Ownership Types for the Join Calculus

Marco Patrignani[1] *Dave Clarke*[1] *Davide Sangiorgi*[2]

Report CW 603, November 26, 2012

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Ownership Types for the Join Calculus

Marco Patrignani[1] *Dave Clarke*[1] *Davide Sangiorgi*[2]

Report CW 603, November 26, 2012

Department of Computer Science, K.U.Leuven

Abstract

This paper investigates ownership types in a concurrent setting using the Join calculus as the model of processes. Ownership types have the effect of statically preventing certain communication, and can block the accidental or malicious leakage of secrets. Intuitively, a channel defines a boundary and forbids access to its inside from outer channels, thus preserving the secrecy of the inner names from malicious outsiders. Furthermore, the secrecy is preserved even in the context of an untyped opponent.

Ownership Types for the Join Calculus

Marco Patrignani[1], Dave Clarke[1], and Davide Sangiorgi[2]

[1] IBBT-DistriNet, Dept. Computer Sciences, Katholieke Universiteit Leuven
[2] Dipartimento di Scienze dell'Informazione, Università degli studi di Bologna

1 Introduction

The Join calculus [16] is a message-based model of concurrency whose expressiveness is the same as the π-calculus up to weak barbed congruence [15]. It is a theoretical foundation of programming languages for distributed, mobile and concurrent systems, such as JOCaml [14], JErlang [22] and Scala's actor model [18]. The presence of several implementations of the calculus motivates the choice of Join calculus over π-calculus. The Join calculus has a notion of locality given by the channel definition construct: **def** `channels` **in** `scope`, but this can be broken by exporting a channel out of the environment it was created in. Ideally, scope boundaries are supposed to be crossed only by channels used for communication while channels defining secrets are supposed to remain within their scope. In practice, the malicious or accidental exportation of a channel name outside of a boundary it is not supposed to cross is a serious threat, because it can result in the leakage of secrets. Since crossing boundaries allows processes to communicate, it should not be eliminated, it has to be controlled.

Object-oriented programming suffers from a similar problem since object references can be passed around leading to issues like aliasing. As a remedy for this problem ownership types have been devised. Ownership types [11,9] statically enforce a notion of object-level encapsulation for object-oriented programming languages. Ownership types impose structural restrictions on object graphs based on the notions of *owner* and *representation*. The owner of an object is another object and its representation is the objects it owns. These two key concepts can be both defined and checked statically. The ultimate benefit imposed by ownership types is a statically-checkable notion of encapsulation: every object can have its own private collection of representation objects which are not accessible outside the object that owns them.

The aim of this paper is to adapt the strong encapsulation property imposed by ownership types, also known as *owners-as-dominators*, to a concurrent setting and use it for security enforcement. We do so by introducing an analogy between objects and channels: just as an object is owned by another object, so is a channel owned by another channel; just as an object owns other objects and limits access to them, so does a channel. As the Join calculus does not have references, the *owners-as-dominators* property is mimicked by limiting how channels are exported. The idea is that allowing a channel to be passed around as a parameter on another channel, enables other processes to refer to it. We impose access limitations by forbidding a channel to be exported outside the scope of the channel that owns it, thus allowing the kinds of restrictions analogous to those found in the object-oriented setting. A channel is however free to be passed around inside the area determined by its *owner*.

The contribution of this paper is a typed version of the Join calculus that enjoys the *owners-as-dominators* form of strong encapsulation. This property allows the calculus to define secrecy policies that are preserved both in a typed and in an untyped setting.

The paper is structured as follows. Section 2 reviews ownership and encapsulation related ideas. Section 3 introduces an ownership types system for the

Join calculus and states its secrecy properties. Section 4 shows the secrecy results in the context of an untyped opponent. Section 5 presents related work and Section 6 concludes.

2 Ownership Types

We describe the notions of object ownership and of encapsulation based on it.

In a basic ownership types system [9] every object has an owner. The owner is either another object or the predefined constant *world* for objects owned by the system. Two notions are key in this system: *owner* and *representation*. The owner defines the entity to which the object belongs to. An object implicitly defines a representation consisting of the objects it owns. The aforementioned *owner* associated to an object is indicated using the function `owner`. Owners form a tree $(C, \prec:)$, where $\prec:$ is called *inside*, that represents the nesting of objects C. The tree has root *world* and is constructed according to the following scheme: $\iota \prec: \texttt{owner}(\iota)$, where ι is a newly created object, and $\texttt{owner}(\iota)$ refers to an already existing object. The following *containment invariant* determines when an object can refer to another one: ι *refers to* $\iota' \Rightarrow \iota \prec: \texttt{owner}(\iota')$. Owners and representations are encoded in a type system and the containment invariant holds for well-typed programs.

The property the containment invariant imposes on object graphs is called *owners-as-dominators* and it states that all access paths to an object from the root of a system pass through the object's owner. In effect, ownership types erect a boundary that protects an object's internal representation from external access. The idea is having a nest of boxes where there is no access from outside to inside a box, and every object defines one such box.

3 A Typed Join Calculus with Owners: J_{OT}

In this section we extend the classical Join calculus with ownership annotations. In this concurrent setting, channels will play an analogous role to objects in the object-oriented world. A channel will have another channel as *owner* and it may have a set of channels as *representation*.

3.1 Syntax

Following the notation of Pierce [21] we write $\overline{x}$ as a shorthand for $x_1, \ldots, x_n$ (similarly $\overline{T}, \overline{t}, \overline{o}$, etc), and $\overline{f}\ \overline{g}$ for $f_1\ g_1 \ldots f_n\ g_n$. Let Σ be a denumerable set of channel names ranged over by: x, y, u, z. The name *world* is not in Σ since it is a constant. Let Ω be a denumerable set of owner variables ranged over by α, β, γ.

$P = \emptyset$	$D = \top$	$J = x\langle\overline{y}\rangle : T$	$T = {}^{o}_{o}\langle\!\langle \overline{t} \rangle\!\rangle$	$o = x$
$\mid\ x\langle\overline{u}\rangle$	$\mid\ D \wedge D$	$\mid\ J \mid J$		$\mid\ \mathit{world}$
$\mid\ P \mid P$	$\mid\ J \rhd P$		$t = \exists\alpha.T$	$\mid\ \alpha$
$\mid\ \mathbf{def}\ D\ \mathbf{in}\ P$				

Fig. 1. Syntax, types and owners definition.

Figure 1 presents the syntax of the calculus. A process P is either an empty process, a message, the parallel composition of processes or a defining process.

A definition D is either an empty definition, the conjunction of definitions or a clause $J \triangleright P$, called a *reaction pattern*, which associates a guarded process P to a specific join pattern J. A join pattern J is either a typed message or the parallel composition thereof. A type $T \equiv {}^{o}_{o'}\langle\!|\overline{t}|\!\rangle$ is used to remember the *owner* (o) and *representation* (o') of a channel and to give types ($\overline{t}$) to the parameters the channel expects to send. Typing annotations explicitly establish ownership relations. The pair owner-representation identifies a single branch of the ownership tree. Define owners o as either a channel x, the constant *world* or an owner variable α. Variables are classified as *received (rv)*, *defined (dv)* and *free (fv)*, defined by structural induction in Figure 2. A *fresh name* is a name that does not appear free in a process.

$$
\begin{array}{rlrl}
rv(x\langle\overline{y}\rangle) &= \{y_1\} \cup \cdots \cup \{y_n\} & rv(J \mid J') &= rv(J) \cup rv(J') \\
dv(x\langle\overline{y}\rangle) &= \{x\} & dv(J \mid J') &= dv(J) \cup dv(J') \\
dv(D \wedge D') &= dv(D) \cup dv(D') & fv(J \triangleright P) &= dv(J) \cup (fv(P) \setminus rv(J)) \\
fv(x\langle\overline{y}\rangle) &= \{x\} \cup \{y_1\} \cup \cdots \cup \{y_n\} & fv(\textbf{def } D \textbf{ in } P) &= (fv(P) \cup fv(D)) \setminus dv(D) \\
fv({}^{o'}_{o}\langle\!|\overline{t}|\!\rangle) &= \{o\} \cup \{o'\} \cup fv(\overline{t}) & fv(\exists\alpha.T) &= fv(T) \setminus \{\alpha\}
\end{array}
$$

Fig. 2. Definition of received, defined and free variables (obvious cases omitted).

The difference with the standard Join calculus [15] is that channels introduced in a message definition have a type annotation.

Existential quantification on owners is used to abstract over the representation of an expected parameter; thus a channel can emit names that have different *representation* while their *owner* is the same. Existential types are used implicitly, thus there are no primitives for *packing* and *unpacking*. Existential pairs would have the form $\langle x, x\rangle$, but since the *witness* and the *value* coincide, we use the binding $x : T$, which is the implicit form of $\langle x, x\rangle : \exists\alpha.T[\alpha/x]$.

Example 1 (Typing and message sending). Consider the following two processes:

$$
\begin{array}{ll}
P = \textbf{def } o\langle a\rangle \triangleright \emptyset & P_T = \textbf{def } o\langle a\rangle : {}^{world}_{o}\langle\!|\exists\alpha.\ {}^{o}_{\alpha}\langle\!|\,|\!\rangle|\!\rangle \triangleright \emptyset \\
\quad \wedge \quad x\langle\rangle \mid y\langle\rangle \triangleright R & \quad \wedge \quad x\langle\rangle : {}^{o}_{x}\langle\!|\,|\!\rangle \mid y\langle\rangle : {}^{o}_{y}\langle\!|\,|\!\rangle \triangleright R \\
\textbf{in} \quad o\langle x\rangle \mid o\langle y\rangle & \textbf{in} \quad o\langle x\rangle \mid o\langle y\rangle
\end{array}
$$

P defines three channels o, x, y and sends x and y on o. P_T is the typed version of P. R is in both cases an arbitrary process. Typing is used to enforce an ordering relation between the defined channels so that o owns x and y, thus $x, y \prec: o$. To allow both x and y to be sent on o, the type of o contains an abstraction over the parameter's representation: $\exists\alpha.\ {}^{o}_{\alpha}\langle\!|\,|\!\rangle$. Since the type of x and y are concretizations of the abstraction obtained by replacing α with x and y respectively, $o\langle x\rangle$ and $o\langle y\rangle$ are well-typed processes.

3.2 Semantics

The expression $t[a/b]$ reads "where a is substituted for all occurrences of b in t". The substitution function $[a/b]$ is defined inductively on P, the most notable case being: $(x\langle\overline{z}\rangle : T)[u/y] \equiv x\langle\overline{z}\rangle : T[u/y]$. For the sake of simplicity we assume names that are to be substituted to be distinct from the defined ones. Substitution into types is also defined inductively and the most notable case is $(\exists\alpha.T)[o/\beta] = \exists\alpha.T$ if $\alpha = \beta$.

The matching of a join pattern and a parallel composition of messages ($\mathcal{J}$) is defined by structural induction below. The procedure identifies a substitution σ that replaces all received variables of the join pattern with the ones of the message sequence, namely if $J \doteq_\sigma \mathcal{J}$ then $J\sigma \equiv \mathcal{J}$. The domains of σ_1 and σ_2 are disjoint for well-typed join patterns, so the union of σ_1 and σ_2 is well defined.

$$\frac{}{x\langle \overline{y} \rangle : T \doteq_{[\overline{u}/\overline{y}]} x\langle \overline{u} \rangle} \qquad \frac{J_1 \doteq_{\sigma_1} \mathcal{J}_1 \qquad J_2 \doteq_{\sigma_2} \mathcal{J}_2}{J_1 \mid J_2 \doteq_{\sigma_1 \cup \sigma_2} \mathcal{J}_1 \mid \mathcal{J}_2}$$

The semantics is specified as a reduced chemical abstract machine RCHAM [17]. The state of the computation is a chemical soup $\mathcal{D} \Vdash \mathcal{P}$ that consists of: $\mathcal{D}$, a set of definitions, and $\mathcal{P}$, a multiset of processes. Terms of soups will be called *molecules*. Two kind of rules describe the evolution of the soup. Structural rules, *heating* $\rightharpoonup$, and *cooling* $\leftharpoonup$, (denoted together by $\rightleftharpoons$), are reversible and are used to rearrange terms. Reduction rules (denoted by $\longrightarrow$) represent the basic computational steps. Each reduction rule consumes a process and replaces it with another. The rules for the RCHAM are given in Figure 3. Rule *S-PAR* expresses

$$\frac{}{\Vdash P_1 \mid P_2 \rightleftharpoons \Vdash P_1, P_2}\ \textit{S-PAR} \qquad \frac{dv(D) \text{ are fresh}}{\Vdash \mathbf{def}\ D\ \mathbf{in}\ P \rightleftharpoons D \Vdash P}\ \textit{S-DEF}$$

$$\frac{J \doteq_\sigma \mathcal{J}}{D \wedge J \rhd P \wedge D' \Vdash \mathcal{J} \longrightarrow D \wedge J \rhd P \wedge D' \Vdash P\sigma}\ \textit{R-BETA}$$

$$\frac{\begin{array}{c}\mathcal{D}_1 \Vdash \mathcal{P}_1 \rightleftharpoons \mathcal{D}_2 \Vdash \mathcal{P}_2 \\ (fv(\mathcal{D}) \cup fv(\mathcal{P})) \cap (dv(\mathcal{D}_1) \setminus dv(\mathcal{D}_2) \cup dv(\mathcal{D}_2) \setminus dv(\mathcal{D}_1)) = \emptyset\end{array}}{\mathcal{D}, \mathcal{D}_1 \Vdash \mathcal{P}_1, \mathcal{P} \rightleftharpoons \mathcal{D}, \mathcal{D}_2 \Vdash \mathcal{P}_2, \mathcal{P}}\ \textit{CTX}$$

Fig. 3. Chemical rules for the RCHAM.

that "|" is commutative and associative, as the soup is a multiset. Rule *S-DEF* describes the heating of a molecule that defines new names. This rule enforces that names defined in D are unique for the soup and limits the binding of such names to the process P. The side condition of this rule mimics the scope extrusion of the ν operator in π-calculus, and at the same time enforces a strict static scope for the definitions. The basic computational step is provided by rule *R-BETA*. Reduction consumes any molecule $\mathcal{J}$ that matches a given pattern J, makes a fresh copy of the guarded process P, substitutes the formal parameters in P with the corresponding actual sent names via the substitution σ, and releases this process into the soup as a new molecule. Rule *CTX* states a general evolution rule for soups. The symbol $\rightleftharpoons$ denotes any of the above reduction steps. The side condition of *CTX* ensures that the names in the additional definitions $\mathcal{D}$ and processes $\mathcal{P}$ do not clash with those already in the soup.

3.3 The Type System

The type system needs to track the ownership relations along with the types of regular variables. These are recorded in an environment parallel to the typing one. When a channel is defined, the relationship between it and its *owner* is added to such an environment.

Define the environment Γ, which provides types for free channel variables, as $\Gamma = \emptyset \mid \Gamma, (x : T)$. Similarly, environment Δ tracks constraints on owners $\Delta = \emptyset \mid \Delta, (o \prec: o')$. The function $\mathtt{dom}(\Delta)$ is define inductively as follows: $\mathtt{dom}(\emptyset) = \emptyset$, $\mathtt{dom}(\Delta, (o \prec: o')) = \mathtt{dom}(\Delta) \cup \{o\}$.

We can now introduce the concept of ownership and how the environments of the type system keep track of it.

Definition 1 (Ownership). *A channel x is said to be* owned by *a channel o w.r.t Γ and Δ if either $(x \prec: o) \in \Delta$ or $(x : \overset{o}{x}\langle\!|\overline{t}|\!\rangle) \in \Gamma$.*

Note that these two notions will coincide in the sense that if $x\langle\overline{u}\rangle$ is a well-typed message (via *P-msg*), then $(x \prec: o) \in \Delta$ iff $(x : \overset{o}{x}\langle\!|\overline{t}|\!\rangle) \in \Gamma$.

A judgment $\Gamma \vdash \overline{j}$ is abbreviation for a sequence of judgments $\Gamma \vdash j_1, \ldots, \Gamma \vdash j_n$. The type system is specified via the typing judgments of Figure 4.

$\Delta \vdash \diamond$	well-formed environment Δ
$\Delta; \Gamma \vdash \diamond$	well-formed environments Γ and Δ
$\Delta; \Gamma \vdash o$	good owner o
$\Delta; \Gamma \vdash x : T$	channel x has type T
$\Delta; \Gamma \vdash T$	good type T
$\Delta; \Gamma \vdash oRo'$	o is R-related to o', where $R \in \{\prec:, \prec:^*, \prec:^+, =\}$
$\Delta; \Gamma \vdash P$	well-typed process P
$\Delta; \Gamma \vdash D :: \Delta'; \Gamma'$	well-typed definition D. Environments Δ' and Γ' contain context relations and bindings for $dv(D)$
$\Delta; \Gamma \vdash J :: \Delta'; \Gamma'$	well-typed join pattern J. Environments Δ' and Γ' contain context relations and bindings for $dv(J)$
$\Delta; \Gamma \vdash \mathcal{D} \Vdash \mathcal{P}$	well-typed soup $\mathcal{D} \Vdash \mathcal{P}$

Fig. 4. Typing judgments for J_{OT}.

Rules for owners and for the inside relation follow Clarke and Drossopolou [10]. Syntax related rules are more or less standard for the Join calculus [17], except for rules *P-msg* and *J-def*, where existential types are implicitly used.

Environment rules are standard.

Environments

$$\frac{}{\emptyset \vdash \diamond}\ \textit{Rel-n} \qquad \frac{o \notin \mathtt{dom}(\Delta) \cup \{world\} \qquad \Delta; \emptyset \vdash o'}{\Delta, (o \prec: o') \vdash \diamond}\ \textit{Rel-c}$$

$$\frac{\Delta \vdash \diamond}{\Delta; \emptyset \vdash \diamond}\ \textit{Env-n} \qquad \frac{x \notin \mathtt{dom}(\Gamma) \qquad \Delta; \Gamma \vdash T}{\Delta; \Gamma, (x : T) \vdash \diamond}\ \textit{Env-c}$$

The following rules give the validity of owners.

Owners

$$\frac{\Delta; \Gamma \vdash \diamond \qquad o \in \mathtt{dom}(\Delta)}{\Delta; \Gamma \vdash o}\ \textit{C-rel} \qquad \frac{\Delta; \Gamma \vdash \diamond \qquad x \in \mathtt{dom}(\Gamma)}{\Delta; \Gamma \vdash x}\ \textit{C-ch}$$

The following rules capture properties of relations, based on the natural inclusions: $\prec: \subseteq \prec:^+ \subseteq \prec:^*$, $= \subseteq \prec:^*$, and $\prec::\prec: \subseteq \prec:^+$; and equivalences: $\prec::\prec:^* \equiv \prec:^*; \prec: \equiv \prec:^+$ and $=; R \equiv R; = \equiv R$ and $\prec:^*; \prec:^* \equiv \prec:^*$.

Inside relation

$$\frac{\Delta; \Gamma \vdash \diamond \qquad oRo' \in \Delta}{\Delta; \Gamma \vdash oRo'}\ \textit{In-rel} \qquad \frac{\Delta; \Gamma \vdash o}{\Delta; \Gamma \vdash o = o}\ \textit{In-ref} \qquad \frac{\Delta; \Gamma \vdash o}{\Delta; \Gamma \vdash o \prec: world}\ \textit{In-wo}$$

$$\frac{\Delta;\Gamma \vdash oRo' \qquad R \subseteq R'}{\Delta;\Gamma \vdash oR'o'}\ \textit{In-weak} \qquad \frac{\Delta;\Gamma \vdash oRo' \qquad \Delta;\Gamma \vdash o'R'o''}{\Delta;\Gamma \vdash oR;R'o''}\ \textit{In-trans}$$

A type is valid (*Type-ch*) if the *representation* is *directly inside* the *owner*, and *inside* all the owners of the types that are sent. To access the *owner* of a type, use the function $\mathtt{own}(\ )$, defined as: $\mathtt{own}({}^{o'}_{o}\langle\!|\bar{t}|\!\rangle) = o'$, $\mathtt{own}(\exists\alpha.T) = \mathtt{own}(T)$.

Types

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash o \prec: o' \\ \Delta;\Gamma \vdash o \prec:^* \mathtt{own}(\bar{t}) \qquad \Delta;\Gamma \vdash \bar{t}\end{array}}{\Delta;\Gamma \vdash {}^{o'}_{o}\langle\!|\bar{t}|\!\rangle}\ \textit{Type-ch} \qquad \frac{\Delta,(\alpha \prec: o);\Gamma \vdash {}^{o}_{\alpha}\langle\!|\bar{t}|\!\rangle}{\Delta;\Gamma \vdash \exists\alpha.\ {}^{o}_{\alpha}\langle\!|\bar{t}|\!\rangle}\ \textit{Type-}\exists$$

Rule *P-msg* enforces that the channel name and the *representation* indicated in its type must coincide. The substitution $T_i[u_i/\alpha_i]$ in *P-msg* is an implicit unpacking of the witness u_i contained in the implicit existential pair $\langle u_i, u_i\rangle$ created in the eventual definition of u_i.

Processes

$$\frac{\Delta;\Gamma \vdash \diamond \qquad (x:T) \in \Gamma}{\Delta;\Gamma \vdash x:T}\ \textit{Chan}$$

$$\frac{\Delta;\Gamma \vdash \diamond}{\Delta;\Gamma \vdash \emptyset}\ \textit{P-null} \qquad \frac{\Delta;\Gamma \vdash P \qquad \Delta;\Gamma \vdash P'}{\Delta;\Gamma \vdash P \mid P'}\ \textit{P-par}$$

$$\frac{\begin{array}{c}\Delta,\Delta';\Gamma,\Gamma' \vdash P \\ \Delta,\Delta';\Gamma,\Gamma' \vdash D :: \Delta';\Gamma' \qquad \mathtt{dom}(\Gamma') = dv(D) = \mathtt{dom}(\Delta')\end{array}}{\Delta;\Gamma \vdash \mathbf{def}\ D\ \mathbf{in}\ P}\ \textit{P-def}$$

$$\frac{\Delta;\Gamma \vdash x:\ {}^{o}_{x}\langle\!|\exists\overline{\alpha}.\overline{T}|\!\rangle \qquad \Delta;\Gamma \vdash u_i : T_i[u_i/\alpha_i] \text{ for each } i \in 1..n}{\Delta;\Gamma \vdash x\langle\overline{u}\rangle}\ \textit{P-msg}$$

Rule *D-run* shows that bindings for defined channels $(\Delta_d;\Gamma_d)$ are collected and available while their scope lasts, while bindings for received channels (Δ_r,Γ_r) are collected only where they are used, namely in the started process P.

Definitions

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash D :: \Delta';\Gamma' \\ \Delta;\Gamma \vdash D' :: \Delta'';\Gamma'' \qquad dv(D) \cap dv(D') = \emptyset\end{array}}{\Delta;\Gamma \vdash D \wedge D' :: \Delta',\Delta'';\Gamma',\Gamma''}\ \textit{D-and} \qquad \frac{\Delta;\Gamma \vdash \diamond}{\Delta;\Gamma \vdash \top}\ \textit{D-top}$$

$$\frac{\begin{array}{rl} & \Delta_r = \Delta' \setminus \Delta_d \quad \Gamma_r = \Gamma' \setminus \Gamma_d \\ \Delta,\Delta_r;\Gamma,\Gamma_r \vdash J :: \Delta';\Gamma' & \Delta_d = \{(x \prec: o) \in \Delta' \mid x \in dv(J)\} \\ \Delta,\Delta_r;\Gamma,\Gamma_r \vdash P & \Gamma_d = \{(x:T) \in \Gamma' \mid x \in dv(J)\}\end{array}}{\Delta;\Gamma \vdash J \rhd P :: \Delta_d;\Gamma_d}\ \textit{D-run}$$

An implicit packing of $\langle x,x\rangle$ and $\langle\overline{y},\overline{y}\rangle$ is made in rule *J-def* dual to the unpacking that occurs in rule *P-msg*. Rule *J-def* also provides bindings for both the defined channel and its formal parameters.

Join patterns

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash J :: \Delta';\Gamma' \qquad dv(J) \cap dv(J') = \emptyset \\ \Delta;\Gamma \vdash J' :: \Delta'';\Gamma'' \qquad rv(J) \cap rv(J') = \emptyset\end{array}}{\Delta;\Gamma \vdash J \mid J' :: \Delta',\Delta'';\Gamma',\Gamma''}\ \textit{J-par}$$

$$\frac{\begin{array}{cc}\Delta;\Gamma \vdash x : T & \Delta;\Gamma \vdash y_i : T_i[y_i/\alpha_i] \text{ for each } i \in 1..n \\ T \equiv {}^{o}_{x}\langle\!|\exists \overline{\alpha}.\overline{T}|\!\rangle & T_i \equiv {}^{o_i}_{\alpha_i}\langle\!|\overline{t_i}|\!\rangle\end{array}}{\Delta;\Gamma \vdash x\langle\overline{y}\rangle : T :: (x \prec: o), (\overline{y} \prec: \overline{o}); (x : T), (\overline{y} : \overline{T}[\overline{y}/\overline{\alpha}])}\ \textit{J-def}$$

Soups

$$\frac{\mathcal{P} = P_1, \ldots, P_n \qquad \Delta;\Gamma \vdash P_i \text{ for each } i = 1..n}{\Delta;\Gamma \vdash \mathcal{P}}\ \textit{P-elim}$$

$$\frac{\begin{array}{cc}\Delta;\Gamma \vdash D_i :: \Delta'_i;\Gamma'_i \text{ for each } i = 1..n & \Gamma' = \Gamma'_1, \ldots, \Gamma'_n \\ dv(D_1) \cap \ldots \cap dv(D_n) = \emptyset & \Delta' = \Delta'_1, \ldots, \Delta'_n\end{array}}{\Delta;\Gamma \vdash \mathcal{D} :: \Delta';\Gamma'}\ \textit{D-elim}$$

$$\frac{\begin{array}{cc}\Delta, \Delta';\Gamma, \Gamma' \vdash \mathcal{P} & \\ \Delta, \Delta';\Gamma, \Gamma' \vdash \mathcal{D} :: \Delta';\Gamma' & \texttt{dom}(\Gamma') = dv(\mathcal{D}) = \texttt{dom}(\Delta')\end{array}}{\Delta;\Gamma \vdash \mathcal{D} \Vdash \mathcal{P}}\ \textit{Soup}$$

3.4 Soundness of the Type System

A type system for a concurrent language is correct whenever two standard theorems hold [24]. The first, subject reduction, ensures that typing is preserved by reduction. This means no typing error arise as the computation proceeds. The second one, no runtime errors, ensures that no error occurs as the computation progresses. An error may be sending a message with a different number of parameters than expected or, as is specific to our type system, breaking the *owners-as-dominators* property. Proofs can be found in a companion technical report [20].

Definition 2 (Typing environment agreement $\frown$). *Two typing environments* agree*, denoted with $\Delta;\Gamma \frown \Delta';\Gamma'$, if the variables they have in common have the same type.*

Theorem 1 (Subject reduction for J_{OT}). *One step chemical reductions preserve typings. If $\Delta;\Gamma \vdash \mathcal{D} \Vdash \mathcal{P}$ and $\mathcal{D} \Vdash \mathcal{P} \rightleftharpoons \mathcal{D}' \Vdash \mathcal{P}'$, then there exists $\Delta';\Gamma'$ such that $\Delta;\Gamma \frown \Delta';\Gamma'$ and $\Delta';\Gamma' \vdash \mathcal{D}' \Vdash \mathcal{P}'$.*

Definition 3 (Runtime errors). *Consider a soup $\mathcal{D} \Vdash \mathcal{P}$. Say that a runtime error occurs if any of these kind of messages occurs in the processes set $\mathcal{P}$:*

- *a message $x\langle\overline{y}\rangle$ that is not defined in $\mathcal{D}$, i.e. no join pattern J in $\mathcal{D}$ has x in its defined variables;*
- *a message $x\langle\overline{y}\rangle$ that is defined in $\mathcal{D}$ but with different arity (e.g. the defined channel x wants four parameters while we call it with three);*
- *a message $x\langle\overline{y}\rangle$ where x is not* inside *some of its arguments' owners, i.e. there is a y_i owned by o such that $x \not\prec:^* o$*

Reduction of a well-typed soup never results in a runtime error.

Theorem 2 (No runtime errors for J_{OT}). *If $\Delta;\Gamma \vdash \mathcal{D} \Vdash \mathcal{P}$ and $\mathcal{D} \Vdash \mathcal{P} \rightleftharpoons^* \mathcal{D}' \Vdash \mathcal{P}'$, then no runtime error occur in $\mathcal{D}' \Vdash \mathcal{P}'$.*

The following statement is a translation of the *owners-as-dominators* property ownership types enforce in the object-oriented setting. No reference from outside the ownership boundaries to an inner channel exists.

Theorem 3 (Owners as dominators). *A channel y owned by o may be sent over a channel x only if x is* transitively inside *o: $\Delta;\Gamma \vdash x\langle y\rangle \Rightarrow \Delta;\Gamma \vdash x \prec:^* o$*

Corollary 1 is a restatement of Theorem 3 from a secrecy perspective. The point of view of Theorem 3 highlights what can be sent on a channel. Dually, the perspective of Corollary 1 points out what cannot be sent on a channel.

Corollary 1 (Secrecy for J_{OT}). *Consider a well-typed soup $\mathcal{D} \Vdash \mathcal{P}$ that defines a channel y owned by o. However the soup evolves, y is not accessible from channels whose owner is not* transitively inside *o.*

Example 2 (Secrecy with a typed opponent). Consider the typed process P of Example 1. Suppose P is a private subsystem of a larger system O. Process P defines two secrets, namely x and y, which are intended to remain private to P. This privacy policy can be violated if, for example, a subsystem R can, after a sequence of reduction steps, send $l\langle x \rangle$, where l is a channel known to O. To typecheck the definition of l, O should know the name o. Fortunately o is not in the environment O uses to typecheck since o is defined in P. As the following proof tree shows, l cannot be typed in order to send channels owned by o. This means that the secrecy of x is preserved.

$$\dfrac{\dfrac{\dots \qquad \dfrac{\text{Impossible since } o \text{ is unknown}}{\Delta; \Gamma \vdash l \prec:^* o} \qquad \dots}{\Delta; \Gamma \vdash l : \ {}^{world}_{l}\langle\!|\exists \alpha.\ {}^{o}_{\alpha}\langle\!|\,|\!\rangle\!\rangle \quad \dots}\ \textit{Type-ch}}{\Delta; \Gamma \vdash l\langle a \rangle : \ {}^{world}_{l}\langle\!|\exists \alpha.\ {}^{o}_{\alpha}\langle\!|\,|\!\rangle\!\rangle :: \ \dots}\ \textit{Chan}$$

Example 3 (Code invariant definition). Consider the following process.

$$\begin{aligned} B = \mathbf{def}\ & mb\langle pb \rangle : T_m \triangleright \mathbf{def}\ guard\langle\rangle : \ {}^{world}_{guard}\langle\!|\,|\!\rangle \triangleright \emptyset \\ & \wedge \quad empty\langle\rangle : \ {}^{guard}_{empty}\langle\!|\,|\!\rangle \mid put\langle d \rangle : T_p \triangleright full\langle d \rangle \\ & \wedge \quad full\langle c \rangle : \ {}^{guard}_{full}\langle\!|T_c|\!\rangle \mid get\langle r \rangle : T_g \triangleright empty\langle\rangle \mid r\langle c \rangle \\ & \mathbf{in} \quad empty\langle\rangle \mid pb\langle put, get \rangle \\ \mathbf{in} \quad & \dots \end{aligned}$$

Process B is a one-place buffer where typing enforces the invariant: the internal representation of the buffer cannot be accessed except via *put* and *get*. The buffer is spawned by sending a channel *pb* over *mb*. Primitives for buffer usage, *put* and *get*, will be returned to the buffer creator on channel *pb*. Since *empty* and *full* are owned by channel *guard*, there is no possibility for them to be used outside the process boundaries, for the same reasons as Example 2. Channel *guard* and type annotations are the only additions to an untyped one-place buffer [16] which are needed to enforce the invariant. We do not fully specify all types involved in B, focussing the attention only on the types needed to define the invariant.

4 Secrecy in the Context of an Untyped Opponent

When computation is run on a collection of machines that we cannot have access to or simply rely on, we expect our programs to interact with code that possibly does not conform to our type system, which could result in a malicious attempt to access the secrets of our programs. The system must be strong enough to prevent such attacks.

We follow the approach of Cardelli et. al. [8] and formalize the idea that a typed process in J_{OT} can keep a secret from any opponent it interacts with, be it well- or ill-typed. The story reads as follows:

- consider a well-typed J_{OT} soup $\mathcal{D} \Vdash \mathcal{P}$ that defines a secret x owned by o;
- consider an untyped soup $\mathcal{D}' \Vdash \mathcal{P}'$ that knows neither o nor x a priori;
- erase typing annotation from $\mathcal{D} \Vdash \mathcal{P}$ and combine its molecules with those of $\mathcal{D}' \Vdash \mathcal{P}'$, the result is an untyped soup $\mathcal{D}'' \Vdash \mathcal{P}''$;

– then, in any way the untyped soup $\mathcal{D}'' \Vdash \mathcal{P}''$ can evolve, it does not leak the secret x.

We work with soups instead of processes as a process P is trivially a soup $\emptyset \Vdash P$.

The notation J_{UN} will be used to refer to the untyped Join calculus. The syntax for J_{UN} [15,16] is analogous to the one presented in Figure 1, except that typing annotations are dropped from channel definitions. The semantics of J_{UN} follows that of Figure 3 ignoring types.

In the next section a common framework where trusted and untrusted channels coexist is introduced. We give a precise definition of when an untyped process preserves the secrecy of a channel x from an opponent. The most important result presented in the paper is that the untyped process obtained by erasing type annotations from a well-typed J_{OT} process preserves the secrecy of x from *any* opponent it interacts with.

4.1 An Auxiliary Type System: J_{AU}

Proving the secrecy theorem in an untyped opponent context is based on an auxiliary type system. The auxiliary type system partitions the set of channels into untrusted and trusted ones with regards to one specific channel: the secret. The idea is that untrusted channels do not have access to the secret, trusted channels on the other side can handle such secret. Typing enforces such a distinction. Types have syntax: $T = Un \mid In\langle\!|\overline{T}|\!\rangle$. Untrusted channels have type Un. Trusted channels have type $In\langle\!|\overline{T}|\!\rangle$, where each T_i is either trusted or untrusted. The property enforced by the type system is that trusted channels cannot be sent over untrusted ones. Hence an opponent knowing only untrusted (Un) names cannot receive a trusted (In) one.

Define Γ as a list of bindings of variables to types: $\Gamma = \emptyset \mid \Gamma, (x : T)$. The typing judgments that define the type system are analogous to those in Figure 4, except that owners and environment Δ are dropped. Typing rules for message sending (*P-msg*) and channel definition (*J-def*) need to be replaced by rules *P-mun* and *P-min* and by rules *J-dun* and *J-din* below, respectively. Judgments for the J_{AU} type system are made against untyped processes, as defined channels in rules *J-dun* and *J-din* do not have typing annotation.

$$\frac{\Gamma \vdash x : Un \qquad \Gamma \vdash \overline{y} : Un}{\Gamma \vdash x\langle\overline{y}\rangle :: (x : Un), (\overline{y} : Un)}\ \textit{J-dun} \qquad \frac{\Gamma \vdash x : In\langle\!|\overline{T}|\!\rangle \qquad \Gamma \vdash \overline{y} : \overline{T}}{\Gamma \vdash x\langle\overline{y}\rangle :: (x : In\langle\!|\overline{T}|\!\rangle), (\overline{y} : \overline{T})}\ \textit{J-din}$$

Rules *P-mun* and *P-min* state that untrusted channels cannot send anything but untrusted ones while trusted channels can mention both kinds.

$$\frac{\Gamma \vdash x : Un \qquad \Gamma \vdash \overline{y} : Un}{\Gamma \vdash x\langle\overline{y}\rangle}\ \textit{P-mun} \qquad \frac{\Gamma \vdash x : In\langle\!|\overline{T}|\!\rangle \qquad \Gamma \vdash \overline{y} : \overline{T}}{\Gamma \vdash x\langle\overline{y}\rangle}\ \textit{P-min}$$

The auxiliary type system enjoys subject reduction, as Theorem 4 shows.

Theorem 4 (Subject reduction). *If $\Gamma \vdash \mathcal{D} \Vdash \mathcal{P}$ and $\mathcal{D} \Vdash \mathcal{P} \rightleftharpoons \mathcal{D}' \Vdash \mathcal{P}'$ then there exists a Γ' such that $\Gamma' \vdash \mathcal{D}' \Vdash \mathcal{P}'$ and $\Gamma \frown \Gamma'$.*

4.2 The Common Framework

The auxiliary type system is used as a meeting point for untyped and typed soups. It serves as a common framework for testing secrecy. Assigning trusted and untrusted types to channels makes J_{AU} a suitable system to reason about coexisting trusted and untrusted processes. From now on the notation $\vdash_{OT}$ will indicate a judgment in the J_{OT} system, while $\vdash_{AU}$ will indicate a judgment in the J_{AU} one. Firstly we must be able to typecheck any untyped opponent. The way of doing it is provided by Proposition 1.

Proposition 1. *For all untyped soups $\mathcal{D} \Vdash \mathcal{P}$, if $fv(\mathcal{D} \Vdash \mathcal{P}) = \{x_1, \ldots, x_n\}$, then $x_1 : Un, \ldots, x_n : Un \vdash_{AU} \mathcal{D} \Vdash \mathcal{P}$.*

Secondly we need a way to add elements from J_{OT} to the common framework. This is done by erasing all type annotations of such elements and by mapping typing environments of J_{OT} to J_{AU} ones. Type annotations can be erased via an erasure function ($\texttt{erase}(\ )$), which is defined by structural induction on P, the most notable case being $\texttt{erase}(x\langle\overline{y}\rangle{:}T) = x\langle\overline{y}\rangle$. Translating the typechecking environment associated with a J_{OT} process to an J_{AU} environment is done via the mapping function $[\![\ ]\!]_o$ presented below.

$$[\![\Delta; \Gamma]\!]_o = [\![\Gamma]\!]_o \qquad [\![\emptyset]\!]_o = \emptyset \qquad [\![\Gamma, (x : T)]\!]_o = [\![\Gamma]\!]_o, (x : [\![T]\!]_o)$$

The environment Δ is dropped since it is not required. Γ is translated in an environment that contains bindings for the auxiliary type system. For any channel o that identifies a secret, map J_{OT} types to J_{AU} ones as follows.

$$[\![\overset{o'}{o}\langle\!|\overline{t}|\!\rangle]\!]_o = \begin{cases} In\langle\!|[\![\overline{t}]\!]_o|\!\rangle & \text{if } o \in fv(\ \overset{o'}{o}\langle\!|\overline{t}|\!\rangle) \\ Un & \text{else} \end{cases} \qquad [\![\exists\alpha.T]\!]_o = [\![T]\!]_o$$

Types that do not mention the secret o are translated as untrusted: Un. All others preserve their structure and are translated as trusted: $In\langle\!|\overline{T}|\!\rangle$. Note that trusted channels are the ones *transitively inside* the given channel o, while untrusted ones are those that are outside the ownership boundaries o defines.

Proposition 2 shows how to combine the erasure and the mapping functions to obtain a well-typed soup in J_{AU} starting from a well-typed one in J_{OT}. The secret o is any channel, be it a free name in the environment, a bound name in the soup or a fresh name.

Proposition 2. *If $\Delta; \Gamma \vdash_{OT} \mathcal{D} \Vdash \mathcal{P}$ then, for any o, $[\![\Delta; \Gamma]\!]_o \vdash_{AU} \texttt{erase}(\mathcal{D} \Vdash \mathcal{P})$.*

Now we have all the machinery to bring together trusted and untrusted processes in a common framework and show that the former ones cannot leak secrets to the latter ones.

4.3 Secrecy Theorem

Before stating the secrecy results in the untyped setting we need to introduce some related concepts.

Soup combination allows us to merge the molecules of two soups if the typechecking environments agree. The agreement implies that names that are common to the two soups have the same degree of trust, there is no name that is considered trusted in a soup but is untrusted in the other one.

Definition 4 (Soups combination). *Consider two untyped soups $\mathcal{D} \Vdash \mathcal{P}$ and $\mathcal{D}' \Vdash \mathcal{P}'$. Suppose they are well-typed in J_{AU}, so there exist Γ, Γ' such that $\Gamma \vdash_{AU} \mathcal{D} \Vdash \mathcal{P}$ and $\Gamma \vdash_{AU} \mathcal{D}' \Vdash \mathcal{P}'$. If $\Gamma \frown \Gamma'$, the molecules of the two soups can be combined into a single one: $\mathcal{D}, \mathcal{D}' \Vdash \mathcal{P}, \mathcal{P}'$.*

The definition of secret leakage we use is inspired by Abadi [1] for the untyped spi calculus [3]. The underlying idea is attributed to Dolev and Yao [13]: a name is kept secret from an opponent if after no series of interactions is the name transmitted to the opponent.

Definition 5 (Secret leakage). *A soup $\mathcal{D} \Vdash \mathcal{P}$ leaks secrets whenever $\mathcal{D} \Vdash \mathcal{P} \rightleftharpoons^* \mathcal{D}' \Vdash \mathcal{P}'$ and in $\mathcal{P}'$ there is an emission of a trusted channel on an untrusted one.*

The following proposition is the crux of the proof of Theorem 5: an opponent who knows only untrusted names cannot learn any trusted one.

Proposition 3. *Suppose* $y_1 : Un, \ldots, y_n : Un, o : T', x : T \vdash_{AU} \mathcal{D} \Vdash \mathcal{P}$*, where* $T, T' \neq Un$*. Then, the untyped soup* $\mathcal{D} \Vdash \mathcal{P}$ *does not leak secrets.*

Theorem 5 shows that an opponent, which does not know any trusted channel a priori, does not learn any trusted name by interacting with a well-typed J_{OT} soup whose type annotations have been erased. Definition 4 allows the two soups to have names in common and to communicate as long as the shared channels have the same degree of trust.

Theorem 5 (Secrecy in unsafe context). *Consider a well-typed* J_{OT} *soup* $\mathcal{D} \Vdash \mathcal{P}$ *and an untyped soup* $\mathcal{D}' \Vdash \mathcal{P}'$ *that does not know a priori any trusted name of the typed one. Let* $\mathcal{D}'' \Vdash \mathcal{P}''$ *be the combination of* $\texttt{erase}(\mathcal{D} \Vdash \mathcal{P})$ *and* $\mathcal{D}' \Vdash \mathcal{P}'$*. Then* $\mathcal{D}'' \Vdash \mathcal{P}''$ *does not leak secrets.*

5 Related Work

Ownership types have had several applications, mostly unrelated to security. They have been combined with effects for reasoning about programs [10]. Ownership types were used to detect data races and deadlocks [5], to allow safe persistent storage of objects [6] and to allow safe region-based memory management in real time computation [7]. Preventing uncontrolled accesses to EJBs [12] was obtained using a notion of containment similar to ownership types. Another similar but more lightweight idea is that of confined types [23], which provide a per-package notion of object encapsulation. To our knowledge ownership types have never been used before as a direct way to enforce secrecy policies. Furthermore, this is the first attempt to translate them from the object-oriented setting to a process calculus. A translation to the π-calculus appears to be straightforward. Ownership types have also been encoded into System F [19].

As stated before, types are not present in standard Join calculus [15]. Absence of type annotations is a difference also with the typed Join calculus [17], where typing is implicit and polymorphic. Since it loosens the constraints imposed by typing, polymorphism is not used in the current work.

In process algebra, security has been achieved via encryption both for the Join calculus [2] and the π-calculus [3]. The cited works require encryption and decryption primitives while the presented work does not. The control flow analysis for the π-calculus [4] testifies that a process provides a certain encapsulation property, on the other side our type system allows the programmer to specify such a property.

Cardelli et al.'s Groups [8] is the closest work to the results presented here. Groups were created for the π-calculus, a translation in the Join calculus seems however straightforward. Groups are created with a specific operator: νG that mimics the scope extrusion principle of channel definition νx. Channels have type $T = G[\overline{T}]$ which indicates the group G a channel belongs to, and its parameters' types $\overline{T}$. $\overline{T}$ can only mention groups which are in scope when T is defined. The notion of secrecy of Groups is thus based on scoping and there are policies that cannot be expressed using Groups, as Example 4 shows.

Example 4 (Expressiveness). Consider a process that is supposed to match the password provided from a client, sent on channel c, and the corresponding database entry, sent on channel d. If the client is able to access any entry, it can impersonate any user. Database entries must be protected from the client but they must be accessible from the database. P_G and P_O specify such a policy using Groups or using ownership types respectively.

$$P_G = \nu G \quad \textbf{def } d\langle \overline{a} \rangle : T_d \mid c\langle \overline{b} \rangle : T_c \triangleright R \quad \textbf{in } \ldots \qquad P_O = \textbf{def } d\langle \overline{a} \rangle : \ {}^{world}_{d}\langle\!|\rangle \mid c\langle \overline{b} \rangle : \ {}^{world}_{c}\langle\!|\rangle \triangleright R \quad \textbf{in } \ldots$$

To design the policy using Groups we can create a group G that is supposed to protect the database entries. In order for the database to access the entries via channel d, the declaration of G should appear before the join pattern that starts the password checking process, since such join pattern defines d. This would allow the type of the client, T_c, to mention G. The client could then access the data hidden within G via the channel c, violating the intended security policy. The designed policy can be expressed using ownership types as P_O shows. Channel c would not be able to access the entries owned by d because $c \not\prec :^* d$. On the other side d would still be able to access the database entries since d owns them.

6 Conclusion

This paper shows how to prevent the leakage of secrets in a mobile setting using ownership types. We provide a type system that encodes ownership types concepts in the Join calculus. The type system enforces the *owners-as-dominators* property and the consequent strong form of encapsulation. A typed process protects its secrets against malicious or accidental leakage. Secrecy is also preserved even in the context of an untyped opponent.

A formal comparison between ownership types and Cardelli et al.'s Groups remains to be done. Additionally, in order to allow a JOCaml implementation of ownership, polymorphism and type inference need to be investigated.

References

1. Martín Abadi. Security protocols and specifications. In *FoSSaCS '99*, volume 1578 of *LNCS*, pages 1–13, London, UK, 1999. Springer-Verlag.
2. Martín Abadi, Cédric Fournet, and Georges Gonthier. Secure implementation of channel abstractions. *Inf. Comput.*, 174:37–83, April 2002.
3. Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Inf. Comput.*, 148(1):1–70, 1999.
4. Chiara Bodei, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. Control flow analysis for the pi-calculus. In *CONCUR '98*, volume 1466 of *LNCS*, pages 84–98, London, UK, 1998. Springer-Verlag.
5. Chandrasekhar Boyapati, Robert Lee, and Martin C. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA*, pages 211–230, 2002.
6. Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In *POPL*, pages 213–223, 2003.
7. Chandrasekhar Boyapati, Alexandru Salcianu, William S. Beebee, and Martin C. Rinard. Ownership types for safe region-based memory management in real-time java. In *PLDI*, pages 324–337, 2003.
8. Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Secrecy and group creation. *Inf. Comput.*, 196(2):127–155, 2005.
9. Dave Clarke. *Object Ownership and Containment*. PhD thesis, University of New South Wales, July 2001.
10. Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA*, pages 292–310, 2002.
11. Dave Clarke, John Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA*, pages 48–64, 1998.
12. Dave Clarke, Michael Richmond, and James Noble. Saving the world from bad beans: deployment-time confinement checking. In *OOPSLA '03*, pages 374–387, New York, NY, USA, 2003. ACM.

13. D. Dolev and A. C. Yao. On the security of public key protocols. In *SFCS*, pages 350–357, Washington, DC, USA, 1981. IEEE Computer Society.
14. Cédric Fournet, Fabrice Le Fessant, Luc Maranget, and Alan Schmitt. *JoCaml: a Language for Concurrent Distributed and Mobile Programming*, pages 129–158. LNCS. Springer-Verlag, November 2002.
15. Cédric Fournet and Georges Gonthier. The reflexive CHAM and the Join-calculus. In *POPL*, pages 372–385, 1996.
16. Cédric Fournet and Georges Gonthier. The Join calculus: A language for distributed mobile programming. In *APPSEM*, volume 2395 of *LNCS*, pages 268–332, 2000.
17. Cédric Fournet, Cosimo Laneve, Luc Maranget, and Didier Rémy. Implicit typing à la ML for the Join-calculus. In *CONCUR*, pages 196–212, 1997.
18. Philipp Haller and Tom Van Cutsem. Implementing Joins using Extensible Pattern Matching. In *COORDINATION'08*, volume 5052 of *LNCS*, pages 135–152. Springer, 2008.
19. Neelakantan R. Krishnaswami and Jonathan Aldrich. Permission-based ownership: encapsulating state in higher-order typed languages. In *PLDI*, pages 96–106, 2005.
20. Marco Patrignani, Dave Clarke, and Davide Sangiorgi. Ownership types for the Join calculus. CW Reports CW603, Dept. of Computer Science, K.U.Leuven, March 2011.
21. Benjamin Pierce. *Types and Programming Languages*. MIT Press, 2002.
22. Hubert Plociniczak and Susan Eisenbach. Jerlang: Erlang with Joins. In *COORDINATION*, volume 6116 of *LNCS*, pages 61–75, 2010.
23. Jan Vitek and Boris Bokowski. Confined types. In *OOPSLA '99*, pages 82–96, New York, NY, USA, 1999. ACM.
24. Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115:38–94, November 1994.

A Extended definition of functions

Extended version of the substitution function $t[a/b]$ defined in Section 3.2.

Substitution for processes

$$(\mathbf{def}\ D\ \mathbf{in}\ P)[u/y] \equiv \mathbf{def}\ D[u/y]\ \mathbf{in}\ P[u/y] \qquad (P \mid P')[u/y] \equiv P[u/y] \mid P'[u/y]$$
$$x\langle\overline{z}\rangle[u/y] \equiv x[u/y]\langle\overline{z}[u/y]\rangle \qquad \emptyset[u/y] \equiv \emptyset$$
$$x[u/y] \equiv x \ \text{ if } x \neq y \qquad y[u/y] = u$$
$$(D \wedge D')[u/y] \equiv D[u/y] \wedge D'[u/y] \qquad (J \rhd P)[u/y] \equiv J[u/y] \rhd P[u/y]$$
$$(J \mid J')[u/y] \equiv J[u/y] \mid J'[u/y] \qquad (x\langle\overline{z}\rangle : T)[u/y] \equiv x\langle\overline{z}\rangle : T[u/y]$$

Substitution for types and owners

$$(\exists\alpha.T)[o/\beta] = \exists\alpha.T \ \text{ if } \alpha = \beta \qquad o[o'/\alpha] = o' \ \text{ if } o = \alpha$$
$$(\exists\alpha.T)[o/\beta] = \exists\alpha.T[o/\beta] \ \text{ if } \alpha \neq \beta \qquad o[o'/\alpha] = o \ \text{ if } o \neq \alpha$$
$$\left(\begin{smallmatrix} o' \\ o \end{smallmatrix} \langle\!|\overline{t}|\!\rangle\right)[o/\alpha] = \begin{smallmatrix} o'[o/\alpha] \\ o[o/\alpha] \end{smallmatrix} \langle\!|\overline{t}[o/\alpha]|\!\rangle$$

Extended version of the erasure function defined in Section 4.2.

$$\mathtt{erase}(P \mid P') = \mathtt{erase}(P) \mid \mathtt{erase}(P') \qquad \mathtt{erase}(\emptyset) = \emptyset$$
$$\mathtt{erase}(\mathbf{def}\ D\ \mathbf{in}\ P) = \mathbf{def}\ \mathtt{erase}(D)\ \mathbf{in}\ \mathtt{erase}(P) \qquad \mathtt{erase}(x\langle\overline{u}\rangle) = x\langle\overline{u}\rangle$$
$$\mathtt{erase}(D \wedge D') = \mathtt{erase}(D) \wedge \mathtt{erase}(D') \qquad \mathtt{erase}(\top) = \top$$
$$\mathtt{erase}(J \rhd P) = \mathtt{erase}(J) \rhd \mathtt{erase}(P)$$
$$\mathtt{erase}(J \mid J') = \mathtt{erase}(J) \mid \mathtt{erase}(J') \qquad \mathtt{erase}(x\langle\overline{y}\rangle : T) = x\langle\overline{y}\rangle$$
$$\mathtt{erase}(\mathcal{P}) = \mathtt{erase}(P_1), \ldots, \mathtt{erase}(P_n)$$
$$\mathtt{erase}(\mathcal{D}) = \mathtt{erase}(D_1), \ldots, \mathtt{erase}(D_n)$$
$$\mathtt{erase}(\mathcal{D} \Vdash \mathcal{P}) = \mathtt{erase}(\mathcal{D}) \Vdash \mathtt{erase}(\mathcal{P})$$

Typing rules for J_{AU}.

Environments

$$\frac{}{\emptyset \vdash \diamond}\ \textit{Env-null} \qquad \frac{\Gamma \vdash \diamond \qquad x \notin \mathtt{dom}(\Gamma)}{\Gamma, (x : T) \vdash \diamond}\ \textit{Env-c}$$

Processes

$$\frac{\Gamma \vdash \diamond \qquad (x : T) \in \Gamma}{\Gamma \vdash x : T}\ \textit{Ch} \qquad \frac{\Gamma \vdash \diamond}{\Gamma \vdash \emptyset}\ \textit{P-null} \qquad \frac{\Gamma \vdash P \qquad \Gamma \vdash P'}{\Gamma \vdash P \mid P'}\ \textit{P-par}$$

$$\frac{\Gamma, \Gamma' \vdash P \qquad \Gamma, \Gamma' \vdash D :: \Gamma' \qquad \mathtt{dom}(\Gamma') = dv(D)}{\Gamma \vdash \mathbf{def}\ D\ \mathbf{in}\ P}\ \textit{P-def}$$

$$\frac{\Gamma \vdash x : Un \qquad \Gamma \vdash \overline{y} : Un}{\Gamma \vdash x\langle\overline{y}\rangle}\ \textit{P-mun} \qquad \frac{\Gamma \vdash x : \ In\langle\!|\overline{T}|\!\rangle \qquad \Gamma \vdash \overline{y} : \overline{T}}{\Gamma \vdash x\langle\overline{y}\rangle}\ \textit{P-min}$$

Definitions

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \top}\ \textit{D-top} \qquad \frac{\begin{array}{c}\Gamma \vdash D :: \Gamma' \\ \Gamma \vdash D' :: \Gamma'' \qquad \mathtt{dom}(\Gamma') \cap \mathtt{dom}(\Gamma'') = \emptyset\end{array}}{\Gamma \vdash D \wedge D' :: \Gamma', \Gamma''}\ \textit{D-and}$$

$$\frac{\Gamma, \Gamma_r \vdash J :: \Gamma' \qquad \Gamma_r = \Gamma' \setminus \Gamma_d \qquad \Gamma, \Gamma_r \vdash P \qquad \Gamma_d = \{(x : T) \in \Gamma' \mid x \in dv(J)\}}{\Gamma \vdash J \rhd P :: \Gamma_d}\ \textit{D-run}$$

Join patterns

$$\frac{\Gamma \vdash J :: \Gamma' \qquad dv(J) \cap dv(J') = \emptyset \qquad \Gamma \vdash J :: \Gamma'' \qquad rv(J) \cap rv(J') = \emptyset}{\Gamma \vdash J \mid J' :: \Gamma', \Gamma''}\ \textit{J-par}$$

$$\frac{\Gamma \vdash x : Un \qquad \Gamma \vdash \overline{y} : Un}{\Gamma \vdash x\langle \overline{y} \rangle :: (x : Un), (\overline{y} : Un)}\ \textit{J-dun} \qquad \frac{\Gamma \vdash x : In\langle\!|\overline{T}|\!\rangle \qquad \Gamma \vdash \overline{y} : \overline{T}}{\Gamma \vdash x\langle \overline{y} \rangle :: (x : In\langle\!|\overline{T}|\!\rangle), (\overline{y} : \overline{T})}\ \textit{J-din}$$

Soups

$$\frac{\mathcal{P} = P_1, \ldots, P_n \qquad \Gamma \vdash P_i \text{ for each } 1..n}{\Gamma \vdash \mathcal{P}}\ \textit{P-elim}$$

$$\frac{\Gamma \vdash D_i :: \Gamma'_i \text{ for each } i = 1..n \qquad dv(D_1) \cap \ldots \cap dv(D_n) = \emptyset \qquad \Gamma' = \Gamma'_1, \ldots, \Gamma'_n}{\Gamma \vdash \mathcal{D} :: \Gamma'}\ \textit{D-elim}$$

$$\frac{\Gamma, \Gamma' \vdash \mathcal{P} \qquad \Gamma, \Gamma' \vdash \mathcal{D} :: \Gamma' \qquad \mathtt{dom}(\Gamma') = dv(\mathcal{D})}{\Gamma \vdash \mathcal{D} \Vdash \mathcal{P}}\ \textit{Soup}$$

B Proofs for the type system J_{OT}

Lemma 1 (Useless relations). *Let x be a name that is neither free nor defined in $\mathcal{T}$ or in $\Delta; \Gamma$ and let o be a valid owner in $\Delta; \Gamma$. Then:*

$$\Delta; \Gamma \vdash \mathcal{T} \Leftrightarrow \Delta, (x \prec: o); \Gamma \vdash \mathcal{T}$$

Proof. Straightforward induction on the typing judgments $\mathcal{T}$.

□

♠

Lemma 2 (Useless variables). *Let v be a name that is neither free nor defined in $\mathcal{T}$ or in Γ. Let v be valid in Δ. Let $T = {}^{o}_{v}\langle\!|\overline{t}|\!\rangle$ be a valid typing annotation in $\Delta; \Gamma$. Then:*

$$\Delta; \Gamma \vdash \mathcal{T} \Leftrightarrow \Delta; \Gamma, (v : T) \vdash \mathcal{T}$$

Proof. Straightforward induction on the typing judgments $\mathcal{T}$.

□

♠

Lemma 3 (Extra substitutions lemma for J_{OT}). *Let x be a name that is neither free nor defined in $\mathcal{T}$ or in $-, \cdot$, then:*

$$\Delta; \Gamma \vdash \mathcal{T} \iff \Delta[o/x]; \Gamma[o/x] \vdash \mathcal{T}$$

Proof. Straightforward induction on the typing judgments $\mathcal{T}$.

♠

Lemma 4 (Single variable substitution lemma for J_{OT}). *If $\Delta; \Gamma \vdash u : T[u/\alpha]$, $\Delta, (y \prec: \mathtt{own}(\mathrm{T})); \Gamma, (y : T[y/\alpha]) \vdash y : T[y/\alpha]$ and $\Delta, (y \prec: \mathtt{own}(\mathrm{T})); \Gamma, (y : T[y/\alpha]) \vdash \mathcal{T}$, then $\Delta[u/y]; \Gamma[u/y] \vdash \mathcal{T}[u/y]$.*

Proof. Straightforward induction on the typing judgments $\mathcal{T}$ often relying on Lemma 3.

♠

Proof. Proof of Theorem 1. The proof goes by induction on the number of applications of rule CTX in the derivation of the reduction.

Base case: Here we consider each reaction rule:

S-PAR: The reduction is: $\Vdash P_1 \mid P_2 \rightleftharpoons \Vdash P_1, P_2$.

Heating $\rightharpoonup$: In this case the hypothesis is $\Delta; \Gamma \vdash \Vdash P_1 \mid P_2$.
The derivation tree is the following. First we can only apply *Soup* and then the only matching rule is *P-par*:

$$\dfrac{\dfrac{\Delta; \Gamma \vdash P_1 \qquad \Delta; \Gamma \vdash P_2}{\Delta; \Gamma \vdash P_1 \mid P_2}\ \textit{P-par}}{\Delta; \Gamma \vdash \Vdash P_1 \mid P_2}\ \textit{Soup}$$

With the assumptions made in this derivation we can apply rule *P-elim* and then *Soup* to prove the thesis $\Delta; \Gamma \vdash \Vdash P_1, P_2$.

$$\dfrac{\dfrac{\Delta; \Gamma \vdash P_1 \qquad \Delta; \Gamma \vdash P_2}{\Delta; \Gamma \vdash P_1, P_2}\ \textit{P-elim}}{\Delta; \Gamma \vdash \Vdash P_1, P_2}\ \textit{Soup}$$

Cooling $\leftharpoondown$: The other way round: first apply *Soup* and *P-elim*. Now, with those hypotheses, apply *P-par* and *Soup* to prove the thesis.

S-DEF: The reduction is: $\Vdash \mathbf{def}\ D\ \mathbf{in}\ P \rightleftharpoons D \Vdash P$.

Heating $\rightharpoonup$: In this case our hypothesis is $\Delta; \Gamma \vdash \Vdash \mathbf{def}\ D\ \mathbf{in}\ P$ with side condition that $dv(D)$ are mapped to fresh names so that $dv(D) \cap fv(\mathcal{E}, \mathcal{M}) = \emptyset$.
The derivation tree is the following. First we can only apply *Soup*, then the only matching rule is *P-def*.

$$\dfrac{\dfrac{\begin{array}{l}\mathtt{dom}(\Gamma') = dv(D) = \mathtt{dom}(\Delta') \\ \dfrac{\Delta, \Delta'; \Gamma, \Gamma' \vdash D :: \Gamma'; \Delta' \qquad \Delta, \Delta'; \Gamma, \Gamma' \vdash P}{\Delta; \Gamma \vdash \mathbf{def}\ D\ \mathbf{in}\ P}\ \textit{P-def}\end{array}}{\Delta; \Gamma \vdash \Vdash \mathbf{def}\ D\ \mathbf{in}\ P}\ \textit{Soup}$$

Now we have all the premises to apply rule *Soup* and prove the thesis $\Delta; \Gamma \vdash D \Vdash P$.

$$\dfrac{\begin{array}{l}\mathtt{dom}(\Gamma') = dv(D) = \mathtt{dom}(\Delta') \\ \dfrac{\Delta, \Delta'; \Gamma, \Gamma' \vdash D :: \Gamma'; \Delta' \qquad \Delta, \Delta'; \Gamma, \Gamma' \vdash P}{\Delta; \Gamma \vdash D \Vdash P}\ \textit{Soup}\end{array}}{}$$

The side condition holds since we have that $\Delta, \Delta'; \Gamma, \Gamma' \vdash D :: \Gamma'; \Delta'$, which implies that $\Delta, \Delta'; \Gamma, \Gamma'$ is well typed. Due to this fact we know there is no double occurrence of the same variable in the environment, ergo the $dv(D)$ are fresh.

Cooling $\leftharpoondown$: The other way round: first apply *Soup*, then use the hypotheses to apply *P-def* and *Soup*.

R-BETA: The reduction is: $D \wedge J \rhd P \wedge D' \Vdash \mathcal{J} \longrightarrow D \wedge J \rhd\ P \wedge D' \Vdash P\sigma$ with the following side condition: $J \mathrel{\mathring{=}}_\sigma \mathcal{J}$.
We rewrite the left side of the soup as $D \wedge J \rhd P$ for the sake of simplicity, adding extra definitions introduces only the environments related to them..
Note that, in order to keep the proof readable, the following shorthands have been used: $\Delta'; \Gamma' \equiv \Delta, \Delta_D, \Delta_J^d; \Gamma, \Gamma_D, \Gamma_J^d$ and $\Delta''; \Gamma'' \equiv \Delta, \Delta_D, \Delta_J^d, \Delta_J^r; \Gamma, \Gamma_D, \Gamma_J^d, \Gamma_J^r$.

$$\dfrac{\dfrac{dv(D)\cap dv(J)=\emptyset \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{T_{ji}\equiv {}^{o_{ji}}_{\alpha_{ji}}\langle\!|\overline{t_{ji}}|\!\rangle}\quad \dfrac{}{T_i\equiv {}^{o_i}_{x_i}\langle\!|\exists\overline{\alpha_i}.\overline{T_i}|\!\rangle}\quad \dfrac{}{\Delta'';\Gamma''\vdash y_{ji}:T_{ji}[y_{ji}/\alpha_{ji}]}}{\dfrac{\Delta'';\Gamma''\vdash x_i:T_i}{\Delta';\Gamma'\vdash x_i\langle\overline{y_i}\rangle}\,J\text{-}def}\,J\text{-}par}{\vdots}\,J\text{-}par}{\Delta'';\Gamma''\vdash J::\Gamma_J;\Delta_J \qquad \Delta_J=\Delta_J^r,\Delta_J^d \quad \Gamma_J=\Gamma_J^r,\Gamma_J^d \quad \Delta''\Gamma''\vdash P}\,D\text{-}run}{\Delta';\Gamma'\vdash J\rhd P::\Gamma_J^d;\Delta_J^d}\,D\text{-}and}{\Delta';\Gamma'\vdash D\wedge J\rhd P::\Gamma_D,\Gamma_J^d;\Delta_D,\Delta_J^d}\,D\text{-}and \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{T_{ji}\equiv {}^{o_{ji}}_{\alpha_{ji}}\langle\!|\overline{t_{ji}}|\!\rangle}\quad \dfrac{}{T_i\equiv {}^{o_i}_{x_i}\langle\!|\exists\overline{\alpha_i}.\overline{T_i}|\!\rangle}\quad \dfrac{}{\Delta';\Gamma'\vdash u_{ji}:T_{ji}[u_{ji}/\alpha_{ji}]}}{\dfrac{\Delta';\Gamma'\vdash x_i:T_i}{\Delta';\Gamma'\vdash x_i\langle\overline{u_i}\rangle}\,P\text{-}msg}\,P\text{-}par}{\vdots}\,P\text{-}par}{\Delta';\Gamma'\vdash\mathcal{J}}\,P\text{-}par}{\Delta;\Gamma\vdash D\wedge J\rhd P\Vdash\mathcal{J}}\,Soup$$

The proof tree gives us all the required hypotheses to apply Lemma 4 for all possible pair of formal-actual parameters y_{ji}, u_{ji}. The hypotheses are the following, they hold for all $j = 1..m_i$ where m_i is the number of parameters channel i expects, and $i = 1..n$ where n is the number of parallel messages of J:

- $\Delta, \Delta_D, \Delta_J^d; \Gamma, \Gamma_D, \Gamma_J^d \vdash u_{ji} : T_{ji}[u_{ji}/\alpha_{ji}]$
- $\Delta, \Delta_D, \Delta_J^d, \Delta_J^r; \Gamma, \Gamma_D, \Gamma_J^d, \Gamma_J^r \vdash y_{ji} : T_{ji}[y_{ji}/\alpha_{ji}]$
- $\Delta, \Delta_D, \Delta_J^d, \Delta_J^r; \Gamma, \Gamma_D, \Gamma_J^d, \Gamma_J^r \vdash P$

Which allow us to conclude the thesis:

- $\Delta, \Delta_D, \Delta_J^d; \Gamma, \Gamma_D, \Gamma_J^d \vdash P[\overline{u_1}/\overline{y_1}], \ldots, [\overline{u_n}/\overline{y_n}]$

Now note that the sequence of substitutions $[\overline{u_1}/\overline{y_1}], \ldots, [\overline{u_n}/\overline{y_n}]$ is precisely the σ of the hypothesis $J \doteq_\sigma \mathcal{J}$, as one can easily see thanks to the unification algorithm described in Section 3.2. We are then allowed to conclude the thesis as follows.

$$\dfrac{\dfrac{\text{Same proof as in hp}}{\Delta';\Gamma'\vdash D\wedge J\rhd P::\Gamma_D,\Gamma_J^d;\Delta_D,\Delta_J^d} \qquad \dfrac{\text{Lemma 4}}{\Delta';\Gamma'\vdash P\sigma}}{\Delta;\Gamma\vdash D\wedge J\rhd P\Vdash P\sigma}\,Soup$$

Inductive case.

The following tree is the one we obtain from the hypothesis.

$$\dfrac{\dfrac{\dfrac{\Delta,\Delta_1,\Delta_D;\Gamma,\Gamma_1,\Gamma_D\vdash\mathcal{D}::\Gamma_D;\Delta_D \quad (*)}{\Delta,\Delta_1,\Delta_D;\Gamma,\Gamma_1,\Gamma_D\vdash\mathcal{D}_1::\Gamma_1;\Delta_1} \qquad dv(\mathcal{D})\cap dv(\mathcal{D}_1)=\emptyset}{\Delta,\Delta_1,\Delta_D;\Gamma,\Gamma_1,\Gamma_D\vdash\mathcal{D},\mathcal{D}_1::\Gamma_1,\Gamma_D;\Delta_1,\Delta_D}\,D\text{-}elim \qquad \dfrac{\Delta,\Delta_1,\Delta_D;\Gamma,\Gamma_1,\Gamma_D\vdash\mathcal{P} \quad (*)}{\Delta,\Delta_1,\Delta_D;\Gamma,\Gamma_1,\Gamma_D\vdash\mathcal{P}_1}\,P\text{-}elim}{\Delta;\Gamma\vdash\mathcal{D},\mathcal{D}_1\Vdash\mathcal{P},\mathcal{P}_1}\,Soup$$

These two are the ones we develop from the inductive hypothesis.

$$\dfrac{\Delta,\Delta_1;\Gamma,\Gamma_1\vdash\mathcal{D}_1::\Gamma_1;\Delta_1 \qquad \Delta,\Delta_1;\Gamma,\Gamma_1\vdash\mathcal{P}_1}{\Delta;\Gamma\vdash\mathcal{D}_1\Vdash\mathcal{P}_1}\,Soup$$

$$\dfrac{\Delta'',\Delta_2;\Gamma'',\Gamma_2\vdash\mathcal{D}_2::\Gamma_2;\Delta_2 \qquad \Delta'',\Delta_2;\Gamma'',\Gamma_2\vdash\mathcal{P}_2}{\Delta'';\Gamma''\vdash\mathcal{D}_2\Vdash\mathcal{P}_2}\,Soup$$

Where $\Delta;\Gamma$ and $\Delta'';\Gamma''$ are the same except for the variables defined in $\mathcal{D}_1, \mathcal{P}_1$ only (which are in $\Delta;\Gamma$) and those defined only in $\mathcal{D}_2, \mathcal{P}_2$ (which are in $\Delta'';\Gamma''$). All the common variable appear both in $\Delta;\Gamma$ and $\Delta'';\Gamma''$ and have the same type since the environments agree in the sense of Definition 2.

Now we first apply Lemma 1 to add relations of $\mathcal{D}_2, \mathcal{P}_2$ to the previously augmented environment. We already have all the required hypotheses to apply

the Lemma since when we add a relation $(o \prec: o')$ we only need o' to be an already defined context. We can exploit the tree-structure imposed by $\prec:$ in contexts to add a relation where o' is an already defined context. Secondly we apply Lemma 2 to enrich the environments of $\mathcal{D}_2, \mathcal{P}_2$ with the channels defined in $\mathcal{D}, \mathcal{P}$. Also in this case we have all the hypotheses. The variables are not in the environment already due to the side condition of the evolution rule *CTX*. The fact that the context associated to the variable must be in the environment is granted by the application of the previous Lemma. The type of the variable is valid since all the relations used to typecheck it have already been added to the environment with the previous Lemma.

We obtain the following hypotheses.

- $\Delta'', \Delta_2, \Delta_D; \Gamma'', \Gamma_2, \Gamma_D \vdash \mathcal{D}_2 :: \Gamma_2; \Delta_2$
- $\Delta'', \Delta_2, \Delta_D; \Gamma'', \Gamma_2, \Gamma_D \vdash \mathcal{P}_2$

Now we first apply Lemma 2, then Lemma 1 to the hypotheses marked with $(*)$ to remove all variables related to $\mathcal{D}_1, \mathcal{P}_1$ only from the environment. All the hypotheses needed by each Lemma are matched since we are applying them in the reverse order with respect to when we used them to eliminate variables.

In the end we use again the same Lemmas but in the order needed for addition (Lemma 1, Lemma 2) to add the relations concerning variables, and channels, used in $\mathcal{D}_2, \mathcal{P}_2$ to the typechecking environments of $\mathcal{D}, \mathcal{P}$.

We obtain the following hypotheses.

- $\Delta'', \Delta_2, \Delta_D; \Gamma'', \Gamma_2, \Gamma_D \vdash \mathcal{D} :: \Gamma_D; \Delta_D$
- $\Delta'', \Delta_2, \Delta_D; \Gamma'', \Gamma_2, \Gamma_D \vdash \mathcal{P}$

Due to the side condition of the evolution rule *CTX* we have the hypothesis $dv(\mathcal{D}) \cap dv(\mathcal{D}_2) = \emptyset$.

We can now prove the thesis.

$$\dfrac{\dfrac{\begin{array}{c}\Delta'', \Delta_2, \Delta_D; \Gamma'', \Gamma_2, \Gamma_D \vdash \mathcal{D} :: \Gamma_D; \Delta_D \\ \Delta'', \Delta_2, \Delta_D; \Gamma'', \Gamma_2, \Gamma_D \vdash \mathcal{D}_2 :: \Gamma_2; \Delta_2 \\ dv(\mathcal{D}) \cap dv(\mathcal{D}_2) = \emptyset\end{array}}{\Delta'', \Delta_2, \Delta_D; \Gamma'', \Gamma_2, \Gamma_D \vdash \mathcal{D}, \mathcal{D}_2 :: \Gamma_2, \Gamma_D; \Delta_2, \Delta_D}\ D\text{-}elim \quad \dfrac{\begin{array}{c}\Delta'', \Delta_2, \Delta_D; \Gamma'', \Gamma_2, \Gamma_D \vdash \mathcal{P} \\ \Delta'', \Delta_2, \Delta_D; \Gamma'', \Gamma_2, \Gamma_D \vdash \mathcal{P}_2\end{array}}{\Delta'', \Delta_2, \Delta_D; \Gamma'', \Gamma_2, \Gamma_D \vdash \mathcal{P}, \mathcal{P}_2}\ P\text{-}elim}{\Delta''; \Gamma'' \vdash \mathcal{D}, \mathcal{D}_2 \Vdash \mathcal{P}, \mathcal{P}_2}\ Soup$$

□

♠

Proof. Proof of Theorem 2. There are three type of runtime errors, we shall present how none of them are allowed in a well-typed soup.

A message $x\langle\overline{y}\rangle$ where x is not defined. A requirement for a message emission $x\langle\overline{y}\rangle$ to be well-typed is that the sending channel x must be in the typing environments. The only way to add an element to the environments is defining a channel, so well-typed soups send messages only on defined channels.

A message $x\langle\overline{y}\rangle$ that is defined with a different arity. The type of a channel contains the arity of the parameters it expects to send. Recall there is only one occurrence of a binding for a channel to a type in the environment. This implies that the type of a channel in rule *J-def* and *P-msg* is the same. This fact does not allow a message with an arity mismatch to be well-typed.

Breaking of the *owners-as-dominators* property. This case holds due to a combination of Theorem 3 and Theorem 1. A well typed soup in fact enjoys the results of Theorem 3, so it does not break the *owners-as-dominators* property. Since well-typedness is preserved by reduction, as Theorem 1 states, the evolution of the starting soup will enjoy Theorem 3.

Since well-typedness is preserved due to Theorem 1, all the soup $\mathcal{D}' \Vdash \mathcal{P}'$ the starting soup can evolve into are well typed. Since they are well typed, they contain no error.

□

♠

Proof. Proof of Theorem 3.

The proof is straightforward. By assuming that $x\langle y\rangle$ is valid, we can develop the following tree. To prove the thesis we just need to apply Lemma 2 to the boxed hypothesis and turn Γ' in Γ. Such an application is allowed since all the requirements are met. We indicate owner of y with o'.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dots \quad \boxed{\Delta;\Gamma' \vdash x \prec:^* o'} \quad \dots}{\Delta;\Gamma' \vdash \diamond \quad \dots \quad \Delta;\Gamma' \vdash {}^{o}_{x}\langle\!\langle \exists\alpha.T \rangle\!\rangle}\,\textit{Type}}{\Delta;\Gamma', (x : {}^{o}_{x}\langle\!\langle \exists\alpha.T \rangle\!\rangle) \vdash \diamond \quad \dots}\,\textit{Env-c}}{\vdots}\,\textit{Env-c}}{\Delta;\Gamma \vdash \diamond \quad \dots}\,\textit{Env-c}}{\Delta;\Gamma \vdash x : {}^{o}_{x}\langle\!\langle \exists\alpha.T \rangle\!\rangle}\,\textit{Chan} \qquad \Delta;\Gamma \vdash y : T[y/\alpha]}{\Delta;\Gamma \vdash x\langle y\rangle}\,\textit{P-msg}$$

□

♠

Lemma 5. *Consider a well-typed soup $\mathcal{D} \Vdash \mathcal{P}$ that defines a channel y owned by o. y is not accessible from channels whose owner is not* transitively inside *o.*

Proof. The proof proceeds by reductio ad absurdum. Note that we will use the terminology *outside* as a synonym for *not transitively inside*.

Consider a channel z whose owner o' is outside o. Transitively we know also that z is outside o. Suppose z can have access to x, that means there exist an environment $\Delta;\Gamma$ such that $\Delta;\Gamma \vdash z\langle x\rangle$. The following proof tree derives from the just stated hypothesis.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dots \quad \boxed{\Delta;\Gamma' \vdash z \prec:^* o} \quad \dots}{\dots \quad \Delta;\Gamma' \vdash {}^{o'}_{z}\langle\!\langle \exists\alpha.\, {}^{o}_{\alpha}\langle\!\langle \dots \rangle\!\rangle \rangle\!\rangle}\,\textit{Type}}{\Delta;\Gamma', (z : {}^{o'}_{z}\langle\!\langle \exists\alpha.\, {}^{o}_{\alpha}\langle\!\langle \dots \rangle\!\rangle \rangle\!\rangle) \vdash \diamond}\,\textit{Env-c}}{\vdots}\,\textit{Env-c}}{\Delta;\Gamma \vdash z : {}^{o'}_{z}\langle\!\langle \exists\alpha.\, {}^{o}_{\alpha}\langle\!\langle \dots \rangle\!\rangle \rangle\!\rangle}\,\textit{Chan} \qquad \Delta;\Gamma \vdash x : {}^{o}_{\alpha}\langle\!\langle \dots \rangle\!\rangle[x/\alpha]}{\Delta;\Gamma \vdash z\langle x\rangle}\,\textit{P-msg}$$

The boxed hypothesis is not provable since we are currently assuming that z is not transitively inside $(\not\prec:^*)$ o.

□

Proof. Proof of Corollary 1. Due to Lemma 5, we know that a well-typed soup contains no access to a channel y owned by o from channels whose owner is not transitively inside o. Theorem 1 ensures that well-typedness is preserved, so, for every soup $\mathcal{D}' \Vdash \mathcal{P}'$ the starting soup $\mathcal{D} \Vdash \mathcal{P}$ can evolve into, the results of Lemma 5 apply as well.

□

♠

C Proofs for J_{AU}

Lemma 6 (Single variable substitution lemma for J_{AU}). *If $\Gamma \vdash u : T$, $\Gamma, \Gamma' \vdash y : T$ and $\Gamma, \Gamma' \vdash P$, (where $\Gamma' = (y : T)$) then $\Gamma \vdash P[u/y]$.*

Proof. Identical to that of Lemma 4.

□

♠

Lemma 7 (Weakening lemma for J_{AU}). *Let v be a name that is not free nor defined in $\mathcal{T}$ or in Γ. Let T be a valid typing annotation in Γ.*

$$\Gamma \vdash \mathcal{T} \Leftrightarrow \Gamma, (v : T) \vdash \mathcal{T}$$

Proof. Identical to that of Lemma 2.

□

♠

Proof. Proof of Theorem 4. The proof is analogous to that of Theorem 1. It also needs a lemma for substitution: Lemma 6 and one for useless binding addition and removal: Lemma 7.

Proofs are omitted since they introduce no new concepts.

□

♠

D Proofs for untyped opponent

Lemma 8. *For all untyped process P, if $fv(P) = \{x_1, \ldots, x_n\}$, then $\emptyset, x_1 : Un, \ldots, x_n : Un \vdash P$.*

Proof. Straightforward induction on P.

□

♠

Proof. Proof of Proposition 1.

Straightforward induction on $\mathcal{D} \Vdash \mathcal{P}$ with application of Lemma 8.

□

♠

Lemma 9. *if $\Delta; \Gamma \vdash \diamond$ then $[\![\Delta; \Gamma]\!]_o \vdash \diamond$.*

Proof. Straightforward induction on the judgment $\diamond$.

□

♠

Lemma 10. *If* $\Delta; \Gamma \vdash P$ *then* $[\![\Delta; \Gamma]\!]_o \vdash \texttt{erase}(\mathrm{P})$.

Proof. Straightforward induction on P, use Lemma 9.

□

♠

Proof. Proof of Proposition 2.

Straightforward induction on $\mathcal{D} \Vdash \mathcal{P}$ with application of Lemma 10 and Lemma 9.

□

♠

Proof. Proof of Proposition 3. The proof goes on by reductio ad absurdum. Suppose there exists a finite sequence of reduction steps $\rightleftharpoons^*$ such that $\mathcal{D} \Vdash \mathcal{P} \rightleftharpoons^* \mathcal{D}' \Vdash \mathcal{P}', y_i\langle x\rangle$.

Theorem 4 tells us that typing is preserved in the auxiliary type system. Let $\Gamma = \emptyset, y_1 : Un, \ldots, y_n : Un, o : T', x : T$. So:

$$\emptyset, y_1 : Un, \ldots, y_n : Un, o : T', x : T \vdash \mathcal{D} \Vdash \mathcal{P} \Rightarrow \exists \Gamma'.\Gamma' \vdash \mathcal{D}' \Vdash \mathcal{P}', y_i\langle x\rangle$$

Furthermore we know that Γ and Γ' are the same except for the names defined in $\mathcal{D}$ only or for those defined in $\mathcal{D}'$ only. This means that $y_i : Un \in \Gamma'$ and $x : In\langle\!|\ldots|\!\rangle \in \Gamma'$. We are then allowed to develop the following proof tree:

$$\dfrac{\Gamma', \Gamma'' \vdash \mathcal{D}' :: \Gamma'' \qquad \dfrac{\Gamma', \Gamma'' \vdash \mathcal{P}' \qquad \dfrac{\Gamma', \Gamma'' \vdash y_i : Un \qquad \boxed{\Gamma', \Gamma'' \vdash x : Un}}{\Gamma', \Gamma'' \vdash y_i\langle x\rangle}\ P\text{-}msg}{\Gamma', \Gamma'' \vdash \mathcal{P}', y_i\langle x\rangle}\ P\text{-}elim}{\Gamma' \vdash \mathcal{D}' \Vdash \mathcal{P}', y_i\langle x\rangle}\ Soup$$

The boxed hypothesis is false and cannot be inferred from Γ', Γ'' since we have $(x : In\langle\!|\ldots|\!\rangle) \in \Gamma'$. The absurdum is stated, ergo no finite sequence of reduction steps $\rightleftharpoons^*$ that satisfies the hypothesis exists.

□

♠

Lemma 11 (Soup combination). *Consider two well typed soups* $\Gamma \vdash_{AU} \mathcal{D} \Vdash \mathcal{P}$ *and* $\Gamma' \vdash_{AU} \mathcal{D}' \Vdash \mathcal{P}'$ *such that* $\Gamma \frown \Gamma'$. *Then* $\Gamma \cup \Gamma' \vdash_{AU} \mathcal{D}.\mathcal{D}' \Vdash \mathcal{P}, \mathcal{P}'$.

Proof. Proof of Lemma 11.

Straightforward induction on the soup $\mathcal{D}' \Vdash \mathcal{P}'$, with application of Lemma 7.

□

♠

Proof. Proof of Theorem 5. The theorem is proven by reduction ad absurdum.

Proposition 2 tells us $[\![\Delta; \Gamma]\!]_o \vdash_{AU} \texttt{erase}(\mathcal{D} \Vdash \mathcal{P})$. By definition of $\texttt{erase}(\ )$ we know that $\texttt{erase}(\mathcal{D} \Vdash \mathcal{P}) = \texttt{erase}(\mathcal{D}) \Vdash \texttt{erase}(\mathcal{P})$.

Let $fv(\mathcal{D}' \Vdash \mathcal{P}') = \{y_1, \ldots, y_n\}$. Proposition 1 tells us that $\emptyset, y_1 : Un, \ldots, y_n : Un \vdash_{AU} \mathcal{D}' \Vdash \mathcal{P}'$.

So far we have two soups: $\texttt{erase}(\mathcal{D}) \Vdash \texttt{erase}(\mathcal{P})$ and $\mathcal{D}' \Vdash \mathcal{P}'$, each one with a typechecking environment in J_{AU}. Thanks to Lemma 11, we can concatenate the

environments and the soups while still preserving their well typedness. Formally: $\emptyset, y_1 : Un, \ldots, y_n : Un, \cup[\![\Delta; \Gamma]\!]_o \vdash_{AU} \mathcal{D}', \texttt{erase}(\mathcal{D}) \Vdash \mathcal{P}', \texttt{erase}(\mathcal{P})$.

Now Proposition 3 tells us that no $\rightleftharpoons^*$ exists for soup $\mathcal{D}', \texttt{erase}(\mathcal{D}) \Vdash \mathcal{P}', \texttt{erase}(\mathcal{P})$ to end up leaking secrets.

□

♠