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

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. Secrecy is also preserved in the context of an untyped opponent.

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.

⋆ Sangiorgi's reasearch partly supported by the MIUR project IPODS and the EU project FP7-231620 HATS

R. Bruni and J. Dingel (Eds.): FMOODS/FORTE 2011, LNCS 6722, pp. 289–303, 2011.
© IFIP International Federation for Information Processing 2011

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.

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

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 $J \rhd P$, 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.

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 send 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 $\langle x, x\rangle : \exists\alpha.T[\alpha/x]$.

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

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

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

$$
\begin{array}{rl}
P = \mathbf{def}\ o\langle a\rangle \rhd \emptyset \\
\wedge\quad x\langle\rangle \mid y\langle\rangle \rhd R \\
\mathbf{in}\quad o\langle x\rangle \mid o\langle y\rangle
\end{array}
\qquad
\begin{array}{rl}
P_T = \mathbf{def}\ o\langle a\rangle : {}^{world}_{o}\langle\!| \exists\alpha.\ {}^{o}_{\alpha}\langle\!||\!\rangle |\!\rangle \rhd \emptyset \\
\wedge\quad x\langle\rangle : {}^{o}_{x}\langle\!||\!\rangle \mid y\langle\rangle : {}^{o}_{y}\langle\!||\!\rangle \rhd R \\
\mathbf{in}\quad o\langle x\rangle \mid o\langle y\rangle
\end{array}
$$

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 $J \mathrel{\hat{=}}_{\sigma} \mathcal{J}$ 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.

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

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*.

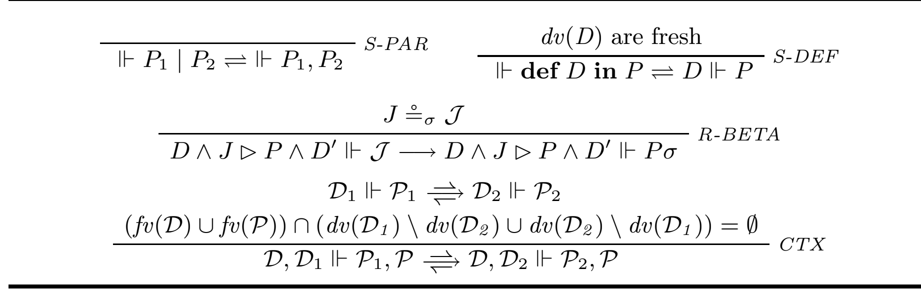

Fig. 3. Chemical rules for the RCHAM

Two kind of rules describe the evolution of the soup. Structural rules, *heating* $\rightharpoonup$, and *cooling* $\leftharpoondown$, (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 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 $\rightleftharpoons\!\!\!\!\rightarrow$ 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 $\texttt{dom}(\Delta)$ is define inductively as follows: $\texttt{dom}(\emptyset) = \emptyset$, $\texttt{dom}(\Delta, (o \prec: o')) = \texttt{dom}(\Delta) \cup \{o\}$.

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 $(x : {}^{o}_{x}\langle\!|\overline{t}|\!\rangle) \in \Gamma$.*

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 $(x : {}^{o}_{x}\langle\!|\overline{t}|\!\rangle) \in \Gamma$.

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

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

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

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

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

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

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: $\mathtt{own}(\,{}^{o'}_{o}\langle\!|\overline{t}|\!\rangle) = o'$, $\mathtt{own}(\exists\alpha.T) = \mathtt{own}(T)$.

Types

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

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

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

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

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

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

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

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

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

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

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

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

Soups

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

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

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

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.

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

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

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

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 (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.

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

Types that do not mention the secret o are translated as untrusted: Un. All others preserve their structure and are translated as trusted: $In\langle\!\langle\overline{T}\rangle\!\rangle$. 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.

$$P_G = \nu G \quad \mathbf{def}\ d\langle\overline{a}\rangle : T_d \mid c\langle\overline{b}\rangle : T_c \triangleright R \quad \mathbf{in}\ \ \ldots$$

$$P_O = \mathbf{def}\ d\langle\overline{a}\rangle :\ {}^{world}_{d}\langle\!|\,|\!\rangle \mid c\langle\overline{b}\rangle :\ {}^{world}_{c}\langle\!|\,|\!\rangle \triangleright R \quad \mathbf{in}\ \ \ldots$$

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. Abadi, M.: Security protocols and specifications. In: Thomas, W. (ed.) FOSSACS 1999. LNCS, vol. 1578, pp. 1–13. Springer, Heidelberg (1999)
2. Abadi, M., Fournet, C., Gonthier, G.: Secure implementation of channel abstractions. Inf. Comput. 174, 37–83 (2002)
3. Abadi, M., Gordon, A.D.: A calculus for cryptographic protocols: The spi calculus. Inf. Comput. 148(1), 1–70 (1999)

4. Bodei, C., Degano, P., Nielson, F., Riis Nielson, H.: Control flow analysis for the π-calculus. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 84–98. Springer, Heidelberg (1998)
5. Boyapati, C., Lee, R., Rinard, M.C.: Ownership types for safe programming: preventing data races and deadlocks. In: OOPSLA 2002, pp. 211–230 (2002)
6. Boyapati, C., Liskov, B., Shrira, L.: Ownership types for object encapsulation. In: POPL 2003, pp. 213–223 (2003)
7. Boyapati, C., Salcianu, A., Beebee, W.S., Rinard, M.C.: Ownership types for safe region-based memory management in real-time java. In: PLDI 2003, pp. 324–337 (2003)
8. Cardelli, L., Ghelli, G., Gordon, A.D.: Secrecy and group creation. Inf. Comput. 196(2), 127–155 (2005)
9. Clarke, D.: Object Ownership and Containment. PhD thesis, University of New South Wales (July 2001)
10. Clarke, D., Drossopoulou, S.: Ownership, encapsulation and the disjointness of type and effect. In: OOPSLA 2002, pp. 292–310 (2002)
11. Clarke, D., Potter, J., Noble, J.: Ownership types for flexible alias protection. In: OOPSLA 1998, pp. 48–64 (1998)
12. Clarke, D., Richmond, M., Noble, J.: Saving the world from bad beans: deployment-time confinement checking. In: OOPSLA 2003, pp. 374–387. ACM, New York (2003)
13. Dolev, D., Yao, A.C.: On the security of public key protocols. In: SFCS 1981, pp. 350–357. IEEE Computer Society Press, Washington, DC, USA (1981)
14. Fournet, C., Fessant, F.L., Maranget, L., Schmitt, A.: JoCaml: a Language for Concurrent Distributed and Mobile Programming. In: Proceedings of the 4th Summer School on Advanced Functional Programming. LNCS, pp. 129–158. Springer, Heidelberg (2002)
15. Fournet, C., Gonthier, G.: The reflexive CHAM and the Join-calculus. In: POPL 1996, pp. 372–385 (1996)
16. Fournet, C., Gonthier, G.: The join calculus: A language for distributed mobile programming. In: Barthe, G., Dybjer, P., Pinto, L., Saraiva, J. (eds.) APPSEM 2000. LNCS, vol. 2395, pp. 268–332. Springer, Heidelberg (2002)
17. Fournet, C., Laneve, C., Maranget, L., Rémy, D.: Implicit typing à la ML for the Join-calculus. In: Mazurkiewicz, A., Winkowski, J. (eds.) CONCUR 1997. LNCS, vol. 1243, pp. 196–212. Springer, Heidelberg (1997)
18. Haller, P., Van Cutsem, T.: Implementing joins using extensible pattern matching. In: Wang, A.H., Tennenholtz, M. (eds.) COORDINATION 2008. LNCS, vol. 5052, pp. 135–152. Springer, Heidelberg (2008)
19. Krishnaswami, N.R., Aldrich, J.: Permission-based ownership: encapsulating state in higher-order typed languages. In: PLDI 2005, pp. 96–106 (2005)
20. Patrignani, M., Clarke, D., Sangiorgi, D.: Ownership types for the Join calculus. CW Reports CW603, Dept. of Computer Science, , K.U.Leuven (March 2011)
21. Pierce, B.: Types and Programming Languages. MIT Press, Cambridge (2002)
22. Plociniczak, H., Eisenbach, S.: JErlang: Erlang with joins. In: Clarke, D., Agha, G. (eds.) COORDINATION 2010. LNCS, vol. 6116, pp. 61–75. Springer, Heidelberg (2010)
23. Vitek, J., Bokowski, B.: Confined types. In: OOPSLA 1999, Denver, Colorado, United States, pp. 82–96. ACM Press, New York (1999)
24. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Inf. Comput. 115, 38–94 (1994)