Safe Dynamic Binding in the Join Calculus

Alan Schmitt
INRIA Rocquencourt, alan.schmitt@inria.fr

Abstract

This paper presents an extension of the distributed Join Calculus with messages dynamically bound to definitions according to their location. New dynamic channels and new definitions for dynamic channels may be created at runtime. A dynamic message is rebound when the location containing the message migrates. A sound type system is introduced to guarantee that every dynamic message is bound to a definition.

1 Introduction

In a distributed world, channel-based communication usually involves senders and receivers that may reside in different locations. A receiver is a *definition*, an association between a channel name and a process that is spawned upon reception of a message on this channel. A message sent on a channel is routed to a location containing a definition for this channel. Thus the binding of senders to receivers is dependent upon the routing of messages. On the one hand, the binding may be *static*, in the sense that the routing does not depend on where the message originates, greatly simplifying remote communication. The distributed Join Calculus [10, 9] makes such a choice, by restricting the definition of a given channel to a single location. On the other hand, the routing may depend on where the message is. This *dynamic* binding models location-dependent behaviors. This paper enriches the distributed Join Calculus with channels whose messages are routed dynamically.

The distributed Join Calculus is a process calculus featuring locations, structured as a tree. Subtrees of locations migrate when the root of the subtree migrates. Channel names in the Join Calculus have a static, global scope: messages on a given name are transparently sent to the location where this name is defined, independently of the tree structure of locations. Because of space constraints, we do not present the distributed Join Calculus. Readers who do not feel familiar with this calculus can find a tutorial in [11].

The JoCaml system [15], an implementation of the distributed Join Calculus, provides some form of dynamic binding to functions defined in the current runtime, such as the standard library. Modeling this behavior is also one of our goals.

One requirement for our system is the ability to create new definitions for existing dynamic channels. For instance, a programmer may introduce a bug in a definition of some dynamic channel named `print`. Later on, he notices this bug and creates a new definition of `print`. Of course, he cannot restrain from adding a few new features to this version. As one of these features is actually a new dynamic channel for setting the printing color, our programmer needs to introduce a new channel name, `setcolor` for instance, and to provide a definition.

However, when considering such a system of first class dynamic channels where definitions may be added and new channel names may be created at runtime, it is difficult to check manually that every message is bound to a definition. This *receptiveness* property should prevent mistakes like trying to send a message on the `setcolor` channel when it is not bound. This paper aims at such a safe calculus of local resources using a static type system.

An important design choice is the relation between the process being rebound to new definitions and the process triggering this rebinding. As in [6], we distinguish between *objective* and *subjective* rebinding with respect to the triggering process. The rebinding is objective if it is triggered by a guard, and the (guarded) process that is being rebound is not active. On the opposite, rebinding is subjective when it involves running processes that are in the environment of the process triggering the rebinding. For instance, dynamically scoped variables, like implicit parameters [16], can be considered as an objective form of dynamic binding, since the value associated to a name is modified in the continuation of the rebinder, and nowhere else. On the contrary, JoCaml is a subjective dynamic binding system, because rebinding occurs when migrating, and migration in JoCaml is subjective. As subjective dynamic calculi have less control on the scope of the rebinding, they let the programmer modify the execution environment of running processes. However, this makes the receptiveness property significantly harder to prove, since the process being rebound is not explicitly available when typing the process triggering the rebinding.

Our extension adds dynamic channels and corresponding definitions, called *dynamic definitions*. The definition bound to a dynamic channel at a given location is the closest definition of this channel in the enclosing locations. We say that a dynamic channel is *available* when there is a definition of this channel in enclosing locations. Since migration in the distributed Join Calculus is subjective, our calculus is a subjective dynamic calculus. As a location may redefine a channel already defined in an enclosing location, this model is an extension of the JoCaml model, where only *runtimes*—top level locations that do not migrate—provide definitions for dynamic channels.

The dynamic Join Calculus is designed to be implemented in a distributed setting. In the distributed Join Calculus, each static channel is defined in a single location. This property does not hold for dynamic channels, as a dynamic channel may be defined in several locations. However, the routing of a message on a dynamic channel is deterministic as there is only one closest enclosing location defining this channel. Moreover, we require the determination of the destination of a message to be local. This insures that there is no need for distributed synchronization.

Part of this work could have been achieved in a distributed π Calculus. However, this would have been more subtle in the π Calculus since the association between senders and receivers is more dynamic than in the Join Calculus. In the π Calculus, receivers may disappear. It is therefore more complex to distinguish between deadlock freedom [1, 14] and availability of receivers.

In section 2 we present the syntax and semantics of the dynamic Join Calculus; in section 3 we discuss our system and we give examples; in section 4 we present a type system that guarantees the presence of definitions, and we state its soundness in section 5; we conclude in section 6.

2 Syntax and semantics

$\mathcal{P}$	::=		process
		$\mathbf{0}$	inert process
	$\mid$	$\mathcal{P} \mid \mathcal{P}'$	parallel composition
	$\mid$	$n\langle \widetilde{n_i} \rangle ; P$	message
	$\mid$	$\texttt{go}\ n ; P$	migration
	$\mid$	$\texttt{def}\ D\ \texttt{in}\ P$	local definition
	$\mid$	$\nu \mathbf{n}.P$	new dynamic channel
	$\mid$	$a.n\langle \widetilde{n_i} \rangle$	resolved message
$\mathcal{D}$	::=		definition
		$\top$	void definition
	$\mid$	$\mathcal{D}, \mathcal{D}'$	composition
	$\mid$	$J \triangleright P$	reaction rule
	$\mid$	$a[\mathcal{D} : \mathcal{P}]^{\Delta, I}$	folded location
J	::=		join pattern
		$n\langle \widetilde{y} \rangle$	message pattern
	$\mid$	$J \mid J'$	synchronization

n	::=		name
		$\mathbf{n}$	dynamic channel name
	$\mid$	$\mathtt{n}$	static channel name
	$\mid$	a	location name
	$\mid$	y	variable
Δ, I	::=		sets of dynamic names
		$\emptyset$	empty set
	$\mid$	$\{\mathbf{n}\}$	single dynamic name
	$\mid$	$\{y\}$	single variable
	$\mid$	$\Delta \cup \Delta$	union
φ	::=		string of locations
			empty string
	$\mid$	$\varphi' a$	a sublocation of φ'
$\mathcal{S}$	::=		configuration
		Ω	nothing
	$\mid$	$\mathcal{S} \parallel \mathcal{S}'$	soup composition
	$\mid$	$\mathcal{D} \vdash_{\varphi a}^{\Delta, I, F} \mathcal{P}$	running location

Figure 1: Syntax for the dynamic Join Calculus

The chemical syntax, structural semantics, and reduction semantics are given in figures 1, 2, and 3 respectively. We presuppose the existence of an infinite set of names ranged over by m, n, x. Location names are ranged over by a, b, c, dynamic channel names are ranged over by $\mathbf{m}, \mathbf{n}, \mathbf{x}$, static names are ranged over by $\mathtt{m}, \mathtt{n}, \mathtt{x}$, and variables are ranged over by u, y. We write $\widetilde{n}$ for possibly empty tuples of names. We write P (resp. D) for processes (resp. definitions) that do not contain any occurrence of resolved messages (of the form $a.n\langle\widetilde{n_i}\rangle$). We write $\mathcal{P}$ (resp. $\mathcal{D}$) for processes (resp. definitions) which may contain occurrences of resolved messages, as they may occur in active processes (resp. active definitions) after several reduction steps. Free names, received names, and defined names are defined as usual. A local definition $\mathtt{def}\ D\ \mathtt{in}\ P$ binds within D and P the static channel names and location names defined in D. However, it does not bind the dynamic channel names defined in D. A restriction $\nu\mathbf{n}.P$ binds the dynamic name $\mathbf{n}$ in P. A reaction rule $J \triangleright P$ binds the received names of J in P. The formal definitions can be found in appendix A. We also introduce the notion of *defined local names* (*dln*) as names that are defined in a given location, *defined static names* (resp. *defined dynamic names*) (*dsn*) (resp. *ddn*) as defined names that are static (resp. defined names that are dynamic).

Intuitively, a configuration consists of several concurrently running locations, also called *soups*. Each location contains a multiset of definitions $\mathcal{D}$ and a multiset of running processes $\mathcal{P}$.

As in the distributed Join Calculus, locations are structured as a tree, and each location has a unique static name. Since the chemical semantics acts on a flat structure of running locations, the tree structure is reflected in the names of the running locations: a location has name φa if its name is a and if the path from the root of the location tree to this location is φ.

In order to account for the different routings of static and dynamic messages, we split the routing in two steps (much as in [12]). The first step, called the *name lookup* step, resolves the location where to route the message, and prepends this location to the message. The destination is the location containing the definition the message is bound to. For static channels, it is the unique location defining the channel; for dynamic channels, it is the closest enclosing location containing a definition of the channel. The second step is the *communication* step, it corresponds to the migration of the resolved message to its destination (rule COMM). We remark that our semantics only focuses on name lookup and does not deal with the actual routing of messages (rule COMM) or locations (rule GO), as in [20].

To simplify the syntax and the semantics, as well as to show that the name lookup of dynamic channels is local, each running location bears a lookup function F from dynamic channels to the the name of the closest enclosing location defining the channel. This function is used in the dynamic name lookup rule (NL-DYN), where $\perp$ is the undefined location. The static name lookup rule (NL-STAT) resolves the unique location defining the channel. Since the name lookup step involves only one location, we let the programmer define a continuation to unresolved messages that is spawned when the lookup occurs (rules NL-STAT and NL-DYN). Section 3 has an example showing the usefulness of this feature. However, we remark that the delivery and consumption of the message are asynchronous. In the following we may write $n\langle\widetilde{m}\rangle$ for $n\langle\widetilde{m}\rangle;\mathbf{0}$.

A definition has the form $n_1\langle\widetilde{y_1}\rangle \mid \ldots \mid n_k\langle\widetilde{y_k}\rangle \triangleright P$ where the n_i are the channel names, the $\widetilde{y_i}$ are the received names, and P is the guarded process. A channel name in a join pattern may either be of the form $\mathtt{n}_i$, if the channel is static, or $\mathbf{n}_i$ if it is dynamic, or y_i if it is a variable (the type system guarantees that these variables may only be substituted to dynamic names, and that they do not occur in evaluation context). A definition is triggered when among the running processes of the location there are messages on each of the n_i. These messages are consumed, and the guarded process is spawned, replacing the formal names (the received names) by the arguments of the messages using the substitution σ_{rn} (as described in the JOIN rule). Note that we use a slightly different JOIN rule: since only resolved messages may be consumed, we write $a.J$ for the join pattern where every message pattern has the prefix a (*i.e.* $a.(J \mid J') = a.J \mid a.J'$). New definitions are introduced using the $\mathtt{def}\ D\ \mathtt{in}\ P$ construct, where the defined static names of D have scope D and P. New dynamic channels are introduced using the $\nu\mathbf{n}.P$ construct.

Locations, either folded or running, gather in the set Δ the dynamic channels they *define*, and in the set I the dynamic channels they *import* (*i.e.* the dynamic names they require to be defined

$$\frac{\mathcal{S} =_\alpha \mathcal{S}'}{\mathcal{S} \equiv \mathcal{S}'}\ [\textsc{Str-}\alpha] \qquad [\textsc{Str-Loc}]\ \frac{\forall \psi \in \mathit{loc}(\mathcal{S}), a \notin \psi \qquad G = \mathit{Lookup}(F, I, \Delta, a)}{a[\mathcal{D} : \mathcal{P}]^{\Delta,I} \vdash_{\varphi}^{\Delta',I',F} \ \equiv\ \vdash_{\varphi}^{\Delta',I',F} \ \|\ \mathcal{D} \vdash_{\varphi a}^{\Delta,I,G} \mathcal{P}}$$

Figure 2: Semantics: structural rules

$$\frac{\mathit{dsn}(D) \cap (\mathit{bn}(\mathcal{S}) \cup \mathit{bn}(\mathcal{D}, D) \cup \mathit{bn}(\mathcal{P} \mid P)) = \emptyset \qquad \mathit{dln}(D) \cap \mathit{ddn}(D) = \emptyset}{\mathcal{S} \ \|\ \mathcal{D} \vdash_{\varphi a}^{\Delta,I,F} \mathcal{P} \mid \mathtt{def}\ D\ \mathtt{in}\ P \ \longrightarrow\ \mathcal{S} \ \|\ \mathcal{D}, D \vdash_{\varphi a}^{\Delta,I,F} \mathcal{P} \mid P}\ [\textsc{Def}]$$

$$\frac{\{\mathbf{n}\} \cap (\mathit{bn}(\mathcal{S}) \cup \mathit{bn}(\mathcal{D}) \cup \mathit{bn}(\mathcal{P} \mid P)) = \emptyset}{\mathcal{S} \ \|\ \mathcal{D} \vdash_{\varphi a}^{\Delta,I,F} \mathcal{P} \mid \nu\mathbf{n}.P \ \longrightarrow\ \mathcal{S} \ \|\ \mathcal{D} \vdash_{\varphi a}^{\Delta,I,F} \mathcal{P} \mid P}\ [\textsc{Nu}] \qquad \frac{\mathtt{n} \in \mathit{dln}(b)}{\vdash_{\varphi a}^{\Delta,I,F} \mathtt{n}\langle\widetilde{v}\rangle; P \ \longrightarrow\ \vdash_{\varphi a}^{\Delta,I,F} b.\mathtt{n}\langle\widetilde{v}\rangle \mid P}\ [\textsc{NL-Stat}]$$

$$\frac{F(\mathbf{n}) = b \wedge b \neq \bot}{\vdash_{\varphi a}^{\Delta,I,F} \mathbf{n}\langle\widetilde{v}\rangle; P \ \longrightarrow\ \vdash_{\varphi a}^{\Delta,I,F} b.\mathbf{n}\langle\widetilde{v}\rangle \mid P}\ [\textsc{NL-Dyn}] \qquad \frac{\mathit{dom}(\sigma_{rn}) = \mathit{rn}(J)}{J \triangleright P \vdash_{\varphi a}^{\Delta,I,F} a.J\sigma_{rn} \ \longrightarrow\ J \triangleright P \vdash_{\varphi a}^{\Delta,I,F} P\sigma_{rn}}\ [\textsc{Join}]$$

$$\frac{}{\vdash_{\varphi a}^{\Delta_a,I_a,F_a} b.n\langle\widetilde{v}\rangle \ \|\ \vdash_{\psi b}^{\Delta_b,I_b,F_b} \ \longrightarrow\ \vdash_{\varphi a}^{\Delta_a,I_a,F_a} \ \|\ \vdash_{\psi b}^{\Delta_b,I_b,F_b} b.n\langle\widetilde{v}\rangle}\ [\textsc{Comm}]$$

$$\frac{}{a[\mathcal{D} : \mathcal{P} \mid \mathsf{go}\ b; Q]^{\Delta_a,I_a} \vdash_{\varphi}^{\Delta,I,F} \ \|\ \vdash_{\psi b}^{\Delta_b,I_b,F_b} \ \longrightarrow\ \vdash_{\varphi}^{\Delta,I,F} \ \|\ a[\mathcal{D} : \mathcal{P} \mid Q]^{\Delta_a,I_a} \vdash_{\psi b}^{\Delta_b,I_b,F_b}}\ [\textsc{Go}]$$

Figure 3: Semantics: reduction rules

in enclosing locations).

When a process $\mathtt{go}(b); P$ is evaluated, the current location as well as all its sublocations migrate to location b (rule Go).

Some running location φa may be folded in its parent location φ (for subsequent migration, for instance) using rule Str-Loc. The first condition of this rule insures that there is no running sublocation of φa, in order to preserve the tree structure (in the original presentation of the distributed Join Calculus, a was said to be *frozen*). Conversely, when unfolding a location, its lookup function needs to be computed using the operator *Lookup*, that takes the lookup function of the enclosing location and patches it to correspond to the current location.

Definition 2.1 (*Lookup* operator) *The operator Lookup takes the lookup function F of the enclosing location, the imported dynamic names I, the locally defined dynamic names Δ, and the name of the current location a, to create a lookup function $G = \mathit{Lookup}(F, I, \Delta, a)$ that associates the current location to locally defined dynamic names and the result of the enclosing lookup function for imported names (we write $\bot$ for the undefined location).*

We have:

$$G(n) = \begin{cases} a & n \in \Delta \\ F(n) & n \notin \Delta \wedge n \in I \\ \bot & n \notin \Delta \wedge n \notin I \end{cases}$$

We define α-conversion as in the Join Calculus (renaming of defined static names bound by a `def` and renaming of received names bound by join patterns), with the additional renaming of dynamic names bound by a ν operator.

In the following, we only consider a restricted class of processes: every location must have a unique name; every defined static name is defined in a single location; join patterns are linear, *i.e.* no defined name nor received name may occur more than once in a given join pattern; free and bound names are distinct ($\mathit{fn}(\mathcal{S}) \cap \mathit{bn}(\mathcal{S}) = \emptyset$). We call this last condition the *hygienic* condition.

The condition of rule Def enforces the preservation of the hygienic condition. Since it must be true before the reduction, the defined names of D, that are free afterward, were bound. Thus they could not occur free in the initial configuration and the reduction cannot capture free names.

The condition thus simply checks that these names are not bound in the final configuration. This rule also checks that no defined local name of D is a dynamic name (well typed definitions satisfy this property). The hygienic condition is also enforced through rule NU.

In figures 2 and 3, only rules DEF and NU explicitly mention the context. In the other rules, the other running locations, definitions, and running processes in the locations involved in the reduction are left implicit.

The structural equivalence $\equiv$ is the smallest reflexive, symmetric and transitive relation generated by rules of figure 2, with the parallel operator "|" (resp. the definition composition operator ",") being associative, commutative and having $\mathbf{0}$ (resp. $\top$) as neutral element. The reduction relation $\rightarrow$ is the smallest relation generated by rules of figure 3 such that $\equiv\rightarrow\equiv\ \subseteq\ \rightarrow$. We recall that most of these rules include implicit contexts.

3 Discussion and examples

Rules NL-STAT and NL-DYN might seem to significantly depart from the asynchronous nature of messages in the distributed Join Calculus, as they spawn the continuation of a message. In fact, messages are still asynchronous since it is not possible to detect when they reach their destination, but it is useful to wait for completion of the name resolution of a dynamic message before migrating, since migration modifies the dynamic binding.

For instance, a process that needs to print on two different locations a and b (representing for instance different machines) using a dynamic channel **print** could be coded as:

$$\mathtt{go}\ a; \mathbf{print}\langle\text{“Hello”}\rangle; \mathtt{go}\ b; \mathbf{print}\langle\text{“World”}\rangle$$

Supposing that no other process makes the current location migrate, this process will successively spawn the two resolved messages $a.\mathbf{print}\langle\text{“Hello”}\rangle$ and $b.\mathbf{print}\langle\text{“World”}\rangle$. The actual consumption of the messages cannot be detected. The only guarantee is that the first message will go to a and the second to b.

An important issue of our design is to restrict to local synchronization. As name lookup only changes in rule STR-LOC, the lookup function is computed only for frozen locations. Since frozen locations have no active sublocation, the computation is local to the location that is being dissolved. Sublocations of this location will inherit the lookup function when dissolving.

Another important issue is the introduction of new definitions for dynamic channels. The second condition for rule DEF of figure 3 requires that all defined local names introduced by a `def` construct are static. However, dynamic definitions may occur in a location $a[D : P]^{\Delta,I}$ inside a `def`. Thus, the only way to introduce new definitions for dynamic channels is to introduce them in new locations. The reason behind this design choice is the following: since dynamic rebinding occurs when migrating, the definition bound to a given dynamic name should not change when no migration occurs. As the definition bound to a dynamic name is in the closest enclosing location, introducing definitions for dynamic channels in running locations would break the *rebinding-only-when-migrating* design. New definitions for dynamic names become active only after applying rule STR-LOC.

Another design choice is the explicit specification of the import interface in locations. Let us suppose that migration is possible only when the import interface is included in the set of dynamic channels available in the destination location (by dynamic checking or static type checking). On the one hand, a larger import interface gives access to more dynamic definitions, and transparently makes them available for sublocations. Locations requiring these definitions may migrate to this location, which can be considered as a *server* or a *runtime*. On the other hand, a smaller interface makes migration to many locations possible since fewer dynamic definitions need to be provided by the destination (in other words, be a *client* or an *agent*). Since these two goals are in opposition, the import interface is explicit.

Our first example describes the interaction between definitions and locations, and illustrates how to transparently update the definitions used by some running clients. More precisely, we model an agent a using a dynamic channel $\mathbf{n}$ in combination with a version manager c controlling the

$$v_1[\mathbf{n}\langle\rangle \triangleright V_1,\ c[\mathtt{nv}\langle b\rangle \triangleright \mathtt{go}\ b,\ a[\mathcal{D}:\mathcal{P}]^{\emptyset,\{\mathbf{n}\}} : \mathbf{0}]^{\emptyset,\{\mathbf{n}\}} : \mathbf{0}]^{\{\mathbf{n}\},\emptyset}\ ,\ v_2[\mathbf{n}\langle\rangle \triangleright V_2 : \underline{\mathtt{nv}\langle v_2\rangle}]^{\{\mathbf{n}\},\emptyset} \quad (1)$$

$$\textsc{Nl-Stat} \rightarrow v_1[\mathbf{n}\langle\rangle \triangleright V_1,\ c[\mathtt{nv}\langle b\rangle \triangleright \mathtt{go}\ b,\ a[\mathcal{D}:\mathcal{P}]^{\emptyset,\{\mathbf{n}\}} : \mathbf{0}]^{\emptyset,\{\mathbf{n}\}} : \mathbf{0}]^{\{\mathbf{n}\},\emptyset}\ ,\ v_2[\mathbf{n}\langle\rangle \triangleright V_2 : \underline{c.\mathtt{nv}\langle v_2\rangle}]^{\{\mathbf{n}\},\emptyset} \quad (2)$$

$$\textsc{Comm} \rightarrow v_1[\mathbf{n}\langle\rangle \triangleright V_1,\ c[\underline{\mathtt{nv}\langle b\rangle \triangleright \mathtt{go}\ b},\ a[\mathcal{D}:\mathcal{P}]^{\emptyset,\{\mathbf{n}\}} : \underline{c.\mathtt{nv}\langle v_2\rangle}]^{\emptyset,\{\mathbf{n}\}} : \mathbf{0}]^{\{\mathbf{n}\},\emptyset}\ ,\ v_2[\mathbf{n}\langle\rangle \triangleright V_2 : \mathbf{0}]^{\{\mathbf{n}\},\emptyset} \quad (3)$$

$$\textsc{Join} \rightarrow v_1[\mathbf{n}\langle\rangle \triangleright V_1,\ c[\mathtt{nv}\langle b\rangle \triangleright \mathtt{go}\ b,\ a[\mathcal{D}:\mathcal{P}]^{\emptyset,\{\mathbf{n}\}} : \underline{\mathtt{go}\ v_2}]^{\emptyset,\{\mathbf{n}\}} : \mathbf{0}]^{\{\mathbf{n}\},\emptyset}\ ,\ v_2[\mathbf{n}\langle\rangle \triangleright V_2 : \mathbf{0}]^{\{\mathbf{n}\},\emptyset} \quad (4)$$

$$\textsc{Go} \rightarrow v_1[\mathbf{n}\langle\rangle \triangleright V_1 : \mathbf{0}]^{\{\mathbf{n}\},\emptyset}\ ,\ v_2[\mathbf{n}\langle\rangle \triangleright V_2,\ c[\mathtt{nv}\langle b\rangle \triangleright \mathtt{go}\ b,\ a[\mathcal{D}:\mathcal{P}]^{\emptyset,\{\mathbf{n}\}} : \mathbf{0}]^{\emptyset,\{\mathbf{n}\}} : \mathbf{0}]^{\{\mathbf{n}\},\emptyset} \quad (5)$$

Figure 4: Dynamic version update

definition of **n** that should be used by a. We suppose that a and c import **n** and define nothing. At the beginning, the agent uses version V_1, defined in location v_1, that defines **n** and imports nothing:

$$v_1[\mathbf{n}\langle\rangle \triangleright V_1,\ c[\mathtt{nv}\langle b\rangle \triangleright \mathtt{go}\ b,\ a[\mathcal{D}:\mathcal{P}]^{\emptyset,\{\mathbf{n}\}} : \mathbf{0}]^{\emptyset,\{\mathbf{n}\}} : \mathbf{0}]^{\{\mathbf{n}\},\emptyset}$$

If a bug is discovered in V_1, it can be fixed by creating a new version of the library V_2 in location v_2, and sending a request to the version manager to use it:

$$v_2[\mathbf{n}\langle\rangle \triangleright V_2 : \mathtt{nv}\langle v_2\rangle]^{\{\mathbf{n}\},\emptyset}$$

Figure 4 details the reduction steps implementing the version change. The initial configuration is (1). First the static message $\mathtt{nv}\langle v_2\rangle$ is resolved using rule NL-STAT, to yield configuration (2). The resulting resolved message $c.\mathtt{nv}\langle v_2\rangle$ is then sent to c using rule COMM, yielding configuration (3). This message then triggers the only join pattern of location c using rule JOIN, creating a $\mathtt{go}\ v_2$ process, in configuration (4). This results in the migration of c to v_2 using rule GO to yield the final state (5). Agent a now uses the new version of the library. The transition to this version was transparent for a as it was not stopped and did not need to collaborate.

A second example illustrates the possibility of mixing static and dynamic channels in join patterns. Resuming the **print** example of the introduction, here is the first implementation for black and white printers (the channel p takes a color and a string, and prints the string with this color on the local printer):

$$s_1[\mathbf{print}\langle s\rangle \triangleright \mathtt{p}\langle\text{“black”}, s\rangle : \mathbf{0}]^{\{\mathbf{print}\},\emptyset}$$

The improved version, for color printers, lets the color be changed for printing, using a simple reference cell called `color`:

$$s_2\left[\begin{array}{l}\mathbf{print}\langle s\rangle \mid \mathtt{color}\langle c\rangle \triangleright \mathtt{p}\langle c, s\rangle \mid \mathtt{color}\langle c\rangle,\\ \mathbf{setcolor}\langle c', \kappa\rangle \mid \mathtt{color}\langle c\rangle \triangleright \mathtt{color}\langle c'\rangle \mid \kappa\langle\rangle\end{array} : \mathtt{color}\langle\text{“black”}\rangle\right]^{\{\mathbf{print},\mathbf{setcolor}\},\emptyset}$$

An agent using the later version could be written as:

$$agent\left[\begin{array}{ll}\mathtt{hi}\langle a\rangle \triangleright & \mathtt{def}\ \kappa\langle\rangle \triangleright \mathbf{print}\langle\text{“hello”}\rangle\ \mathtt{in}\\ & \mathtt{go}\ a; \mathbf{setcolor}\langle\text{“red”}, \kappa\rangle\end{array} : \mathbf{0}\right]^{\emptyset,\{\mathbf{print},\mathbf{setcolor}\}}$$

Calling $\mathtt{hi}\langle s_2\rangle$ would result in the expected behavior. However, if by mistake the call is $\mathtt{hi}\langle s_1\rangle$, the subsequent call to **setcolor** will freeze the agent, as this dynamic channel is not available. This example is continued in section 4, where we describe how the type system guarantees the presence of a definition of **setcolor**.

Our third example shows the usefulness of dynamic channel creation and of dynamic channels as first class values. Let $\mathtt{reg}_1, \ldots, \mathtt{reg}_n$ be channels that can be used to register copies of services $P_1, \ldots, P_n$ and bind them to a name received as argument:

$$\mathtt{reg}_i\langle n, \kappa\rangle \triangleright \mathtt{def}\ a[n\langle\rangle \triangleright P_i : \mathbf{0}]^{\{n\},\emptyset}\ \mathtt{in}\ \kappa\langle a\rangle$$

$$
\begin{array}{llll}
\tau & ::= & & \text{simple type} \\
 & & \widetilde{\tau} & \text{tuple} \\
 & | & \langle\tau\rangle_w^+ & \text{redefinable channel} \\
 & | & \langle\tau\rangle_{\mathbf{\Delta}} & \text{channel} \\
 & | & loc(\mathbf{\Delta}) & \text{location} \\
 & | & \alpha & \text{type variable} \\
 \\
\sigma & ::= & & \text{type scheme} \\
 & & \forall\widetilde{\alpha}\widetilde{\delta}.\tau & \text{type scheme}
\end{array}
\qquad
\begin{array}{llll}
w & ::= & & \text{dynamic name type} \\
 & & \mathbf{n} & \text{dynamic channel name} \\
 & | & \delta & \text{name type variable} \\
 \\
\mathbf{\Delta}, \mathbf{I} & ::= & & \text{sets of dynamic name types} \\
 & & \emptyset & \text{empty set} \\
 & | & \{w\} & \text{singleton} \\
 & | & \mathbf{\Delta} \cup \mathbf{\Delta} & \text{union}
\end{array}
$$

Figure 5: Types

For instance, such channels could be used to bind some local accounting process to a private service port. As accounting may be decentralized in a distributed network, each location may be requested to create a local accounting server associated to the private port.

The following process creates a new service port $\mathbf{n}$, and successively registers this name using the channels $\mathtt{reg}_1, \ldots, \mathtt{reg}_n$, creating local accounting servers $a_1, \ldots, a_n$.

$$
\begin{array}{ll}
\nu\mathbf{n}. & \mathtt{def}\ \kappa_1\langle a_1\rangle \triangleright \ldots\ \ \mathtt{def}\ \kappa_n\langle a_n\rangle \triangleright Q \\
 & \mathtt{in}\ \mathtt{reg}_n\langle \mathbf{n}, \kappa_n\rangle \ldots\ \mathtt{in}\ \mathtt{reg}_1\langle \mathbf{n}, \kappa_1\rangle
\end{array}
$$

The process Q may freely roam between the servers and use the service through the private port $\mathbf{n}$ independently of the associated accounting processes.

We see in this example that dynamic channels are first class values that may be redefined and that may be created at runtime with the ν operator.

As in the previous example, if the process Q erroneously migrates to a location that does not define $\mathbf{n}$, then any subsequent call to $\mathbf{n}$ will be stuck until a migration to one of the a_i occurs. Our type system will rule out such errors by ensuring that requested dynamic definitions are available before migration can occur.

4 Safe dynamic binding

We describe a type system that allows only configurations where dynamic messages are bound to a dynamic definition in some enclosing location. Types are defined in figure 5.

This type system is similar to the one for the distributed Join Calculus. It uses the same generalization criterion as the one implemented in JoCaml and formalized in [12].

Types and subtyping (fig. 5 and 6)

Name variable types δ occur in the types of processes guarded by a join pattern, and represent a dynamic channel name (as a name variable y represents a channel or location name).

Intuitively, locations provide dynamic definitions, either by directly defining them, or by requesting enclosing locations to define them. The type of a location is $loc(\mathbf{\Delta})$, where $\mathbf{\Delta}$ is the set of dynamic names that are available in the location. Thus, inside such a location a message sent on a dynamic name of $\mathbf{\Delta}$ is correct, whereas a message sent on any other dynamic name should be considered as incorrectly typed.

The type of dynamic channel names reflects the required dynamic definitions. A message on a channel of type $\langle\tau\rangle_{\mathbf{\Delta}}$ carries an argument of type τ, and requires the availability of definitions for the names of $\mathbf{\Delta}$. For instance, a dynamic channel $\mathbf{n}$ that does not carry any argument has type $\langle\rangle_{\{\mathbf{n}\}}$, since a message on such a channel requires a definition for $\mathbf{n}$ to be available. Since static channels do not require the presence of any dynamic definition in enclosing locations, their type is of the form $\langle\tau\rangle_{\emptyset}$ which is simply written $\langle\tau\rangle$.

Since channel names are first class values, it is possible to write a definition such as $\mathtt{send}\langle x\rangle \triangleright x\langle\rangle$, that receives a name and sends a message on it. Any use of `send` with a static name is correct, and using `send` with a dynamic name is correct only if the dynamic name is defined in an enclosing location. In order to represent this behavior, we say that the type of the argument of `send` is $\langle\rangle_{\mathbf{\Delta}}$

$$\frac{}{\langle\tau\rangle_w^+ \leq \langle\tau\rangle_{\{w\}}}\ [\text{PLUS}] \qquad \frac{\tau' \leq \tau \qquad \mathbf{\Delta} \subseteq \mathbf{\Delta}'}{\langle\tau\rangle_{\mathbf{\Delta}} \leq \langle\tau'\rangle_{\mathbf{\Delta}'}}\ [\text{N-SUB}] \qquad \frac{\mathbf{\Delta}' \subseteq \mathbf{\Delta}}{loc(\mathbf{\Delta}) \leq loc(\mathbf{\Delta}')}\ [\text{L-SUB}]$$

Figure 6: Subtyping rules

if $\mathbf{\Delta}$ is the set of available dynamic channels. Thus `send` has the type $\langle\langle\rangle_{\mathbf{\Delta}}\rangle$. As $\mathbf{\Delta}$ represents an upper bound of the definitions that may be used, we have an immediate notion of subtyping (written $\leq$) on dynamic channels: $\langle\tau\rangle_{\mathbf{\Delta}'} \leq \langle\tau\rangle_{\mathbf{\Delta}}$ if $\mathbf{\Delta}' \subseteq \mathbf{\Delta}$; a channel that may access fewer dynamic definitions is a subtype of a channel that may access more dynamic definitions. Thus a static channel has a type that is a subtype of any dynamic channel carrying the same type of arguments. The subtyping rule N-SUB of figure 6 also introduces contravariant subtyping on the argument type.

Since a location that provides more dynamic definitions may be used instead of one that provides fewer dynamic definitions, we introduce a notion of subtyping on location types, in rule L-SUB.

One important condition for insuring soundness of the type system is to forbid subtyping on dynamic names that are redefined. We write $\langle\tau\rangle_w^+$ for the type of these channels. Since a redefinable channel may be used instead of a plain dynamic channel for message sending, we introduce the PLUS subtyping rule. In the following we consider $\leq$ to be the smallest reflexive transitive closure generated by the rules of figure 6.

A type scheme $\forall\widetilde{\alpha}\widetilde{\delta}.\tau$ is composed of generalized type variables $\widetilde{\alpha}$, generalized name variables $\widetilde{\delta}$, and type τ. An instantiation of this type scheme, written $Inst(\forall\widetilde{\alpha}\widetilde{\delta}.\tau)$, is a type $\tau\theta$, where θ is a substitution from the type variables $\widetilde{\alpha}$ to types and from the name variable types $\widetilde{\delta}$ to dynamic name types (either name type variables or dynamic channel names).

For soundness reasons, we do not allow polymorphic redefinable channel types. Similarly, we do not allow polymorphic location types. All other types are said to be *well formed*.

A *type environment* B is an association map between names and types, where each name occurs at most once. A *type scheme environment* A or Γ is an association map between names and type schemes, where each name occurs at most once.

In the following, we call $ftv(\tau)$ the free type variables in τ, $fnv(\tau)$ the free name type variables in τ (where the only interesting rules are: $fnv(\langle\tau\rangle_{\mathbf{\Delta}}) = \{\delta, \delta \in \mathbf{\Delta}\} \cup fnv(\tau)$, $fnv(\langle\tau\rangle_\delta^+) = \{\delta\} \cup fnv(\tau)$, $fnv(\langle\tau\rangle_{\mathbf{d}}^+) = fnv(\tau)$, and $fnv(loc(\mathbf{\Delta})) = \{\delta, \delta \in \mathbf{\Delta}\}$). We write $fv(\tau)$ for $ftv(\tau) \cup fnv(\tau)$.

We extend ftv and fnv to type environments and to type scheme environments in the following way:

$$ftv(A) \stackrel{\text{def}}{=} \bigcup_{m:\forall\widetilde{\alpha}\widetilde{\delta}.\tau \in A} ftv(\tau) \setminus \widetilde{\alpha} \qquad fnv(A) \stackrel{\text{def}}{=} \bigcup_{m:\forall\widetilde{\alpha}\widetilde{\delta}.\tau \in A} (fnv(\tau) \setminus \widetilde{\delta})$$

Generalization In the following typing rules, we use a *generalization* operator $Gen(B, \Lambda, \Theta)$ defined as:

$$Gen(B, \Lambda, \Theta) = \bigcup_{m:\tau \in B} \{m : \forall\widetilde{\alpha}\widetilde{\delta}.\tau\}$$

where $\widetilde{\alpha} = ftv(\tau) \setminus (\Lambda \cup \Theta)$ and $\widetilde{\delta} = fnv(\tau) \setminus (\Lambda \cup \Theta)$.

The set Λ contains the names and type variables that occur in the typing environment (in typing rule DEF, Λ is $fv(\Gamma)$); the set Θ contains the names and type variables that may not be generalized because they are shared in a join pattern (as in [12]).

Typing judgments A typing judgment has one of the following forms:

$\Gamma \Vdash \widetilde{n} : \widetilde{\tau}$	name tuple	$\mathbf{\Delta}; \mathbf{I}; \Gamma \Vdash P$	process
$\mathbf{\Delta}; \mathbf{I}; \Gamma \Vdash \mathcal{D} :: B; \Delta_1; \Theta$	definition	$\Gamma \Vdash \mathcal{S}$	configuration
$\mathbf{\Delta}; \mathbf{I}; \Gamma \Vdash \mathcal{D} : \Delta_1$	definition of running location		

where:

- Γ is the type scheme environment;
- $\mathbf{\Delta}$ is a set of dynamic name types, the dynamic channels defined in the current location;
- $\mathbf{I}$ is a set of dynamic names, the dynamic channels imported by the current location;
- B gathers the types of the static defined names of $\mathcal{D}$;
- Δ_1 collects the dynamic channels locally defined by $\mathcal{D}$ (without looking inside sublocations);
- Θ is a set of type variables and names that cannot be generalized.

The four main typing rules to guarantee the presence of a dynamic definition for every dynamic message are Msg, Loc, Soup-Loc, and Go. Rule Msg checks that the channel used does not require more dynamic definitions than the ones available locally (*i.e.* the ones specified in the left-hand side of the judgment: $\mathbf{\Delta}$ and $\mathbf{I}$). Rules Loc and Soup-Loc are very similar, the former being more complex as it can occur in the process guarded by a join pattern. These rules check that the specified dynamic channels are defined and that the imported dynamic channels are available locally (in rule Configuration for the Soup-Loc case). The contents of the location are typed in an environment where the defined and imported dynamic channels are available (these rules modify $\mathbf{\Delta}$ and $\mathbf{I}$ in the left hand side of the typing judgment). Rule Go checks that the target of the migration provides at least the dynamic channels imported by the current location.

Typing rules for names (fig. 7) These rules are as usual.

$$\frac{m : \forall\widetilde{\alpha}\widetilde{\delta}.\tau' \in \Gamma \qquad \tau = \mathit{Inst}(\forall\widetilde{\alpha}\widetilde{\delta}.\tau')}{\Gamma \Vdash m : \tau}\ [\textsc{Name}] \qquad\qquad \frac{\Gamma \Vdash m : \tau \qquad \tau \leq \tau'}{\Gamma \Vdash m : \tau'}\ [\textsc{Sub}]$$

$$\frac{\Gamma \Vdash m_i : \tau_i \text{ for } i \in [1..n]}{\Gamma \Vdash m_1, \ldots, m_n : \tau_1, \ldots, \tau_n}\ [\textsc{Tuple}]$$

Figure 7: Typing rules for names

$$\frac{\Gamma \Vdash n : \langle\widetilde{\tau}\rangle_{\mathbf{\Delta}\cup\mathbf{I}} \qquad \Gamma \Vdash \widetilde{m} : \widetilde{\tau} \qquad \mathbf{\Delta};\mathbf{I};\Gamma \Vdash P}{\mathbf{\Delta};\mathbf{I};\Gamma \Vdash n\langle\widetilde{m}\rangle; P}\ [\textsc{Msg}]$$

$$\frac{\Gamma \Vdash n : \langle\widetilde{\tau}\rangle_{\mathbf{\Delta}\cup\mathbf{I}} \qquad \Gamma \Vdash \widetilde{m} : \widetilde{\tau} \qquad \Gamma \Vdash a : \mathit{loc}(\emptyset) \qquad n \in \mathit{dln}(a)}{\mathbf{\Delta};\mathbf{I};\Gamma \Vdash a.n\langle\widetilde{m}\rangle}\ [\textsc{R-Msg}]$$

$$\frac{\mathbf{\Delta};\mathbf{I};\Gamma + A \Vdash D :: B;\emptyset;\Theta \qquad A = \mathit{Gen}(B, \mathit{fv}(\Gamma), \Theta) \qquad \mathbf{\Delta};\mathbf{I};\Gamma + A \Vdash P \qquad \mathit{dom}(A) \cap \mathit{dom}(\Gamma) = \emptyset}{\mathbf{\Delta};\mathbf{I};\Gamma \Vdash \mathtt{def}\ D\ \mathtt{in}\ P}\ [\textsc{Def}]$$

$$\frac{\mathbf{\Delta};\mathbf{I};\Gamma \Vdash \mathcal{P}_1 \qquad \mathbf{\Delta};\mathbf{I};\Gamma \Vdash \mathcal{P}_2}{\mathbf{\Delta};\mathbf{I};\Gamma \Vdash \mathcal{P}_1 \mid \mathcal{P}_2}\ [\textsc{Par}] \qquad\qquad \frac{\Gamma \Vdash n : \mathit{loc}(\mathbf{I}) \qquad \mathbf{\Delta};\mathbf{I};\Gamma \Vdash P}{\mathbf{\Delta};\mathbf{I};\Gamma \Vdash \mathtt{go}\ n; P}\ [\textsc{Go}]$$

$$\frac{\mathbf{d} \notin \mathit{fn}(\Gamma) \qquad \mathbf{\Delta};\mathbf{I};\Gamma + \mathbf{d} : \langle\tau\rangle^{+}_{\mathbf{d}} \Vdash P}{\mathbf{\Delta};\mathbf{I};\Gamma \Vdash \nu\mathbf{d}.P}\ [\textsc{Nu}] \qquad\qquad \frac{}{\mathbf{\Delta};\mathbf{I};\Gamma \Vdash \mathbf{0}}\ [\textsc{Nil}]$$

Figure 8: Typing rules for processes

Typing rules for processes (fig. 8) The two typing rules for messages Msg and R-Msg are very similar, and follow the different states of a message as it is first resolved, then sent to its destination. In these rules, the name on which the message is sent needs to satisfy the typing

$$\text{[JOIN]}\ \frac{\begin{array}{c}\mathbf{\Delta};\mathbf{I};\Gamma + (\widetilde{u_i} : \widetilde{\tau_i})^i + (\widetilde{y_j} : \widetilde{\tau_j})^j \Vdash P \qquad \Gamma \Vdash \mathtt{x_i} : \langle \widetilde{\tau_i} \rangle \qquad \Gamma \Vdash m_j : \langle \widetilde{\tau_j} \rangle^+_{w_j} \qquad \left(\bigcup_i \widetilde{u_i} \cup \bigcup_j \widetilde{y_j} \right) \cap dom(\Gamma) = \emptyset \\ \forall (n : \tau), (n' : \tau') \in \left(\{\mathtt{x_i} : \langle \widetilde{\tau_i} \rangle\} \cup \{m_j : \langle \widetilde{\tau_j} \rangle^+_{w_j}\} \right) \ .\ n \neq n' \implies fv(\tau) \cap fv(\tau') \subseteq \Theta\end{array}}{\mathbf{\Delta};\mathbf{I};\Gamma \Vdash (\mathtt{x_i}\langle \widetilde{u_i} \rangle)^i \mid (m_j \langle \widetilde{y_j} \rangle)^j \triangleright P :: (\mathtt{x_i} : \langle \widetilde{\tau_i} \rangle)^i; \bigcup_j \{m_j\}; \Theta}$$

$$\text{[TOP]}\ \frac{}{\mathbf{\Delta};\mathbf{I};\Gamma \Vdash \top :: \emptyset; \emptyset; \Theta} \qquad \text{[AND]}\ \frac{\mathbf{\Delta};\mathbf{I};\Gamma \Vdash \mathcal{D}_1 :: B_1; \Delta_1; \Theta \qquad \mathbf{\Delta};\mathbf{I};\Gamma \Vdash \mathcal{D}_2 :: B_2; \Delta_2; \Theta}{\mathbf{\Delta};\mathbf{I};\Gamma \Vdash \mathcal{D}_1, \mathcal{D}_2 :: B_1 \oplus B_2; \Delta_1 \cup \Delta_2; \Theta}$$

$$\text{[LOC]}\ \frac{\begin{array}{c}\forall m_i \in \Delta \ .\ m_i : \langle \tau_i \rangle^+_{w_i} \in \Gamma \qquad \forall n_j \in I \ .\ n_j : \langle \tau_j \rangle^+_{w'_j} \in \Gamma \qquad \mathbf{\Delta}' = \bigcup_i w_i \quad \mathbf{I}' = \bigcup_j w'_j \\ \mathbf{\Delta}';\mathbf{I}';\Gamma \Vdash \mathcal{D} :: B; \Delta; \Theta \qquad \mathbf{\Delta}';\mathbf{I}';\Gamma \Vdash \mathcal{P} \qquad \Gamma \Vdash a : loc(\mathbf{\Delta}' \cup \mathbf{I}') \qquad \mathbf{I}' \subseteq (\mathbf{\Delta}'' \cup \mathbf{I}'')\end{array}}{\mathbf{\Delta}'';\mathbf{I}'';\Gamma \Vdash a[\mathcal{D} : \mathcal{P}]^{\Delta, I} :: B + a : loc(\mathbf{\Delta}' \cup \mathbf{I}'); \emptyset; \Theta}$$

Figure 9: Typing rules for definitions

judgment $\Gamma \Vdash n : \langle \widetilde{\tau} \rangle_{\mathbf{\Delta} \cup \mathbf{I}}$. Because of subtyping on channel names, this judgment gives an upper bound on the dynamic names that the message may use, thus on the definitions accessed; this upper bound consists of the available dynamic definitions at this point. These rules make sure that every dynamic message is bound to a definition. The R-MSG rule checks that the location specified in the prefix of the message is present with a location type in Γ, and that the channel name is a defined local name of this location.

The typing rule DEF checks that no local dynamic name is defined in D ($\Delta_1 = \emptyset$), and also checks that the defined static names of D—which are the domain of B and A—do not clash with names of the typing environment. We remark that we use polymorphic recursion here, which drastically simplifies the subject reduction proof for the distributed Join Calculus (see appendix B).

Rule GO requires that the destination location has type $loc(\mathbf{I})$, where $\mathbf{I}$ is the set of imported dynamic channels. By definition of subtyping on location, this set is a lower bound, and any location providing more dynamic definitions may be the target of migration.

Rule NU introduces a new dynamic channel, which is monomorphic, and which has the type of dynamic channels sending messages on their own name. Since every dynamic channel created has a monomorphic redefinable type, every subsequent definition must exactly follow this type.

Typing rules for definitions (fig. 9) The first rule JOIN is used to type one join pattern. The defined names of the join pattern are partitioned into two sets: the static names and the dynamic names. Static names $\mathtt{x_i}$ are given the type $\langle \widetilde{\tau_i} \rangle$, and are collected in the typing environment B. Dynamic names need to be present in Γ, with a redefinable type $\langle \widetilde{\tau_j} \rangle^+_{w_j}$, as they are redefined. Dynamic names are not written $\mathbf{m}_j$ since they may be variables bound by an enclosing join pattern. In evaluation context, we will always have $m_j = w_j = \mathbf{m}_j$. They may however be different if the join pattern occurs in the guarded process of another join pattern that receives a dynamic name and then redefines it. All dynamic names are collected in the set of local dynamic names. The set of non generalized variables Θ is checked to be big enough: any type variable or name that is shared between two types cannot be generalized (as in [12]).

The typing rule AND uses the $\oplus$ operator in $B_1 \oplus B_2$, that requires names that are both in the domain of B_1 and B_2 to have the same type.

The rule LOC extracts from the typing environment the types associated to the names of Δ and I, in order to collect the dynamic name types associated to these channels. As in rule JOIN, in evaluation context the names are the same. They may be different if the typing rule is used to type the guarded process of a join pattern that receives a dynamic name that is redefined. The typing of the definition $\mathcal{D}$ must yield a set of local dynamic names Δ identical to the one declared in the location. The typing of $\mathcal{D}$ and $\mathcal{P}$ may use the available dynamic channels associated to the ones declared by the location being typed. This rule also checks that the imported names are available

$$\frac{\begin{array}{c}\{\varphi_i\} \text{ form a tree with root } b \qquad \forall \psi a \in \{\varphi_i\}. I_{\psi a} \subseteq \Delta_\psi \cup I_\psi \\ I_b = \emptyset \qquad \forall \psi a \in \{\varphi_i\}. F_{\psi a} = \mathit{Lookup}(F_\psi, I_{\psi a}, \Delta_{\psi a}, a) \qquad (\Gamma \Vdash \mathcal{D}_i \vdash_{\varphi_i}^{\Delta_{\varphi_i}, I_{\varphi_i}, F_{\varphi_i}} \mathcal{P}_i)^i\end{array}}{\Gamma \Vdash \prod_i (\mathcal{D}_i \vdash_{\varphi_i}^{\Delta_{\varphi_i}, I_{\varphi_i}, F_{\varphi_i}} \mathcal{P}_i)} \text{[Configuration]}$$

$$\frac{n \in \Delta \cup I \implies n = \mathbf{n} \wedge \mathbf{n} : \langle\tau\rangle^+_{\mathbf{n}} \in \Gamma \qquad a : \mathit{loc}(\Delta \cup I) \in \Gamma \qquad \Delta; I; \Gamma \Vdash \mathcal{P} \qquad \Delta; I; \Gamma \Vdash \mathcal{D} : \Delta}{\Gamma \Vdash \mathcal{D} \vdash_{\varphi a}^{\Delta, I, F} \mathcal{P}} \text{[Soup-Loc]}$$

$$\frac{\boldsymbol{\Delta}; \mathbf{I}; \Gamma \Vdash \mathcal{D} :: B; \Delta_{\mathcal{D}}; \Theta \qquad A = \mathit{Gen}(B, \mathit{fv}(\Gamma), \Theta) \qquad A \subseteq \Gamma}{\boldsymbol{\Delta}; \mathbf{I}; \Gamma \Vdash \mathcal{D} : \Delta_{\mathcal{D}}} \text{[Chem-Def]}$$

Figure 10: Typing rules for configurations

in the enclosing location. In the conclusion of the rule, the set of local dynamic names is empty, since all dynamic names are defined in a sublocation.

Typing rules for configurations (fig. 10) The CONFIGURATION rule checks that the running locations form a tree, all running locations import names that are available in the enclosing location (the root location does not import any name), every name lookup functions is correctly computed, and all running locations are well typed.

The SOUP-LOC rule is used to type a running location. It first checks that all the names declared in Δ and I are dynamic channels (not variables), and that the type of these channels in Γ is the type of redefinable channels sending messages on their own name. It then types the definition and process of the location, using the available dynamic channels declared by the location, and checks that the location is present with the correct type in the typing environment. This rule is similar to rule LOC, although simpler as running locations occur only in evaluation contexts. There is no name variable in Δ (resp. I), thus no name type variable in the corresponding $\boldsymbol{\Delta}$ (resp. $\mathbf{I}$), which are identical.

The CHEM-DEF rule is used to type the definition of a running location. It types the definition and checks that the resulting generalization is present in the environment.

Typing examples A simple example is a forwarder:

$$\texttt{def fwd}\langle n, \mathit{args}\rangle \triangleright n\langle \mathit{args}\rangle \texttt{ in } P$$

If the static channel `fwd` is typed in a location where dynamic names $\mathbf{d}$ and $\mathbf{d}'$ are available, then when typing P `fwd` has the type (by typing rules JOIN and DEF):

$$\texttt{fwd} : \forall \alpha. \langle \langle \alpha \rangle_{\{\mathbf{d}, \mathbf{d}'\}}, \alpha \rangle$$

Thus, the first argument of `fwd` may be any static channel, or a dynamic channel bound to $\mathbf{d}$ or $\mathbf{d}'$, that are the only dynamic channels available locally. A later use of `fwd` as $\texttt{fwd}\langle \mathbf{d}'', m\rangle$, where m is a correct argument for $\mathbf{d}''$ but with $\mathbf{d}''$ different from $\mathbf{d}$ or $\mathbf{d}'$ would result in a typing error, since $\langle\tau\rangle_{\{\mathbf{d}''\}}$ is not a subtype of $\langle\tau\rangle_{\{\mathbf{d}, \mathbf{d}'\}}$.

Continuing the example from the introduction (see page 6), the old version of the **print** server s_1 has type $\mathit{loc}(\{\mathbf{print}\})$. The new version s_2 has type $\mathit{loc}(\{\mathbf{print}, \mathbf{setcolor}\})$. When typing the agent *agent*, the static channel `hi` has type $\langle \mathit{loc}(\{\mathbf{print}, \mathbf{setcolor}\})\rangle$. Thus $\texttt{hi}\langle s_2\rangle$ is well typed, whereas $\texttt{hi}\langle s_1\rangle$ is not, preventing this incorrect use of `hi`.

Dynamic channels may also be redefined, as in the last example of section 3:

$$\texttt{def reg}\langle n, \kappa\rangle \triangleright \texttt{def } a[n\langle\rangle \triangleright P : \mathbf{0}]^{\{n\}, \emptyset} \texttt{ in } \kappa\langle a\rangle \texttt{ in } P$$

when typing P, `reg` has the following type:

$$\texttt{reg} : \forall \delta. \langle \langle\rangle^+_\delta, \langle \mathit{loc}(\{\delta\})\rangle\rangle$$

The channel $\mathtt{reg}$ may receive any redefinable dynamic channel, along with a continuation for sending back a location name defining this channel.

The previous example does not depend on which dynamic channels are available locally. A variant of this example is the channel $\mathtt{r}'$ that not only creates a location defining the dynamic channel, but also uses it:

$$\mathtt{def}\ \mathtt{r}'\langle n,\kappa\rangle \triangleright \mathtt{def}\ a[n\langle\rangle \triangleright P : \mathbf{0}]^{\{n\},\emptyset}\ \mathtt{in}\ \kappa\langle a\rangle \mid n\langle\rangle\ \mathtt{in}\ P$$

If the only dynamic channel available locally is $\mathbf{d}$, the type of $\mathtt{r}'$ is $\langle\langle\rangle^{+}_{\mathbf{d}}, \langle loc(\{\mathbf{d}\})\rangle\rangle$. However, if $\mathbf{d}$ and $\mathbf{d}'$ are available, the type of $\mathtt{r}'$ is either $\langle\langle\rangle^{+}_{\mathbf{d}}, \langle loc(\{\mathbf{d}\})\rangle\rangle$ or $\langle\langle\rangle^{+}_{\mathbf{d}'}, \langle loc(\{\mathbf{d}'\})\rangle\rangle$, but there is no principal type. To express such a type, we would need a more complex type system using bounded quantification and constraints, which is beyond the scope of this paper.

5 Type Soundness

To prove the soundness of our system, we first prove a subject reduction theorem, then we prove a progress property that insures that well-typed configurations do not go wrong.

We first specify *well-formed* typing environments.

Definition 5.1 *A typing environment* Γ *is* well formed *if and only if its types are well formed and every type binding is of the form* $\mathbf{n} : \langle\tau\rangle^{+}_{\mathbf{n}}$, $\mathtt{n} : \forall\widetilde{\alpha}\widetilde{\delta}.\langle\tau\rangle$, *and* $a : loc(\boldsymbol{\Delta})$ *where* $\boldsymbol{\Delta}$ *contains no name type variable.*

We now prove that structural equivalence and the reduction relation preserve typing.

Lemma 5.2 *Let* $\mathcal{S}$ *be a configuration and* $\Gamma \Vdash \mathcal{S}$ *a typing of this configuration where* Γ *is well-formed. If* $\mathcal{S} \equiv \mathcal{S}'$, *there is a well-formed* Γ' *such that* $\Gamma' \Vdash \mathcal{S}'$.

Theorem 1 (Subject reduction) *Let* $\mathcal{S}$ *be a configuration and* Γ *a well formed environment. If* $\Gamma \Vdash \mathcal{S}$ *and* $\mathcal{S} \to \mathcal{S}'$, *then there exists a well-formed environment* Γ' *such that* $\Gamma' \Vdash \mathcal{S}'$.

We remark that Γ and Γ' need not be related, since reduction steps involve configurations including implicit contexts.

We now need to prove that the NL-DYN step resolves messages to the closest enclosing location defining them (where $dyn(b)$ is the set of dynamic names locally defined in b). This proof insures that our way of computing the name lookup function corresponds to our specification.

Lemma 5.3 *Let* $\Gamma \Vdash \mathcal{S}$ *be a typing derivation. For any dissolved location* φa *of* $\mathcal{S}$, *if we have* $F_{\varphi a}(n) = b \neq \bot$, *then we have* $n \in dyn(b)$, $\varphi a = \psi b \psi'$ *with* $\forall c \in \psi'.n \notin dln(c)$.

We define the notion of a *stuck configuration.*

Definition 5.4 *We say that a configuration* $\mathcal{S}$ *is* stuck *when one of the following is true:*

1. *There is a message* $n\langle\widetilde{m}\rangle$ *or* $a.n\langle\widetilde{m}\rangle$ *in evaluation context where* n *is not a channel name;*
2. *there is a* $n \in dn(\mathcal{S})$ *that is not a channel name or a location name;*
3. *there is a process* $\mathsf{go}\ n; P$ *where* n *is not a location name;*
4. *there is a message* $n\langle\widetilde{m}\rangle$ *or* $a.n\langle\widetilde{m}\rangle$ *and a definition of* n *with different arities;*
5. *there is a message* $\mathbf{n}\langle\widetilde{m}\rangle$ *in evaluation context that cannot be reduced by rule* NL-DYN;
6. *there is a message* $a.n\langle\widetilde{m}\rangle$ *with* $n \notin dln(a)$.

We can now state that no well-typed configuration is stuck.

Theorem 2 (Progress) *Let $\mathcal{S}$ be a configuration and Γ a well-formed environment. If $\Gamma \Vdash \mathcal{S}$, then $\mathcal{S}$ is not stuck.*

Combining lemma 5.2, 5.3, theorems 1 and 2, we have the property that well-typed configurations cannot become stuck, thus dynamic messages are always sent to the closest enclosing location providing a definition and there is always such a location.

The proofs of these theorems are fairly complex, especially the substitution lemma, as channel names occur in the types. They are available in the draft of the long version of this paper [18].

6 Conclusion

Future work One of our first priorities is the integration of dynamic channels into JoCaml. A form of dynamic binding at the module level is already present in JoCaml, but is difficult to use. The implementation would provide the programmers with easier management of local resources. As our system was designed with implementation in mind, this should not present any difficulty. In particular, it seems clear that maintaining the lookup function F (that resolves dynamic names to the closest enclosing location defining them) would be cheap because it does not require more locking than the one already present in JoCaml.

On the theoretical side, it is unfortunate that the construct $a.n\langle\widetilde{m}\rangle$, very similar to the message construct of [20], cannot be made available to programmers. To use such a construct soundly, it is necessary to check that n is indeed defined in location a, where a may be a variable bound by an enclosing join pattern. We have been designing such a type system, which is very similar to the one introduced in this paper, and we are finishing the soundness proofs.

Another extension is the design of a type reconstruction algorithm. Since our type system uses polymorphic recursion, which greatly simplifies the proof of subject reduction, a type reconstruction algorithm requires a different type system (see appendix B). This issue is already present in the distributed Join Calculus.

Another extension, related to type reconstruction, is to use a constraint based type system to recover principal types. We are currently adapting the $\mathrm{B}(T)$ framework of [8] to this end.

In the present system, any location that eventually uses a dynamic name that it does not define must import it, and the import interface is static. Thus this name must be defined at all times. An interesting refinement would be to use some sequentiality information (such as the use of a dynamic name only after a given migration) to let the import interface be explicitly modified. Similarly, the export interface of a location is simply the dynamic names it defines and imports. Adding explicit restriction to the export interface, by restricting the type of the location, would let a location better control the resources it exposes.

Related work In this paper, our strategy for handling a message on a dynamic channel name is to resolve in one step the closest enclosing location defining the channel, and then to send the message to this location in an other step. An alternate approach would simply consist in sending a message on a dynamic name that is not defined locally to the parent location, which would then deal with the message (this second incremental approach is similar to [6] and [19]). The two semantics yield two different behaviors but, surprisingly enough, the type system presented in this paper is also sound for the second system.

The primary goal of [1] is to guarantee receptiveness of channel names and deadlock freedom in a π-calculus with localities. The receptiveness property is harder in the π-calculus as receivers may disappear. This implies that our type system is simpler as we do not guarantee deadlock freedom because of join patterns. However, the resulting type system of [1] requires that names passed on channels cannot be used as receivers, unlike our dynamic channels as shown as in the last example of section 3.

Several works deal with resource control (as opposed to resource availability) in a π-calculus with localities and objective rebinding. In [13], localities have a flat structure, and processes may migrate objectively between localities. A type system insures that no agent may access a resource if it was not given the capability to do so. In the *local area calculus* [7], localities form a fixed hierarchy of levels that do not migrate. Channels have a level of operation, meaning that no communication on such a channel may cross the boundary of an higher level area. In the box-π calculus [19], localities also form a fixed hierarchy, and communication may cross only one locality boundary at a time. This calculus aims at controlling the flow of information between localities.

The higher order π-calculus of [22, 21] also deals with access control, by explicitly specifying for each input which resources may be accessed by the input process, distinguishing between read and write accesses. This calculus is objective and does not guarantee the availability of resources (it guarantees a *locality* property for channels, *i.e.* every channel is defined in at most of location). Moreover, this calculus uses dependent types instead of polymorphism with name type variables. Similarly, access control for Klaim [17] deals with access control for objective migration of processes.

The work on secrecy and groups [5] also deals with controlling access to some names through the creation of fresh groups, and the assignment of channels to these groups. However, the different intent (secrecy vs receptiveness) leads to different type systems. In our system, polymorphism lets dynamic names escape the scope in which they are created, allowing the typing of more processes, whereas in [5] secrecy is achieved by checking that groups cannot escape their initial scope.

A very simple calculus of localized resources is the Ambient Calculus [6]. There is a notion of dynamic binding in the Ambient Calculus since several ambients may bear the same name, and as an ambient name a occurring in a capability running in a given ambient b only refers to ambients in b's local environment (parent, siblings, children). Since the local environment of b changes as ambients migrate, the association between the name a and possible targets for the capability also changes. Recent works on the Ambient Calculus add type systems to analyze the behavior of ambients or enforce security policies. In [4], groups are introduced to refine mobility types of ambients in order to control the escape of capabilities. In [2], safe ambients are typed according to a security policy. The type system takes into account the capabilities that the ambient may acquire. Boxed ambients [3] drops the *open* capabilities of the Ambient Calculus, but allows communication between parent and children. A type system allows the analysis of ambient behavior, distinguishing local communications from communications with the context. In these previous ambient calculi, the emphasis is on restricting the behavior of processes, more than checking the availability of resources, moreover there is no notion of static names.

Conclusion We have presented an extension of the distributed Join Calculus that features the creation of definitions accessible through dynamic channels that are rebound at migration. New definitions for a given channel may be created at runtime, and new dynamic channels may be generated as well. A type system that has the subject reduction property was presented, which guarantees the presence of a definitions for any dynamic channel that may be used.

Acknowledgments We would like to thank Sylvain Conchon, Cédric Fournet, James Leifer, Jean-Jacques Lévy, François Pottier, and Didier Rémy for comments.

References

[1] R. M. Amadio, G. Boudol, and C. Lhoussaine. The receptive distributed pi-calculus (extended abstract). In *FST-TCS'99*, LNCS, 1999.

[2] M. Bugliesi and G. Castagna. Secure safe ambients. In *Proceedings of POPL '01*, pages 222–235. ACM Press, 2001.

[3] M. Bugliesi, G. Castagna, and S. Crafa. Boxed ambients. In *TACS'01*, LNCS, 2001.

[4] L. Cardelli, G. Ghelli, and A. D. Gordon. Ambient groups and mobility types. In *IFIP TCS 2000 (Sendai, Japan)*, LNCS. IFIP, Springer, Aug. 2000.

[5] L. Cardelli, G. Ghelli, and A. D. Gordon. Secrecy and group creation. In *CONCUR 2000 (University Park, PA, USA)*, LNCS. Springer, Aug. 2000.

[6] L. Cardelli and A. D. Gordon. Mobile ambients. In *Foundations of Software Science and Computational Structures*. Springer (LNCS), 1998.

[7] T. Chothia and I. Stark. A distributed calculus with local areas of communication. In *Proceedings of HLCL '00*, 2001.

[8] S. Conchon and F. Pottier. JOIN(X): Constraint-Based Type Inference for the Join-Calculus. In *Proceedings of ESOP'01*, LNCS, Apr. 2001.

[9] C. Fournet. *The Join-Calculus: a Calculus for Distributed Mobile Programming*. PhD thesis, Ecole Polytechnique, Palaiseau, Nov. 1998. INRIA, TU-0556.

[10] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In *Proceedings of CONCUR'96*, Aug. 1996. LNCS 1119.

[11] C. Fournet, J.-J. Lévy, and A. Schmitt. A distributed implementation of Ambients. Draft of long version, `http://join.inria.fr/ambients.html`, 1999.

[12] C. Fournet, L. Maranget, C. Laneve, and D. Rémy. Inheritance in the join calculus. In *FST-TCS'00*, LNCS, Dec. 2000.

[13] M. Hennessy and J. Riely. Resource access control in systems of mobile agents. *Information and Computation*, To appear.

[14] N. Kobayashi, S. Saito, and E. Sumii. An implicitly-typed deadlock-free process calculus. In *Proceedings of CONCUR 2000*, LNCS, Aug. 2000.

[15] F. Le Fessant. The JoCAML system prototype. Software and documentation available from `http://pauillac.inria.fr/jocaml`, 1998.

[16] J. Lewis, M. Shields, E. Meijer, and J. Launchbury. Implicit parameters: Dynamic scoping with static types. In *Proceedings POPL 2000*, pages 108–118, Jan 2000.

[17] R. D. Nicola, G. L. Ferrari, R. Pugliese, and B. Venneri. Types for access control. *Theoretical Computer Science*, 240(1):215–254, 2000.

[18] A. Schmitt. Safe Dynamic Binding in the Join Calculus, Draft. Available from `http://pauillac.inria.fr/~aschmitt/publications.html`, 2001.

[19] P. Sewell and J. Vitek. Secure composition of insecure components. In *Proceedings of CSFW 99 (Mordano, Italy)*, June 1999.

[20] A. Unyapoth and P. Sewell. Nomadic Pict: Correct communication infrastructure for mobile computation. In *Proceedings of POPL 2001 (London)*, Jan. 2001.

[21] N. Yoshida and M. Hennessy. Subtyping and locality in distributed higher-order processes. In *Proceedings CONCUR 99, LNCS no 1664*, 1999.

[22] N. Yoshida and M. Hennessy. Assigning types to processes (extended abstract). In *Fifteenth Annual IEEE Symposium on Logic in Computer Science*, 2000.

A Formal sets of names

Definition A.1 *The set of free names fn is defined as:*

$$
\begin{array}{rl}
fn(\mathbf{0}) = \emptyset & fn(\mathcal{P} \mid \mathcal{P}') = fn(\mathcal{P}) \cup fn(\mathcal{P}') \\
fn(n\langle\widetilde{m}\rangle; P) = \{n\} \cup \widetilde{m} \cup fn(P) & fn(\mathtt{go}\ n; P) = \{n\} \cup fn(P) \\
fn(\mathtt{def}\ D\ \mathtt{in}\ P) = (fn(D) \cup fn(P)) \setminus dsn(D) & fn(\nu\mathbf{n}.P) = fn(P) \setminus \{n\} \\
fn(a.n\langle\widetilde{m}\rangle) = \{a\} \cup \{n\} \cup \widetilde{m} & fn(\top) = \emptyset \\
fn(\mathcal{D}, \mathcal{D}') = fn(\mathcal{D}) \cup fn(\mathcal{D}') & fn(J \triangleright P) = (fn(J) \cup fn(P)) \setminus rn(J) \\
fn(a[\mathcal{D} : \mathcal{P}]^{\Delta, I}) = \{a\} \cup \Delta \cup fn(\mathcal{D}) \cup fn(\mathcal{P}) \cup I & fn(n\langle\widetilde{y_i}\rangle) = \{n\} \cup \bigcup_i \{y_i\} \\
fn(J \mid J') = fn(J) \cup fn(J') & fn(\Omega) = \emptyset \\
fn(\mathcal{S} \parallel \mathcal{S}') = fn(\mathcal{S}) \cup fn(\mathcal{S}') & fn(\mathcal{D} \vdash_{\varphi a}^{\Delta, I, F} \mathcal{P}) = fn(\mathcal{D}) \cup \Delta \cup I \cup \{a\} \cup fn(\mathcal{P})
\end{array}
$$

We now define received names rn that occur in join patterns:

$$
rn(n\langle\widetilde{y_i}\rangle) = \bigcup_i \{y_i\} \qquad rn(J \mid J') = rn(J) \cup rn(J')
$$

We now define defined names dn:

$$
\begin{array}{rl}
dn(\top) = \emptyset & dn(\mathcal{D}, \mathcal{D}') = dn(\mathcal{D}) \cup dn(\mathcal{D}') \\
dn(J \triangleright Q) = dn(J) & dn(a[\mathcal{D} : \mathcal{P}]^{\Delta, I}) = \{a\} \cup dn(\mathcal{D}) \\
dn(n\langle\widetilde{y_i}\rangle) = \{n\} & dn(J \mid J') = dn(J) \cup dn(J') \\
dn(\Omega) = \emptyset & dn(\mathcal{S} \parallel \mathcal{S}') = dn(\mathcal{S}) \cup dn(\mathcal{S}') \\
dn(\mathcal{D} \vdash_{\varphi a}^{\Delta, I, F} \mathcal{P}) = dn(\mathcal{D}) \cup \{a\} &
\end{array}
$$

A subset of all defined names is the set of defined static names dsn.

$$
\begin{array}{rl}
dsn(\top) = \emptyset & dsn(\mathcal{D}, \mathcal{D}') = dsn(\mathcal{D}) \cup dsn(\mathcal{D}') \\
dsn(J \triangleright Q) = dsn(J) & dsn(a[\mathcal{D} : \mathcal{P}]^{\Delta, I}) = \{a\} \cup dsn(\mathcal{D}) \\
dsn(\mathtt{n}\langle\widetilde{y_i}\rangle) = \{\mathtt{n}\} & dsn(\mathbf{n}\langle\widetilde{y_i}\rangle) = \emptyset \\
dsn(y\langle\widetilde{y_i}\rangle) = \emptyset & dsn(J \mid J') = dsn(J) \cup dsn(J') \\
dsn(\Omega) = \emptyset & dsn(\mathcal{S} \parallel \mathcal{S}') = dsn(\mathcal{S}) \cup dsn(\mathcal{S}') \\
dsn(\mathcal{D} \vdash_{\varphi a}^{\Delta, I, F} \mathcal{P}) = \{a\} \cup dsn(\mathcal{D}) &
\end{array}
$$

We remark that we do not define $dsn(a\langle\widetilde{y_i}\rangle)$ since they do not occur in well typed processes. Another subset of defined names is the set of defined dynamic names ddn.

$$
\begin{array}{rl}
ddn(\top) = \emptyset & ddn(\mathcal{D}, \mathcal{D}') = ddn(\mathcal{D}) \cup ddn(\mathcal{D}') \\
ddn(J \triangleright Q) = ddn(J) & ddn(a[\mathcal{D} : \mathcal{P}]^{\Delta, I}) = ddn(\mathcal{D}) \\
ddn(\mathtt{n}\langle\widetilde{y_i}\rangle) = \emptyset & ddn(\mathbf{n}\langle\widetilde{y_i}\rangle) = \{\mathbf{n}\} \\
ddn(y\langle\widetilde{y_i}\rangle) = \{y\} & ddn(J \mid J') = ddn(J) \cup ddn(J') \\
ddn(\Omega) = \emptyset & ddn(\mathcal{S} \parallel \mathcal{S}') = ddn(\mathcal{S}) \cup ddn(\mathcal{S}') \\
ddn(\mathcal{D} \vdash_{\varphi a}^{\Delta, I, F} \mathcal{P}) = ddn(\mathcal{D}) &
\end{array}
$$

Another subset of all defined names is the set of defined local names dln (names defined in $\mathcal{D}$ that are not in a sublocation). They are straightforwardly defined on definitions as:

$$dln(\top) = \emptyset \qquad dln(\mathcal{D}, \mathcal{D}') = dln(\mathcal{D}) \cup dln(\mathcal{D}')$$
$$dln(J \triangleright P) = dn(J) \qquad dln(a[\mathcal{D} : \mathcal{P}]^{\Delta, I}) = \emptyset$$

In the following we write $dln(a)$ for $dln(\mathcal{D})$ for a running location $\mathcal{D} \vdash_{\varphi a}^{\Delta, I, F} \mathcal{P}$, or for a folded location $a[\mathcal{D} : \mathcal{P}]^{\Delta, I}$ occurring unguarded in $\mathcal{S}$.

We also define the set of local dynamic names dyn (respectively imported dynamic names imp) for a running location $\mathcal{D} \vdash_{\varphi a}^{\Delta, I, F} \mathcal{P}$ or for folded sublocation $a[\mathcal{D} : \mathcal{P}]^{\Delta, I}$ as $dyn(a) = \Delta$ (respectively $imp(a) = I$).

The set of bound names bn is:

$$\begin{aligned}
bn(\mathbf{0}) &= \emptyset & bn(\mathcal{P} \mid \mathcal{P}') &= bn(\mathcal{P}) \cup bn(\mathcal{P}') \\
bn(n\langle \widetilde{m} \rangle; P) &= bn(P) & bn(\texttt{go}\ n; P) &= bn(P) \\
bn(\texttt{def}\ D\ \texttt{in}\ P) &= dsn(D) \cup bn(D) \cup bn(P) & bn(\nu \mathbf{n}.P) &= \{n\} \cup bn(P) \\
bn(a.n\langle \widetilde{m} \rangle) &= \emptyset & bn(\top) &= \emptyset \\
bn(\mathcal{D}, \mathcal{D}') &= bn(\mathcal{D}) \cup bn(\mathcal{D}') & bn(J \triangleright P) &= rn(J) \cup bn(P) \\
bn(a[\mathcal{D} : \mathcal{P}]^{\Delta, I}) &= bn(\mathcal{D}) \cup bn(\mathcal{P}) & bn(n\langle \widetilde{y_i} \rangle) &= \emptyset \\
bn(J \mid J') &= \emptyset & bn(\Omega) &= \emptyset \\
bn(\mathcal{S} \parallel \mathcal{S}') &= bn(\mathcal{S}) \cup bn(\mathcal{S}') & bn(\mathcal{D} \vdash_{\varphi a}^{\Delta, I, F} P) &= bn(\mathcal{D}) \cup bn(P)
\end{aligned}$$

The set of all names of a configuration $names(\mathcal{S})$ is simply defined as $fn(\mathcal{S}) \cup bn(\mathcal{S})$

B Typing the distributed Join Calculus

A technical issue in the typing of the Join Calculus is the interaction between recursive definitions, subject reduction and type reconstruction. Let us suppose we are typing $\texttt{def}\ D\ \texttt{in}\ P$. As recursive definitions are allowed, to make type reconstruction possible, the definitions are typed in the environment extended monomorphically:

$$\Gamma + B \Vdash D :: B$$

whereas the guarded process is typed in the generalized environment A:

$$\Gamma + A \Vdash P$$

As definitions are dissolved using the chemical semantics, two previously unrelated definitions may come together:

$$\vdash \texttt{def}\ D_1\ \texttt{in}\,\texttt{def}\ D_2\ \texttt{in}\ P \rightarrow^* D_1, D_2 \vdash P$$

If D_2 uses D_1 in a polymorphic way (which is correct), the typing of the right hand process fails if D_1 and D_2 are typed as recursive processes. The solution proposed in [9] consists of rigidly keeping the structure of recursive definitions, written $D_1 \wedge D_2$, to distinguish them from definitions that came together through dissolution.

This solution cannot work in a distributed setting, as locations are definitions, and recursively defined locations may need to part:

$$\texttt{def}\ a[n\langle\rangle \triangleright m\langle\rangle : \texttt{go}\ far] \wedge b[m\langle\rangle \triangleright n\langle\rangle : \texttt{go}\ away]\ \texttt{in}\ P$$

Two approaches are possible to solve this issue. The first consists of keeping the structure of the typing (*i.e.* the order used in typing definitions, and which definitions are recursive) on the side, and applying this order when typing the process after each reduction. The problem with this solution is that the typing system is no longer directed by the syntax.

Another solution consists of proving subject reduction for a richer system, one where polymorphic recursion is available, and reconstructing the types in a system where polymorphic recursion is forbidden. As all terms that may be typed in the second system can be typed in the first, we still have type soundness (but no longer subject reduction) for the second system. This is the reason why we present a type system with polymorphic recursion.