightarrow \langle J \triangleright P \rangle | P\{\widetilde{V}_i / \widetilde{x}_i\}$$

with $\langle J \triangleright P \rangle = \langle r_1 \widetilde{x}_1 | \dots | r_n \widetilde{x}_n \triangleright P \rangle$.

The typing of the initial process is:

$$[\text{J OIN}] \frac{(*)}{\Gamma \vdash \langle J \triangleright P \rangle : \emptyset} \quad \Gamma \vdash r_1 \widetilde{V}_1 | \dots | r_n \widetilde{V}_n : \Delta}{\Gamma \vdash \langle J \triangleright P \rangle | r_1 \widetilde{V}_1 | \dots | r_n \widetilde{V}_n : \Delta} [\text{PAR}]$$

To conclude, we need to replace the derivation $\Gamma \vdash r_1 \widetilde{V}_1 | \dots | r_n \widetilde{V}_n : \Delta$ with the derivation $\Gamma \vdash P\{\widetilde{V}_i / \widetilde{x}_i\} : \Delta_0$ with $\Delta_0 \subseteq \Delta$.

To this end, we now detail the typing of the reaction rule:

$$\frac{\begin{array}{c} (r_i : s_i = \forall \tilde{\beta}_i. \langle \tilde{\sigma}_i \rangle_{\Delta_i} \in \Gamma \text{ or } r_i : s_i = \langle \tilde{\sigma}_i \rangle_{\Delta_i}^+ \in \Gamma)^{i \in [1..n]} \\ \Delta' \leq \Delta_1, \dots, \Delta_n \quad (\tilde{x}_i)^i \cap \text{dom}(\Gamma) = \emptyset \quad \Gamma + \tilde{x}_1 : \tilde{\sigma}_1 + \dots + \tilde{x}_n : \tilde{\sigma}_n \vdash P : \Delta' \\ \forall i \in [1..n]. \tilde{\beta}_i \cap \text{fv}(\Gamma) = \emptyset \quad \forall i, j \in [1..n]^2. i \neq j \implies \tilde{\beta}_i \cap \tilde{\beta}_j = \emptyset \end{array}}{(*) \Gamma \vdash \langle r_1 \tilde{x}_1 \mid \dots \mid r_n \tilde{x}_n \triangleright P \rangle : \emptyset} \text{ [JOIN]}$$

as well as the typing derivation for each message, first for resources that have a plain resource type:

$$\frac{\text{[NAME]} \frac{r_i : \forall \tilde{\beta}_i. \langle \tilde{\sigma}_i \rangle_{\Delta_i} \in \Gamma \quad \langle \tilde{\sigma}_i \theta_i \rangle_{\Delta_i \theta_i} = \text{Inst}(\forall \tilde{\beta}_i. \langle \tilde{\sigma}_i \rangle_{\Delta_i})}{\Gamma \vdash r_i : \langle \tilde{\sigma}_i \theta_i \rangle_{\Delta_i \theta_i}} \quad \Gamma \vdash \tilde{V}_i : \tilde{\sigma}'_i \text{ [APP]}}{\Gamma \vdash r_i \tilde{V}_i : \Delta_i \theta_i}$$

with $\text{fn}(\text{ran}(\theta_i)) \subseteq \text{dom}(\Gamma)$ and $\tilde{\sigma}'_i \leq \tilde{\sigma}_i \theta_i$; then for resources with a sendable resource type:

$$\text{[NAME]} \frac{r_i : \langle \tilde{\sigma}_i \rangle_{\Delta_i}^+ \in \Gamma}{\Gamma \vdash r_i : \langle \tilde{\sigma}_i \rangle_{\Delta_i}^+} \quad \Gamma \vdash \tilde{V}_i : \tilde{\sigma}'_i \text{ [APP]} \quad \Gamma \vdash r_i \tilde{V}_i : \Delta_i$$

with $\tilde{\sigma}'_i \leq \tilde{\sigma}_i$.

To simplify the presentation, we consider for sendable resources an empty instantiation substitution θ_i (the identity) that has empty domain and empty range. We thus immediately have $\text{fn}(\text{ran}(\theta_i)) \subseteq \text{dom}(\Gamma)$.

We have $\Delta = \Delta_1 \theta_1, \dots, \Delta_n \theta_n$ by several applications of the PAR typing rule for the messages.

We assume in the following that the generalized variables of the resource type schemes s_i are all distinct from the type variables occurring in the range of the θ_i (renaming these generalized variables to fresh variables if necessary, in Γ as well as in the typing of the guarded process P , by lemma A.9, in the case where the final type is identical to the initial type).

The second generalization condition of the JOIN typing rule guarantees that no generalized variable is shared by two type schemes, thus the domains of the θ_i are pairwise disjoint. Moreover, the range of any θ_i is disjoint from the domain of any θ_j since generalized variables are disjoint from type variables occurring in the range of the θ_i . Let θ be the union of all the substitutions θ_i , we then have $\theta\theta = \theta$. Thus θ is a well-formed substitution.

Since we have $\text{fn}(\text{ran}(\theta)) \subseteq \text{dom}(\Gamma)$ (as it is the case for every θ_i), we may apply lemma A.9 to obtain:

$$\Gamma + \tilde{x}_1 : \tilde{\sigma}_1 \theta + \dots + \tilde{x}_n : \tilde{\sigma}_n \theta \vdash P : \Delta''$$

with $\Delta'' \subseteq \Delta' \theta$ (we recall that the first generalization condition guarantees that generalized variables, the domain of θ , do not occur free in Γ).

We have:

$$\Delta'' \subseteq \Delta' \theta \subseteq \Delta_1 \theta, \dots, \Delta_n \theta = \Delta$$

We now apply several times the substitution lemma A.13 (which is possible since the environment is extended with variables bound to monomorphic types) and obtain:

$$\Gamma \vdash P\{\tilde{V}_i/\tilde{x}_i\} : \Delta_0$$

with $\Delta_0 \subseteq \Delta''$.

We remark that the order used to apply the substitutions does not matter, since no \tilde{x}_i occur in Γ , thus they cannot occur in \tilde{V}_i .

We conclude by rule PAR.

R.CONTEXT We consider the reduction step: $\mathbf{E}(P) \rightarrow \mathbf{E}(Q)$ with $P \rightarrow Q$.

Since we have an initial typing derivation $\Gamma \vdash \mathbf{E}(P) : \tau_0$, we split this derivation into $\Gamma' \vdash P : \tau$ and $\Gamma \vdash \mathbf{E}(\cdot : \tau) : \tau_0$, where Γ' is the environment used to type the hole. By lemma A.12, Γ' is well-formed. Thus by induction we have a type derivation $\Gamma' \vdash Q : \tau'$ with $\tau' \leq \tau$. We conclude by lemma A.7 to obtain $\Gamma \vdash \mathbf{E}(Q) : \tau'_0$ with $\tau'_0 \leq \tau_0$.

R.EQUIV We consider the reduction step: $P_1 \rightarrow P_2$ with $P_1 \equiv P'_1 \rightarrow P'_2 \equiv P_2$. We have the initial typing derivation $\Gamma \vdash P_1 : \tau_1$. By lemma 3.2, we have a typing derivation $\Gamma \vdash P'_1 : \tau_1$. By induction, we have a typing derivation $\Gamma \vdash P'_2 : \tau_2$ with $\tau_2 \leq \tau_1$. Finally, by lemma 3.2, we have a typing derivation $\Gamma \vdash P_2 : \tau_2$.

ROUTING Every routing step immediately satisfy the subject reduction property if we prove the following cases: $\Gamma \vdash a.r\tilde{V} : \Delta$ implies $\Gamma \vdash r\tilde{V} : \Delta$, $\Gamma \vdash a.r\tilde{V} : \Delta$ implies $\Gamma \vdash i(a, r, \tilde{V}) : \Delta$ and $\Gamma \vdash o(a, r, \tilde{V}) : \Delta$.

For the first property, we have the initial typing derivation:

$$\frac{\frac{\Gamma \vdash a : \text{dom}(a) \quad \Gamma \vdash r : \text{chan}(\sigma; \Delta)}{\Gamma \vdash a.r : \sigma \rightarrow \Delta} [\text{ADDR}] \quad \Gamma \vdash \tilde{V} : \sigma' \quad \sigma' \leq \sigma}{\Gamma \vdash a.r\tilde{V} : \Delta} [\text{APP}]$$

We then build the derivation:

$$\frac{\Gamma \vdash r : \text{chan}(\sigma; \Delta) \quad \Gamma \vdash \tilde{V} : \sigma' \quad \sigma' \leq \sigma}{\Gamma \vdash r\tilde{V} : \Delta} [\text{APP}]$$

We now prove the second property. We only deal we the *i* resource, the *o* case being identical. The initial typing derivation is the same as the initial one for the first property. We now build the derivation:

$$\frac{i : \forall \alpha \delta \rho. \langle \text{dom}(\delta), \langle \alpha \rangle_\rho, \alpha \rangle_\rho \in \Gamma \quad \{a\} \cup \text{fn}(\sigma) \cup \text{fn}(\Delta) \subseteq \text{dom}(\Gamma)}{\Gamma \vdash i : \langle \text{dom}(a), \langle \sigma \rangle_\Delta, \sigma \rangle_\Delta} [\text{NAME}]$$

$$\vdots$$

$$\frac{\Gamma \vdash a : \text{dom}(a) \quad \Gamma \vdash r : \text{chan}(\sigma; \Delta) \quad \Gamma \vdash \tilde{V} : \sigma'}{\Gamma \vdash (a, r, \tilde{V}) : (\text{dom}(a), \text{chan}(\sigma; \Delta), \sigma')} [\text{TUPLE}]$$

$$\vdots$$

$$\frac{(\text{dom}(a), \text{chan}(\sigma; \Delta), \sigma') \leq (\text{dom}(a), \langle \sigma \rangle_\Delta, \sigma)}{\Gamma \vdash i(a, r, \tilde{V}) : \Delta} [\text{APP}]$$

since $\text{chan}(\sigma; \Delta)$ is either $\langle \sigma \rangle_\Delta$ or $\langle \sigma \rangle_\Delta^+$, and both are subtypes of $\langle \sigma \rangle_\Delta$.

We remark that the condition $\{a\} \cup \text{fn}(\sigma) \cup \text{fn}(\Delta) \subseteq \text{dom}(\Gamma)$ is satisfied for the following reason. Since we initially have $\Gamma \vdash a.r : \sigma \rightarrow \Delta$, by lemma A.3 we have $\text{fn}(\sigma) \cup \text{fn}(\Delta) \subseteq \text{fn}(\Gamma)$. Since Γ is well-formed, we thus have $\text{fn}(\sigma) \cup \text{fn}(\Delta) \subseteq \text{dom}(\Gamma)$. Moreover, as we have $\Gamma \vdash a : \text{dom}(a)$, we necessarily have $a \in \text{dom}(\Gamma)$. \square

Proof of theorem 2: This proof proceeds in three steps. We first show that for any well-typed process P , the multiset $\text{locs}(P)$ is the multiset of free and active localities of P . Then we show that if we have a typing derivation $\Gamma \vdash P : \Delta$ with Γ well-formed, then we have $\text{locs}(P) \subseteq \Delta$. Finally we show that if we have $\Gamma \vdash \mathbf{E}(\cdot : \Delta') : \Delta$ with Δ a set, then we Δ' is a set.

The first step is immediate for most cases, by the definition of *free and active*, thus by definition of evaluation contexts. However, the application and tuple cases are not immediate: we need to justify that $\text{locs}(PQ) = \emptyset$ and $\text{locs}(P_1, \dots, P_q) = \emptyset$, since those two constructions are allowed

in evaluation contexts. To this end, we show that if we have a typing derivation $\Gamma \vdash P : \tau$ where τ is not a multiset, then P contains no process in evaluation context having a multiset type. This implies there is no active domain in evaluation context in P . We proceed by induction on the typing derivation, and only consider cases where the type of the conclusion of the rule may not be a multiset. The cases NAME, VOID, WILD, and ADDR are immediate because of the shape of the process typed which cannot be an evaluation context. The case FUN is immediate since the process contains no subprocess in evaluation context (there is no evaluation under a λ -abstraction). The cases TUPLE and APP are immediate by induction (that we may apply since no type in the hypotheses of the rules may be a multiset). All other cases either give a multiset type to the process typed (NIL, DOM, PAR, NU.RES.1, NU.RES.2, NU.DOM, PASS, JOIN), or are not evaluation contexts (TEST).

Let $\Gamma \vdash P : \Delta$ a typing derivation. We may now show that a locality is free and active in P if and only if it is in $locs(P)$. We suppose that a is free and active in P . There is an evaluation context $\mathbf{E}(\cdot)$ such that no restriction for a restricts the hole and such that $P = \mathbf{E}(a(P')[Q'])$ for some P' and Q' . We proceed by induction on the structure of $\mathbf{E}(\cdot)$. Most cases are immediate, except for the restriction, where we use the fact that a is not restricted, and the application and tuple cases. For the application case, we remark that typing such an application $P_a Q_a$ necessarily uses rule APP. Moreover, we remark that the types of P_a and Q_a cannot be multisets. Thus neither P_a nor Q_a contain processes in evaluation context that have a multiset type, thus no active domain. The tuple case is identical. We now suppose that a is in $locs(P)$. The result is immediate by induction on the structure of P . For instance, we have $a \in locs(\nu n.P)$. Thus necessarily we have $a \in locs(P)$ and $a \neq n$. Thus, by induction, a is free and active in P . Since $\nu n.(\cdot)$ is an evaluation context, and since n is not a , a is free and active in $\nu n.P$.

We now show that if we have $\Gamma \vdash P : \Delta$ with Γ well-formed, then we have $locs(P) \subseteq \Delta$. We proceed by induction on the typing derivation (we only consider the cases where the concluding type may be a multiset).

NIL Immediate.

DOM Since Γ is well-formed, we necessarily have $w = a = \mathbf{a}$. We conclude by induction.

PAR Immediate by induction.

NU.RES.1, NU.RES.2 Immediate by induction (we may use the induction hypothesis since the extended environment is well-formed, as $fn(s) \subseteq dom(\Gamma)$). We conclude using the fact that $locs(P) \setminus \{r\} = locs(P)$.

NU.DOM We first remark that the extended environment is well formed. We apply the induction hypothesis, thus we have $locs(P) \subseteq \Delta$. We immediately have $locs(P) \setminus \{a\} \subseteq \Delta - a$.

APP Immediate since $locs(PQ) = \emptyset$.

PASS Immediate since $locs(\text{pass } V) = \emptyset$.

TEST Immediate since $locs([\mu = V]P, Q) = \emptyset$.

JOIN Immediate since $locs(\langle J \triangleright P \rangle) = \emptyset$.

We now show that if $\Gamma \vdash \mathbf{E}(\cdot : \Delta') : \Delta$ where Δ is a set, then Δ' is a set. We proceed as in lemma A.12 by induction on the evaluation context structure. We recall that this induction proceeds in the reverse direction compared to usual inductions on type derivations: we show that the hypothesis of the last typing rule has a multiset type that is a set, then by induction we deduce that the type of the hole is a set.

(\cdot) Immediate since $\Delta' = \Delta$.

EV, PE, $(P_1, \dots, \mathbf{E}, \dots, P_q)$ These cases cannot occur. For tuples, because the concluding type is not a multiset type. For applications, we recall that well-typed processes that have a type that is not a multiset cannot contain a subprocess in evaluation context that has a multiset type. Thus for the application case, the function and argument do not have a multiset type, thus the hole cannot have a multiset type.

E | $P, P | \mathbf{E}, a(P)[\mathbf{E}], a(\mathbf{E})[P], \nu r : s.\mathbf{E}$ In all these cases, the multiset type of the hypothesis is included in the multiset type in the conclusion of the typing rule. Thus if the latter is a set (by hypothesis), then the former is also a set and we conclude by induction.

$\nu a.\mathbf{E}$ The typing rule used is necessarily **NU.DOM**. Since $\Delta - a$ is a set and a occurs at most once in Δ (as $a \notin \Delta - a$), then Δ is a set. We conclude by induction.

We now use these three results to conclude. Let $\Gamma \vdash P : \Delta$ a typing derivation with Γ well-formed and Δ a set. Let $\mathbf{E}(\cdot)$ be an evaluation context and P' a process such that $P = \mathbf{E}(P')$. By the first property, we know that the set of free and active localities of P' is $locs(P')$. Let $\Gamma' \vdash P' : \tau'$ be the subderivation typing P' . We know that P' does not contain any active locality if τ' is not a multiset. Otherwise, let τ' be Δ' . By lemma A.12, we know that Γ' is well-formed. Thus by the second property, we have $locs(P') \subseteq \Delta'$. Finally, since Δ is a set, by the third property Δ' is also a set, thus $locs(P')$ is a set: every free and active locality of P' bears a unique name. \square