Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Electronic Notes in Theoretical Computer Science 192 (2008) 3–22
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

A Parametric Calculus for Mobile Open Code

Davide Ancona[2], Sonia Fagorzi[3] and Elena Zucca[4]

DISI
University of Genova, Italy

Abstract

We present a simple parametric calculus of processes which exchange *open* mobile code, that is, code which may contain free variables to be bound by the receiver's code.
Type safety is ensured by a combination of static and dynamic checks. That is, internal consistency of each process is statically verified, by relying on local type assumptions on missing code; then, when code is sent from a process to another, a runtime check based on a subtyping relation ensures that it can be successfully received, without requiring re-inspection of the code. In order to refuse communication in as few cases as possible, the runtime check accepts even mobile code which would be rejected if statically available, by automatically inserting *coercions* driven by the subtyping relation, as in the so-called *Penn translation*.
The calculus is parametric in some ingredients which can vary depending on the specific language or system. Notably, we abstract away from the specific nature of the code to be exchanged, and of the static and dynamic checks. We formalize the notion of *type safety* in our general framework and provide sufficient conditions on the above ingredients which guarantee this property.
We illustrate our approach on a simple lambda-calculus with records, where type safe exchange of mobile code is made problematic by conflicts due to components which were not explicitly required. In particular, we show that the standard coercion semantics given in the literature, with other aims, for this calculus, allows to detect and eliminate conflicts due to inner components, thus solving a problem which was left open in previous work on type-safe exchange of mobile code.

Keywords: Process calculi, mobile code, rebinding, dynamic typing, subtyping

Introduction

In a previous paper [8], we have presented a parametric calculus of processes which exchange mobile code in a type-safe manner. This calculus, built on a simple coordination mechanism with standard send/receive primitives, formalizes in a language-independent setting the ideas advocated in MoMi [3,4,2]:

- Each process statically checks type safety of its local code, by relying on requirements on missing code, formally expressed by types.

[1] Partially supported by MIUR EOS DUE - Extensible Object Systems for Dynamic and Unpredictable Environments.

[2] Email: davide@disi.unige.it
[3] Email: fagorzi@disi.unige.it
[4] Email: zucca@disi.unige.it

doi:10.1016/j.entcs.2008.10.024

- Mobile code exchanged among processes is equipped with its type, obtained by the previous phase.
- Dynamic checks ensure that code sent from a process to another is accepted only if it satisfies receiver's requirements.
- Hence, whenever code is accepted, it can be safely composed with local code without being inspected again.

The calculus is parametric in some ingredients which can vary depending on the specific language or system. Notably, we abstract away from the specific nature of the code to be exchanged (modeled by a *core calculus*), and of the static and dynamic checks.

We consider two distinct subtyping relations in our framework: the *static subtyping* relation simply models subtyping which could be possibly provided by the static type system, whereas dynamic checks are modeled by a *dynamic subtyping* relation, which is intuitively expected to be more liberal. Indeed, in order to refuse communication in as few cases as possible, the runtime check accepts even mobile code which would be rejected if statically available, by automatically inserting *coercions* driven by the dynamic subtyping relation. In this way, mobile code exchange is both *safe*, since after coercion code has a statically permitted type, and *flexible*, since more code can be accepted.

In this paper we extend this previous work in two respects.

First, and more importantly, we extend the above ideas to the case where mobile code is *open*, that is, may contain free variables to be rebound in receiver's code. To this end, the send primitive explicitly specifies a set of *unbinders*, that is, which variables in sent code have to be remotely bound, possibly discarding their local definitions, if any; and the receive primitive, conversely, specifies a set of *rebinders*, that is, which variables are allowed to be free in code to be received, also providing corresponding local definitions. That is, the unbinding/rebinding mechanism is controlled by the programmer (no accidental captures may happen), analogously to what has been proposed, e.g., in [6].

Mobile code is now equipped with, besides its type, a type context specifying expected types for free variables. The runtime check becomes symmetric, since mobile code must satisfy receiver's requirements, and conversely the receiver must provide appropriate definitions for the free variables. More interestingly, coercions are inserted in both directions as well.

Second, we realized that our approach for modeling flexible and type safe mobile code exchange, that is, by coercions driven by a subtyping relation, is the same which can be used, mainly with performance reasons, for compiling source code with subtyping in lower-level code without subtyping, see Sect.15.6 of [13]. In this context, the translation which inserts run-time coercions is often called the *Penn translation*, after the group at the University of Penn that first studied it [7]. Recognizing this coincidence leaded to a much cleaner presentation of our framework. Moreover, and more substantially, in one classical case-study, that is,

when mobile code to be exchanged has a record-based structure [5], and type safe exchange of mobile code is made problematic by conflicts due to components which were not explicitly required, choosing a runtime check based on the Penn translation found in the literature allows to simply and nicely express detection and elimination of conflicts due to arbitrarily nested components, whereas in previous work on type safe exchange of mobile code [4,8] only top-level conflicts were considered.

The rest of the paper is organized as follows: we first present the untyped version of our calculus in Sect.1, then add static and dynamic checks in Sect.2. We formalize the notion of *type safety* in our parametric framework and provide sufficient conditions on the ingredients to be provided as arguments which guarantee this property. In Sect.3 we formally define an instantiation which takes a simple lambda-calculus with records as core calculus, and coercions which delete, at any nested level, components which were not explicitly required [6]. Finally, in Sect.4 we summarize our contribution and briefly discuss related and further work.

1 The Untyped Calculus

The untyped calculus for exchange of mobile open code is defined in a parametric way on top of a *core calculus* providing the following ingredients:

- *variables* $x, y, z, \ldots \in \mathsf{Var}$;
- *(core) expressions* $e \in \mathsf{Exp^c}$, with $\mathsf{Var} \subseteq \mathsf{Exp^c}$; a *substitution* ρ is a mapping from variables into (core) expressions, written $x_i \overset{i\in I}{\mapsto} e_i$, ;
- *free variables* $FV(e)$ of an expression e;
- application of a substitution ρ to an expression e, written $e\{\rho\}$;
- *(core) reduction relation* $e \xrightarrow{\mathsf{c}} e'$.

The syntax is given in Fig.1. Since the focus of our framework is on dynamic retrieval and typechecking of open code, we consider a very simple coordination mechanism based on standard synchronous send/receive primitives. In particular, a process can be, besides a process variable, the null process nil, a parallel composition of processes, a sending or a receiving process. A process $\mathsf{send}([x_i^{i\in I}]E).p$ sends open code E (which can be either core code or in turn a process) with free variables (contained in) $x_i^{i\in I}$. Conversely, a process $\mathsf{receive}(x[x_i \overset{i\in I}{\mapsto} E_i]).p$ receives open code, say E, and makes it close by binding free variables in E as specified by the substitution $x_i \overset{i\in I}{\mapsto} E_i$ (a mapping from variables into expressions); the resulting code is available in the subsequent process p via x. Note that we keep the language as simple as possible, hence do not consider additional syntactic constructs (e.g., let-in) which could be useful in practice, but are not significant to our aim.

We will use the following notations for mappings (e.g., substitutions): $\rho \setminus x$ is the map obtained from ρ by removing the association for x (if present); ρ_1, ρ_2 is the

[5] For instance, when exchanging records, objects, classes, mixins: in this paper we will study the problem in the more foundational context of records for simplicity.

[6] Corresponding, as explained above, to the Penn translation found in the literature.

$$
\begin{array}{lll}
p \in \mathsf{Proc} ::= & x \mid \mathsf{nil} \mid p_1 \| p_2 \mid & \textbf{process} \\
& \mathsf{send}([\upsilon]E).p \mid \mathsf{receive}(x[\rho]).p & \\
E \in \mathsf{Exp} ::= & e \mid p & \textbf{mobile code} \\
\upsilon ::= & x_i^{i \in I} & \textbf{unbinding} \\
\rho ::= & x_i \overset{i \in I}{\mapsto} E_i & \textbf{rebinding} \\
\lambda ::= & \tau \mid ![\upsilon]E \mid ?[\upsilon]E & \textbf{label} \\
\overline{![\upsilon]E} = & ?[\upsilon]E & \textbf{complement} \\
\overline{?[\upsilon]E} = & ![\upsilon]E &
\end{array}
$$

Fig. 1. Untyped calculus: syntax

union of substitutions ρ_1 and ρ_2 with disjoint domains. Moreover, we will use the following abbreviations:

- $\mathsf{send}(E).p$ for $\mathsf{send}([\,]E).p$, that is, when sent code is closed,
- $\mathsf{receive}(x).p$ for $\mathsf{receive}(x[\,]).p$, that is, when received code must be closed,
- $\mathsf{receive}(x[\rho, y]).p$ for $\mathsf{receive}(x[\rho, y \mapsto y]).p$, that is, when a variable in received code is bound to an outer binder in local code (see below).

Reduction semantics of process terms is modeled by a labelled relation $p \xrightarrow{\lambda} p'$ where the label is either τ, denoting an internal step, or $![\upsilon]E$, $?[\upsilon]E$, denoting, respectively, sending and receiving an expression E with free variables υ. An internal step occurs as effect of either a reduction step at the core level, or an exchange of code in a parallel composition of processes (see below).

We denote by $\overline{\lambda}$ the complement of λ, defined for $\lambda \neq \tau$ in the usual way. Moreover, we will use on labels the same abbreviations used for processes and write $?E$ and $!E$ when υ is empty.

Before giving the formal reduction rules, we illustrate how exchange of mobile code works by some examples.

First of all, consider the following parallel composition:

$$\mathsf{send}([x]x + 1).\mathsf{nil} \parallel \mathsf{receive}(y[x \mapsto 2]).\mathsf{send}(y).\mathsf{nil}$$

The left-side process sends open code $x+1$, whereas the right-side process is willing to receive code with a free variable x to be locally bound to 2. As a result of synchronization between the two processes, the right-side process replaces y by the code sent by the left-side process, where x has been in turn replaced by 2, hence $2 + 1$ is then sent. Formally we have the following reduction sequence:

$$\mathsf{send}([x]x + 1).\mathsf{nil} \parallel \mathsf{receive}(y[x \mapsto 2]).\mathsf{send}(y).\mathsf{nil} \xrightarrow{\tau}$$

$$\mathsf{nil} \parallel \mathsf{send}(2 + 1).\mathsf{nil} \xrightarrow{!2+1} \mathsf{nil} \parallel \mathsf{nil}$$

Note that in the calculus there are three different kinds of binders: in a process $\mathsf{receive}(x[x_i \overset{i\in I}{\mapsto} E_i]).p$, x binds subsequent local code p, whereas the $x_i^{i\in I}$ will (re)bind dynamically received code; in a process $\mathsf{send}([x_i^{i\in I}]E).p$, the $x_i^{i\in I}$ bind sent code E, in such a way that free occurrences of $x_i^{i\in I}$ are unbound from their local binders, if any. We will call these three kinds of binders *local binders*, *rebinders*, and *unbinders*, respectively. In the process p above, the first occurrence of x is an unbinder, the first occurrence of y is a local binder, and the third occurrence of x is a rebinder.

A local binder can also affect subsequent dynamically received code, when it binds free variables in a rebinding ρ, as shown by the following example:

$$\mathsf{receive}(x).\mathsf{receive}(y[x]).\mathsf{send}(y+x).\mathsf{nil} \xrightarrow{?2}$$
$$\mathsf{receive}(y[x \mapsto 2]).\mathsf{send}(y+2).\mathsf{nil} \xrightarrow{?[x]x*3}$$
$$\mathsf{send}(2*3+2).\mathsf{nil}$$

In this example, note the use of the abbreviation $y[x]$, which means that free variable x in received code will be bound to a definition which is still to be received as well. This abbreviation formally stands for $y[x \mapsto x]$. It is also worth noting that, since the process $\mathsf{send}(y+x).\mathsf{nil}$ has no unbinders specified, both y and x must be locally replaced before sending the code; compare with the following reduction sequence where x is unbound instead.

$$\mathsf{receive}(x).\mathsf{receive}(y[x]).\mathsf{send}([x]y+x).\mathsf{nil} \xrightarrow{?2}$$
$$\mathsf{receive}(y[x \mapsto 2]).\mathsf{send}([x]y+x).\mathsf{nil} \xrightarrow{?[x]x*3}$$
$$\mathsf{send}([x]2*3+x).\mathsf{nil}$$

The following example illustrates the case where mobile code is in turn a process.

$$\mathsf{receive}(x).\mathsf{send}(\ \ \mathsf{receive}(y[x]).\mathsf{send}(y+x).\mathsf{nil}\ \ ).\mathsf{nil} \xrightarrow{?1}$$
$$\mathsf{send}(\ \ \mathsf{receive}(y[x \mapsto 1]).\mathsf{send}(y+1).\mathsf{nil}\ \ ).\mathsf{nil}$$

Finally, the following example shows that a local binder can affect not only dynamically received code but also, in case process code is received, code dynamically received by this code, and so on.

$$\mathsf{receive}(x).\mathsf{receive}(y[x]).y \xrightarrow{?1}$$
$$\mathsf{receive}(y[x \mapsto 1]).y \xrightarrow{?[x]\mathsf{send}(x+2).\mathsf{receive}(z[x]).z}$$
$$\mathsf{send}(1+2).\mathsf{receive}(z[x \mapsto 1]).z$$

Before formally defining the reduction relation, we extend, in Fig.2, the definitions of free variables and application of a substitution, provided as ingredients at the core level, to mobile code. We denote by ρ^{c} the subset of substitution ρ mapping variables into core expressions. Conditions $\upsilon \cap FV(\rho) = \emptyset$ and $x \notin FV(\rho)$ avoid unexpected captures of free variables in ρ.

Reduction rules are defined in Fig.3. Rules (CORE-SEND) and (CORE-RCV) allow reduc-

p	$FV(p)$
x	$\{x\}$
nil	$\emptyset$
$\mathsf{send}([\upsilon]E).p$	$(FV(E) \setminus \upsilon) \cup FV(p)$
$\mathsf{receive}(x[\rho]).p$	$FV(\rho) \cup (FV(p) \setminus \{x\})$
$p_1 \| p_2$	$FV(p_1) \cup FV(p_2)$

E	$E\{\rho\}$
e	$e\{\rho^{\mathsf{c}}\}$
$x, x \notin \mathsf{dom}(\rho)$	x
$x, x \in \mathsf{dom}(\rho)$	$\rho(x)$
nil	nil
$p_1 \| p_2$	$p_1\{\rho\} \| p_2\{\rho\}$
$\mathsf{send}([\upsilon]E').p, \upsilon \cap FV(\rho) = \emptyset$	$\mathsf{send}([\upsilon]E'\{\rho \setminus \upsilon\}).p\{\rho\}$
$\mathsf{receive}(x[\rho']).p, x \notin FV(\rho)$	$\mathsf{receive}(x[\rho'\{\rho\}]).p\{\rho \setminus \{x\}\}$

ρ'	$\rho'\{\rho\}$
$x_i \overset{i\in I}{\mapsto} E_i$	$x_i \overset{i\in I}{\mapsto} (E_i\{\rho\})$

Fig. 2. Untyped calculus: free variables and substitution

$$(\text{CORE-SEND})\ \frac{e \xrightarrow{\mathsf{c}} e'}{\mathsf{send}([\upsilon]e).p \xrightarrow{\tau} \mathsf{send}([\upsilon]e').p} \qquad (\text{SEND})\ \frac{}{\mathsf{send}([\upsilon]E).p \xrightarrow{![\upsilon]E} p}\ FV(E) \subseteq \upsilon$$

$$(\text{CORE-RCV})\ \frac{e \xrightarrow{\mathsf{c}} e'}{\mathsf{receive}(x[\rho, y \mapsto e]).p \xrightarrow{\tau} \mathsf{receive}(x[\rho, y \mapsto e']).p}$$

$$(\text{RCV})\ \frac{}{\mathsf{receive}(x[\rho]).p \xrightarrow{?[\upsilon]E} p\{x \mapsto E\{\rho\}\}}\ \upsilon \subseteq \mathsf{dom}(\rho)$$

$$(\text{PAR-LEFT})\ \frac{p_1 \xrightarrow{\lambda} p_1'}{p_1 \| p_2 \xrightarrow{\lambda} p_1' \| p_2} \qquad (\text{PAR-RIGHT})\ \frac{p_2 \xrightarrow{\lambda} p_2'}{p_1 \| p_2 \xrightarrow{\lambda} p_1 \| p_2'} \qquad (\text{SYNC})\ \frac{p_1 \xrightarrow{\lambda} p_1' \quad p_2 \xrightarrow{\overline{\lambda}} p_2'}{p_1 \| p_2 \xrightarrow{\tau} p_1' \| p_2'}$$

Fig. 3. Untyped calculus: reduction rules

tion at the core level. Note that core code can be either sent or further reduced in a non deterministic way, and analogously for core code in a rebinding. This means that we do not care about where core mobile code is executed, either by the sender or the receiver, even though this will of course make a difference in practice, e.g., in case of non termination. Sending a process term, instead, intuitively means sending coordination code to be executed by the receiver.

In rule (SEND), mobile code can be sent only if it does not contain free variables apart from those specified by the unbinders. That is, unbinders are used by the programmer to specify whether a variable has to be bound locally or remotely, as illustrated by the second example above.

In rule (RCV), mobile code can be received only if all variables declared as free are explicitly rebound in receiver's code. That is, rebinders are used by the programmer to control which free variables in mobile code can be accepted, thus preventing accidental captures. Rules (PAR-LEFT), (PAR-RIGHT) and (SYNC), are straightforward.

The use of explicit unbinders and rebinders guarantees that exchange of open code does not introduce unbound variables (of course, provided that core reduction does neither), as stated below.

Assumption 1 (Core Free Variables) *If* $e \xrightarrow{\mathsf{c}} e'$*, then* $FV(e') \subseteq FV(e)$*.*

Proposition 1.1 (Free Variables) *Under Assumption 1:*

If $p \xrightarrow{\tau} p'$*, then* $FV(p') \subseteq FV(p)$*.*

We prove the above proposition as a case of the following, which takes into account communication steps with the outside world. Intuitively, when receiving code E, no unbound variables are introduced only if E has no more free variables than those it declares. Conversely, code sent to the external world has no more free variables than those it declares (this is inductively used to prove the property on internal steps).

Proposition 1.2 *Under Assumption 1:*

- *If* $p \xrightarrow{\tau} p'$*, then* $FV(p') \subseteq FV(p)$*.*
- *If* $p \xrightarrow{![\upsilon]E} p'$*, then* $FV(p') \subseteq FV(p)$ *and* $FV(E) \subseteq \upsilon$*.*
- *If* $p \xrightarrow{?[\upsilon]E} p'$ *and* $FV(E) \subseteq \upsilon$*, then* $FV(p') \subseteq FV(p)$*.*

Proof. *By induction on reduction rules. We show the most interesting cases:*

*(***core-send***)* *We have that* $\mathsf{send}([\upsilon]e).p \xrightarrow{\tau} \mathsf{send}([\upsilon]e').p$ *and* $e \xrightarrow{\mathsf{c}} e'$*. Hence the thesis follows by Assumption 1.*

*(***send***)* *We have that* $\mathsf{send}([\upsilon]E).p \xrightarrow{![\upsilon]E} p$*, with* $FV(E) \subseteq \upsilon$*. Hence the thesis trivially follows.*

*(***rcv***)* *We have that* $\mathsf{receive}(x[\rho]).p \xrightarrow{?[\upsilon]E} p\{x \mapsto E\{\rho\}\}$*, with* $\upsilon \subseteq \mathsf{dom}(\rho)$*. Since, by hypothesis,* $FV(E) \subseteq \upsilon$*, we have* $FV(E) \subseteq \mathsf{dom}(\rho)$*; hence,* $FV(E\{\rho\}) \subseteq FV(\rho)$ *and the thesis trivially follows.*

□

We conclude this section with two slight variants, expressed in our framework, of examples presented in [6] (Fig. 5) to show rebinding scenarios in distributed systems. We assume the core calculus to include expressions of string, unit and functional types (we write some type annotations as an help to the reader, but types are not relevant here), and we enrich the process syntax with the construct

$\mathsf{let}\,\rho\,\mathsf{in}\,p$, with the usual semantics.

Let us consider the process $\mathsf{let\,print}:\mathsf{string}\to\mathsf{unit}\mapsto\ldots\ \mathsf{in}\ (p_1\|p_2)$, where:

$$p_1 = \mathsf{let\,here}:\mathsf{string}\mapsto\text{“}\texttt{site 1}\text{”}\ \mathsf{in}$$
$$\mathsf{send}(\mathsf{print\,here}:\mathsf{unit}).\mathsf{send}([\mathsf{here}]\mathsf{print\,here}:\mathsf{unit}).\mathsf{nil}$$
$$p_2 = \mathsf{receive}(\mathsf{c}[\mathsf{here}\mapsto\text{“}\texttt{site 2}\text{”}]:\mathsf{unit}).\mathsf{send}(\mathsf{c}:\mathsf{unit}).\mathsf{nil}$$

This process reduces as follows:

$\xrightarrow{\tau}\ \mathsf{let\,print}\ldots\ \mathsf{in}\ (\mathsf{send}(\mathsf{print}\text{“}\texttt{site 1}\text{”}:\mathsf{unit}).\mathsf{send}([\mathsf{here}]\mathsf{print\,here}:\mathsf{unit}).\mathsf{nil}\|p_2)$

$\xrightarrow{!\mathsf{print}\text{“}\texttt{site 1}\text{”}:\mathsf{unit}}\ \mathsf{let\,print}\ldots\ \mathsf{in}\ (\mathsf{send}([\mathsf{here}]\mathsf{print\,here}:\mathsf{unit}).\mathsf{nil}\|p_2)$

$\xrightarrow{\tau}\mathsf{let\,print}\ldots\ \mathsf{in}\ (\mathsf{nil}\|\mathsf{send}(\mathsf{print}\text{“}\texttt{site 2}\text{”}:\mathsf{unit}).\mathsf{nil})\xrightarrow{!\mathsf{print}\text{“}\texttt{site 2}\text{”}:\mathsf{unit}}$

Hence, in the left-hand side process, variable here is first sent to be printed with its local definition, i.e. ,“site 1”, then is sent and rebound at a remote site to the label “site 2”.

Let us now consider a variant of the process above, able to perform a customized linking. This is obtained by changing the definition of p_2 in the following way:

$$p_2 = \mathsf{receive}(\mathsf{c}[\mathsf{here}\mapsto e]:\mathsf{unit}).\mathsf{send}(\mathsf{c}:\mathsf{unit}).\mathsf{nil}$$

where $e = \mathsf{if}\,\mathit{trusted}()\,\mathsf{then}\text{“}\texttt{site 2}\text{”}\,\mathsf{else}\text{“}\texttt{site 33}\text{”}$.

Here, p_2 has two possible rebindings for the variable here: the real site name “site 2” for trusted programs and the fake name “site 33” for untrusted ones. Which rebinding to perform is determined by the hypothetical function *trusted*, which takes into account some security criteria, such as the origin of the message.

It is worth to note that in our framework the rebinding is obtained without any need of a lazy semantics for the substitution, as instead happens in [6], where a delayed instantiation is required.

2 The Typed Calculus

To define the typed calculus, we need the following additional core ingredients:

- *(core) types* $t\in\mathsf{Type}^{\mathsf{c}}$,
- *(core) type judgment* $\Gamma\vDash_{\mathsf{c}} e:t$, where Γ is a *type context*, that is, a mapping from variables into (core) types, written $x_i : t_i^{i\in I}$,
- *static subtyping relation* $\vdash t'\leq_{\mathsf{s}} t$, required to be a preorder.
- *dynamic subtyping relation* $\vdash t'\leq_{\mathsf{d}} t\rightsquigarrow\mathcal{T}$, where $\mathcal{T}$ is a partial mapping, called *coercion*, $\mathcal{T}:\mathsf{Exp}^{\mathsf{c}}\to\mathsf{Exp}^{\mathsf{c}}$.

Dynamic subtyping is expected to accept more terms than static subtyping, and coercion consequently adapts the received code to the local context; indeed, mobile code exchange requires, besides dynamic checks guaranteeing type safety, also the ability of the system to dynamically modify code.

Intuitively, we expect static and dynamic subtyping to satisfy a number of properties, such as:

- if $\vdash t' \leq_{\mathsf{d}} t \rightsquigarrow \mathcal{T}$, then coercion $\mathcal{T}$ transforms expressions of (a static subtype of) type t' to expressions of (a static subtype of) type t, and is undefined on other expressions;
- in $\vdash t' \leq_{\mathsf{d}} t \rightsquigarrow \mathcal{T}$, the pair t', t uniquely determines $\mathcal{T}$,
- $\leq_{\mathsf{d}}$ is a preorder as well,
- $\vdash t \leq_{\mathsf{d}} t \rightsquigarrow \mathsf{id}$ (the identity mapping),
- if $\vdash t \leq_{\mathsf{d}} t' \rightsquigarrow \mathcal{T}, \vdash t' \leq_{\mathsf{d}} t'' \rightsquigarrow \mathcal{T}', \vdash t' \leq_{\mathsf{d}} t'' \rightsquigarrow = \mathcal{T}' \circ \mathcal{T}$ [7]
- $\leq_{\mathsf{s}}$ is a subset of $\leq_{\mathsf{d}}$, and $\vdash t \leq_{\mathsf{d}} t' \rightsquigarrow \mathsf{id}$ whenever $\vdash t \leq_{\mathsf{s}} t'$.

However, we do not formally assume here any of the above properties, since they are not necessary for our main result, that is, type safety (Theorem 2.4), which can be proved under somewhat weaker assumptions, see Assumption 3 and Assumption 5. We leave to further work the investigation of other significant requirements the framework should satisfy which will likely explicitly require some, if not all, of the assumptions as above.

As mentioned in the Introduction, coercions driven by a subtyping relation are also used, mainly with performance reasons, for compiling source code with subtyping in lower-level code without subtyping, see Sect.15.6 of [13]. In this context, the translation which inserts coercions is often called the *Penn translation* [7]. Apart from the different context and aims, our presentation here differs for some other reasons.

First, our technical treatment is lighter, since, following the style of recent work where type-checking is generalized to compilation, as, e.g., [1], we pack relation between types and coercion in a unique "compilation" judgment, which we expect to be inductively defined in instantiations of the framework, as, for instance, we do in Sect.3. In [13], on the contrary, the translation is modeled as a function which takes *derivations* of subtyping judgments as arguments. Another drastic simplification is that we need to insert coercions only in a single situation, that is, when receiving code, whereas in the original Penn translation coercions must be inserted in a term everywhere there is a subterm of a certain type which appears in a context of a supertype. The technical counterpart of this simplification is that our coercion function can take just terms as arguments, instead of requiring to keep the typing judgment of the term as in [7].

Second, and more interestingly, since we handle open terms, subtyping is naturally extended to contexts and coercions are inserted in both directions. We believe this is a nice and important generalization of the coercions-driven-by-subtyping approach.

Finally, whereas the original approach is purely syntactic, that is, coercions are expressed as terms of the lower-level language (e.g., λ-abstractions), here, since our aim is to define an abstract framework where core language is not fixed, we take an extensional approach, where coercions are modeled as functions from terms into

[7] These properties altogether amount to say that there is a functor from the category which has types as objects and $\leq_{\mathsf{d}}$ as arrows to the subcategory of *Set* which has as objects the sets of expressions of (a static subtype of) a certain type.

$$
\begin{array}{lll}
p \in \mathsf{Proc} & ::= x \mid \mathsf{nil} \mid p_1 \| p_2 \mid \mathsf{send}(\Gamma \vdash [\upsilon]E : T).p \mid & \textbf{process} \\
 & \quad \mathsf{receive}(\Gamma \vdash x[\rho] : T).p & \\
E \in \mathsf{Exp} & ::= e \mid p & \textbf{mobile code} \\
T \in \mathsf{Type} & ::= t \mid \diamond & \textbf{type} \\
\Gamma & ::= x_i{:}T_i^{\,i \in I} & \textbf{type context} \\
\lambda & ::= \tau \mid !\Gamma \vdash [\upsilon]E : T \mid ?\Gamma \vdash [\upsilon]E : T & \textbf{label} \\
\overline{!\Gamma \vdash [\upsilon]E : T} & = \quad ?\Gamma \vdash [\upsilon]E : T & \textbf{complement} \\
\overline{?\Gamma \vdash [\upsilon]E : T} & = \quad !\Gamma \vdash [\upsilon]E : T &
\end{array}
$$

Fig. 4. Typed calculus: syntax

$$
\frac{}{\vdash \diamond \leq_{\mathsf{s}} \diamond} \qquad \frac{\vdash T_i \leq_{\mathsf{s}} T_i',\ i \in I'}{\vdash x_i{:}T_i^{\,i \in I} \leq_{\mathsf{s}} x_i{:}T_i'^{\,i \in I'}}\, I' \subseteq I \text{ (implicit)}
$$

$$
\frac{}{\vdash \diamond \leq_{\mathsf{d}} \diamond \rightsquigarrow \mathcal{T}}\, \mathcal{T}(p) = p \qquad \frac{\vdash T_i \leq_{\mathsf{d}} T_i' \rightsquigarrow \mathcal{T}_i,\ i \in I'}{\vdash x_i{:}T_i^{\,i \in I} \leq_{\mathsf{d}} x_i{:}T_i'^{\,i \in I'} \rightsquigarrow \mathcal{T}}\, \mathcal{T}(x_i \overset{i \in I}{\mapsto} E_i) = x_i \overset{i \in I'}{\mapsto} \mathcal{T}_i(E_i)
$$

Fig. 5. Typed calculus: subtyping

terms. The fact that coercions could be internalized in the language or not will then depend on the specific instantiation of the framework: for instance, in the following section we will present an instantiation on a simple λ-calculus with records where coercions are expressed by λ-abstractions as in the original approach.

The syntax of the typed calculus is in Fig.4. The main novelty w.r.t. the untyped version is that mobile code is annotated with a type context Γ (mapping variables into types) and a type T. Types are either core types or the *process type* $\diamond$. As well-formedness condition, in send and labels we assume $\upsilon = \mathsf{dom}(\Gamma)$, and in $\mathsf{receive}$ we assume $\mathsf{dom}(\rho) = \mathsf{dom}(\Gamma)$. Hence, υ is redundant, but we keep it for uniformity with the untyped version.

We will use the following additional notations for mappings (e.g., type contexts): $\mathsf{dom}(\Gamma)$ is the domain of Γ; $\Gamma[\Gamma']$ is the mapping obtained by updating Γ with the associations in Γ'.

In Fig.5, we extend subtyping relations to the process type and to type contexts. The process type is in relation only with itself and the corresponding coercion is the identity. Subtyping relations on type contexts are defined in the natural pointwise way and the associated coercion transforms substitutions of the subtype context into substitutions of the supertype context (substitutions have contexts as types, see rule (T-SUBST) in Fig.7).

Reduction rules for the extended calculus are in Fig.6. They are a straightforward extension to annotated mobile code of those seen for the untyped calculus,

$$(\textsc{core-send})\ \frac{e \xrightarrow{\mathsf{c}} e'}{\mathsf{send}(\Gamma \vdash [\upsilon]\, e : t\,).p \xrightarrow{\tau} \mathsf{send}(\Gamma \vdash [\upsilon]\, e' : t\,).p}$$

$$(\textsc{send})\ \frac{}{\mathsf{send}(\Gamma \vdash [\upsilon] E : T\,).p \xrightarrow{!\Gamma \vdash [\upsilon] E : T} p}\ FV(E) \subseteq \mathsf{dom}(\Gamma)$$

$$(\textsc{core-rcv})\ \frac{e \xrightarrow{\mathsf{c}} e'}{\mathsf{receive}(\Gamma \vdash x[\rho, y \mapsto e] : T\,).p \xrightarrow{\tau} \mathsf{receive}(\Gamma \vdash x[\rho, y \mapsto e'] : T\,).p}$$

$$(\textsc{rcv})\ \frac{}{\mathsf{receive}(\Gamma \vdash x[\rho] : T\,).p \xrightarrow{?\Gamma' \vdash [\upsilon] E : T'} p\{x \mapsto \mathcal{T}'\ (E\{\mathcal{T}(\rho)\})\}}\ \begin{array}{l} \vdash T' \leq_{\mathsf{d}} T \rightsquigarrow \mathcal{T}' \\ \vdash \Gamma \leq_{\mathsf{d}} \Gamma' \rightsquigarrow \mathcal{T} \end{array}$$

$$(\textsc{par-left})\ \frac{p_1 \xrightarrow{\lambda} p_1'}{p_1 \| p_2 \xrightarrow{\lambda} p_1' \| p_2} \qquad (\textsc{par-right})\ \frac{p_2 \xrightarrow{\lambda} p_2'}{p_1 \| p_2 \xrightarrow{\lambda} p_1 \| p_2'} \qquad (\textsc{sync})\ \frac{p_1 \xrightarrow{\lambda} p_1' \quad p_2 \xrightarrow{\overline{\lambda}} p_2'}{p_1 \| p_2 \xrightarrow{\tau} p_1' \| p_2'}$$

Fig. 6. Typed calculus: reduction rules

except for (RCV), which is the key rule illustrating our approach. The side condition expresses the fact that incoming code E can be retrieved only if its type information Γ', T' is compliant with that specified by the receiver Γ, T, as formally expressed by the subtyping relation. In this case, appropriate coercions are inserted before combining E with local code, to bridge the gap between provided and required type information. [8]

More precisely, all variables explicitly declared as free in the incoming code are rebound to local definitions via coercion from the provided type context Γ to the expected type context Γ'; then, the resulting (now closed since $FV(E) \subseteq \upsilon = \mathsf{dom}(\Gamma')$ and $\mathsf{dom}(\Gamma') \subseteq \mathsf{dom}(\Gamma)$) expression is substituted in local code via coercion from the declared type T' to the required type T.

Typing rules, given in Fig.7, are straightforward. In rule (T-CORE), we denote by Γ^{c} the subset of a context Γ which maps core variables into core types. Rule (T-SEND) allows sending of code which has a static subtype of that it declares, and conversely rule (T-RCV) allows the rebinding to have a static subtype of that declared. Recall also that by well-formedness conditions we have $\mathsf{dom}(\Gamma_2) = \upsilon$ in rule (T-SEND) and $\mathsf{dom}(\Gamma_2) = \mathsf{dom}(\rho)$ in rule (T-RCV).

We illustrate now how dynamic subtyping and coercion work by an example, where we consider the instantiation of the framework which will be formally detailed in the following section. That is, we assume that expressions of the core calculus include numbers and records with a sum (concatenation) operator denoted by $+$ and standard record types. Consider the process:

$$\mathsf{receive}(y : \mathsf{posint} \vdash x[y \mapsto 1] : \{X : \mathsf{int}, Y : \mathsf{int}\}\,).\mathsf{send}(x + \{Z : 3\}).\mathsf{nil}$$

[8] For simplicity, here communicating something of a wrong type corresponds to no reduction at all; a more realistic model should include reduction into a distinguished *error* term of either the receiver only or the communicating pair.

$$(\text{T-SUBST})\ \frac{\Gamma \vdash E_i : T_i, i \in I}{\Gamma \vdash x_i \overset{i\in I}{\mapsto} E_i : x_i{:}T_i^{\,i\in I}} \qquad (\text{T-CORE})\ \frac{\Gamma^{\mathsf{c}} \vdash_{\mathsf{c}} e : t}{\Gamma \vdash e : t}$$

$$(\text{T-VAR-PROC})\ \frac{}{\Gamma \vdash x : \diamond}\ \Gamma(x) = \diamond \qquad (\text{T-NIL})\ \frac{}{\Gamma \vdash \mathsf{nil} : \diamond}$$

$$(\text{T-PAR})\ \frac{\Gamma \vdash p_1 : \diamond \quad \Gamma \vdash p_2 : \diamond}{\Gamma \vdash p_1 \| p_2 : \diamond} \qquad (\text{T-SEND})\ \frac{\Gamma_1[\Gamma_2] \vdash E : T' \quad \Gamma_1 \vdash p : \diamond \quad \vdash T' \leq_{\mathsf{s}} T}{\Gamma_1 \vdash \mathsf{send}(\Gamma_2 \vdash [v]E : T\,).p : \diamond}$$

$$(\text{T-RCV})\ \frac{\Gamma_1[x{:}T] \vdash p : \diamond \quad \Gamma_1 \vdash \rho : \Gamma \quad \vdash \Gamma \leq_{\mathsf{s}} \Gamma_2}{\Gamma_1 \vdash \mathsf{receive}(\Gamma_2 \vdash x[\rho] : T\,).p : \diamond}$$

Fig. 7. Typed calculus: typing rules

and assume that code $?y : \mathsf{int} \vdash [y]\{X : 0, Y : y, Z : 2\} : \{X : \mathsf{int}, Y : \mathsf{int}, Z : \mathsf{int}\}$ is received.

We ensure type safe exchange of mobile code by a runtime check analogous to that considered in [4] for mixin classes, to solve the classical problem of interference in record/object types. That is, dynamic subtyping corresponds to standard width subtyping on record types, together with a coercion function which removes additional fields [9]. Then, the type declared by mobile code is a subtype of the expected type, hence communication can take place. Mobile code is adapted to the local code by the following steps. First, y is replaced in the received code via coercion from posint to int, which is the identity, obtaining $\{X : 0, Y : 1, Z : 2\}$. Then, x is replaced in the local code via coercion from $\{X : \mathsf{int}, Y : \mathsf{int}, Z : \mathsf{int}\}$ to $\{X : \mathsf{int}, Y : \mathsf{int}\}$, obtaining a safe record extension in $\mathsf{send}(\{X : 0, Y : 1\} + \{Z : 3\}).\mathsf{nil}$.

The combination of the static type system and the dynamic checks should ensure *type safety*, that is, that internal steps can never lead to ill-formed process terms (for steps of communication with the "external world" this requires to be confident on the fact that received code complies with its accompanying type information, see below). [10]

Definition 2.1 (Type Safety) Exchange of mobile code is *type safe* if the following (SR) property holds:

If $\Gamma \vdash p : \diamond$ and $p \xrightarrow{\tau} p'$, then $\Gamma \vdash p' : \diamond$.

We list now a number of assumptions the core calculus should satisfy in order to have type safety. They are mostly standard properties, plus Assumption 5, which states that, whenever the dynamic check on core mobile code succeeds (that is, its declared type is in the dynamic subtyping relation with the required type), this code

[9] If objects rather than (non recursive) records are considered, additional fields must be *frozen* rather than just removed, see [8] for details.

[10] Note that in distributed scenarios type safety, usually expressed by *subject reduction* (SR) and *progress* properties [9], reduces to SR (as in, e.g., [14,10]), since ensuring progress would require a sophisticated static analysis (*deadlock detection*).

can be safely incorporated with local code via the corresponding coercion function.

Assumption 2 *If* $\Gamma \vDash_{\mathsf{c}} e : t$, $x \notin \mathsf{dom}(\Gamma)$, *then* $e\{x \mapsto e'\} = e$.

Assumption 3 (Core Weakening) *If* $\Gamma \vDash_{\mathsf{c}} e : t$ *and* $\vdash \Gamma' \leq_{\mathsf{s}} \Gamma$, *then* $\Gamma' \vDash_{\mathsf{c}} e : t'$, *with* $\vdash t' \leq_{\mathsf{s}} t$. *Moreover, if* $FV(e) \cap \mathsf{dom}(\Gamma') = \emptyset$, *then* $\Gamma[\Gamma'] \vdash e : t$.

Assumption 4 (Core SR) *If* $\Gamma \vDash_{\mathsf{c}} e : t$ *and* $e \xrightarrow{\mathsf{c}} e'$, *then* $\Gamma \vDash_{\mathsf{c}} e' : t'$ *for some* $\vdash t' \leq_{\mathsf{s}} t$.

Assumption 5 (Core Coercion Substitution) *If* $\Gamma[x{:}t_x] \vDash_{\mathsf{c}} e : t$, $\Gamma \vDash_{\mathsf{c}} e' : t''_x$, $\vdash t''_x \leq_{\mathsf{s}} t'_x$, *and* $\vdash t'_x \leq_{\mathsf{d}} t_x \rightsquigarrow \mathcal{T}$, *then* $\Gamma \vDash_{\mathsf{c}} e\{x \mapsto \mathcal{T}(e')\} : t'$, *for some* $\vdash t' \leq_{\mathsf{s}} t$.

Here t_x is the required type, t'_x the type declared by the mobile code and t''_x its actual type.

We now give some useful lemmas.

Lemma 2.2 (Weakening) *If Assumption 3 holds, then if* $\Gamma \vdash E : T$ *and* $\vdash \Gamma' \leq_{\mathsf{s}} \Gamma$, *then* $\Gamma' \vdash E : T'$, *with* $\vdash T' \leq_{\mathsf{s}} T$; *moreover, if* $FV(E) \cap \mathsf{dom}(\Gamma') = \emptyset$, *then* $\Gamma[\Gamma'] \vdash E : T$.

Lemma 2.3 (Coercion Substitution) *Under assumption 5, if* $\Gamma[x{:}T_x] \vdash E : T$, $\Gamma \vdash E' : T''_x$, $\vdash T''_x \leq_{\mathsf{s}} T'_x$, *and* $\vdash T'_x \leq_{\mathsf{d}} T_x \rightsquigarrow \mathcal{T}$, *then* $\Gamma \vdash E\{x \mapsto \mathcal{T}(E')\} : T'$, *for some* $\vdash T' \leq_{\mathsf{s}} T$.

Proof. By induction on typing rules. We show the most interesting cases.

(t-var-proc) We have that $\Gamma[x{:}T_x] \vdash y : \diamond$ and $(\Gamma[x{:}T_x])\,(y) = \diamond$, and thus either $x = y$, hence $T_x = T'_x = T''_x = \diamond$, E' is a process p', $\vdash \diamond \leq_{\mathsf{d}} \diamond \rightsquigarrow \mathsf{id}$, $y\{y \mapsto \mathsf{id}(p')\} = p'$ and $\Gamma \vdash p' : \diamond$ holds by hypothesis, or $x \neq y$, and thus $y\{x \mapsto \mathcal{T}(E')\} = y$, $\Gamma(y) = \diamond$, and $\Gamma \vdash y : \diamond$ holds by applying typing rule (T-VAR-PROC).

(t-nil) Trivial.

(t-send) We have that $\Gamma[x{:}T_x] \vdash \mathsf{send}(\Gamma_2 \vdash [\upsilon]E : T\,).p : \diamond$ (1), and $\Gamma[x{:}T_x][\Gamma_2] \vdash E : T'$, (2) $\vdash T' \leq_{\mathsf{s}} T$, $\mathsf{dom}(\Gamma_2) = \upsilon$ and $\Gamma[x{:}T_x] \vdash p : \diamond$ (3). By applying the inductive hypothesis to (3), we get $\Gamma \vdash p\{x \mapsto \mathcal{T}(E')\} : \diamond$ (4). There are two cases to be considered. If $x \in \mathsf{dom}(\Gamma_2)$, we can conclude by applying the typing rule (T-SEND) to (1) and (4). Otherwise, for definition of substitution, $\mathsf{dom}(\Gamma_2) \cap FV(E') = \emptyset$, hence, by applying Lemma 2.2 to the hypothesis $\Gamma \vdash E' : T''_x$, we get $\Gamma[\Gamma_2] \vdash E' : T''_x$. We can now apply the inductive hypothesis to (2) obtaining $\Gamma[\Gamma_2] \vdash E\{x \mapsto \mathcal{T}(E')\} : T''$ (5), for some $\vdash T'' \leq_{\mathsf{s}} T'$. Then, since $\leq_{\mathsf{s}}$ is a preorder, we have $\vdash T'' \leq_{\mathsf{s}} T$ and we get the thesis by applying typing rule (T-SEND) to (4) and (5).

(t-core) We have that $\Gamma[x{:}T_x] \vdash e : t$. Moreover, if $T_x = \diamond$ then $\Gamma^{\mathsf{core}} \vDash_{\mathsf{c}} e : t$, hence, by Assumption 2, we have $e\{x \mapsto \mathcal{T}(E')\} = e$ and the thesis follows by applying rule (T-CORE). Otherwise, T_x is a core type t_x, hence $\Gamma^{\mathsf{core}}[x{:}t_x] \vDash_{\mathsf{c}} e : t$. Then, T'_x must be a core type t'_x as well, $\vdash t'_x \leq_{\mathsf{d}} t_x \rightsquigarrow \mathcal{T}$ and E' a core expression e', and by Assumption 5 we get $\Gamma^{\mathsf{core}} \vDash_{\mathsf{c}} e\{x \mapsto \mathcal{T}(e')\} : t'$, for some $\vdash t' \leq_{\mathsf{s}} t$. Hence, we get the thesis by applying typing rule (T-CORE).

(t-rcv) We have that $\Gamma[x{:}T_x] \vdash \mathsf{receive}(\Gamma_2 \vdash y[\rho] : T\,).p : \diamond$ (1), and $\Gamma[x{:}T_x][y{:}T] \vdash p : \diamond$

(2), $\Gamma[x{:}T_x] \vdash \rho : \Gamma$, and $\vdash \Gamma \leq_{\mathsf{s}} \Gamma_2$ (3). We apply the inductive hypothesis to all $y \in \mathsf{dom}(\rho)$ in (3) obtaining $\Gamma \vdash \rho\{x \mapsto \mathcal{T}(E')\} : \Gamma'$ (4) for some $\vdash \Gamma' \leq_{\mathsf{s}} \Gamma$. Hence, since $\leq_{\mathsf{s}}$ is a preorder, $\vdash \Gamma' \leq_{\mathsf{s}} \Gamma_2$. There are two cases to be considered. If $x = y$, then the thesis follows by applying typing rule (T-RCV) to (2) and (4). Otherwise, for definition of substitution we know that $y \notin FV(E')$, hence, by applying Lemma 2.2 to the hypothesis $\Gamma \vdash E' : T''_x$, we get $\Gamma[y{:}T] \vdash E' : T''_x$ (5). We can now apply the inductive hypothesis to (2) and (5) obtaining $\Gamma[y{:}T] \vdash p\{x \mapsto \mathcal{T}(E')\} : \diamond$ (6), and conclude by applying typing rule (T-RCV) to (4) and (6).

(t-par) Trivially by inductive hypothesis. □

Theorem 2.4 *If assumption 5 holds, then exchange of mobile code is type safe.*

We prove type safety as a case of the following *generalized* type safety which takes into account communication steps with the outside world. Intuitively, when receiving code E, safety is guaranteed only if E actually complies its accompanying type information Γ, T. We assume here to trust this type information to be correct: a more sophisticated approach would require a *proof*, as in [12]. Conversely, we can prove that code sent to the external world always complies the declared type information (this is inductively used to prove safety of internal steps).

Proposition 2.5 *Under assumption 5:*

- *If* $\Gamma \vdash p : \diamond$ *and* $p \xrightarrow{\tau} p'$, *then* $\Gamma \vdash p' : \diamond$.
- *If* $\Gamma_1 \vdash p : \diamond$ *and* $p \xrightarrow{!\Gamma_2 \vdash [v]E:T} p'$, *then* $\Gamma_1 \vdash p' : \diamond$, $\Gamma_1[\Gamma_2] \vdash E : T'$, *for some* $\vdash T' \leq_{\mathsf{s}} T$.
- *If* $\Gamma_1 \vdash p : \diamond$, $p \xrightarrow{?\Gamma_2 \vdash [v]E:T} p'$, *and* $\Gamma_1[\Gamma_2] \vdash E : T'$, *with* $\vdash T' \leq_{\mathsf{s}} T$, *then* $\Gamma_1 \vdash p' : \diamond$.

Proof. By induction on reduction rules. We show the most interesting cases.

(core) We have that $\mathsf{send}(\Gamma_2 \vdash [v]e : t).p \xrightarrow{\tau} \mathsf{send}(\Gamma_2 \vdash [v]e' : t).p$, $e \xrightarrow{\mathsf{c}} e'$, and, since we must have applied typing rules (T-SEND) and (T-CORE), $\Gamma_1 \vdash \mathsf{send}(\Gamma_2 \vdash [v]e : t).p : \diamond$, $(\Gamma_1[\Gamma_2])^{\mathsf{core}} \vdash_{\mathsf{c}} e : t'$, $\vdash t' \leq_{\mathsf{s}} t$, $\mathsf{dom}(\Gamma_2) = v$ and $\Gamma_1 \vdash p : \diamond$. Since SR holds for the core calculus (Assumption 4), we get that $(\Gamma_1[\Gamma_2])^{\mathsf{core}} \vdash_{\mathsf{c}} e' : t''$, with $\vdash t'' \leq_{\mathsf{s}} t'$, and, since $\leq_{\mathsf{s}}$ is a preorder, $\vdash t'' \leq_{\mathsf{s}} t$. Hence by applying typing rules (T-CORE) and (T-SEND) the thesis follows.

(send) We have that $\mathsf{send}(\Gamma_2 \vdash [v]E : T).p \xrightarrow{!\Gamma_2 \vdash [v]E:T} p$, with $FV(E) \subseteq \mathsf{dom}(\Gamma_2)$; moreover, we have $\Gamma_1 \vdash \mathsf{send}(\Gamma_2 \vdash [v]E : T).p : \diamond$. To derive this last judgment, we must have applied typing rule (T-SEND), hence $\Gamma_1 \vdash p : \diamond$ and $\Gamma_1[\Gamma_2] \vdash E : T'$, with $\vdash T' \leq_{\mathsf{s}} T$.

(rcv) We have that

$$\mathsf{receive}(\Gamma_2 \vdash x[\rho] : T).p \xrightarrow{?\Gamma'_2 \vdash [v]E:T'} p\{x \mapsto \mathcal{T}\left(E\{\mathcal{T}'(\rho)\}\right)\}$$

with $\vdash T' \leq_{\mathsf{d}} T \rightsquigarrow \mathcal{T}$ and $\vdash \Gamma_2 \leq_{\mathsf{d}} \Gamma'_2 \rightsquigarrow \mathcal{T}'$ (1); moreover, we have $\Gamma_1 \vdash \mathsf{receive}(\Gamma_2 \vdash x[\rho] : T).p : \diamond$ (2) and $\Gamma_1[\Gamma'_2] \vdash E : T''$ (3), with $\vdash T'' \leq_{\mathsf{s}} T'$ (4). To derive (2), we must have applied typing rule (T-RCV), hence $\Gamma_1[x{:}T] \vdash p : \diamond$ (5), $\Gamma_1 \vdash \rho : \Gamma$,

$\vdash \Gamma \leq_{\mathsf{s}} \Gamma_2$ (6) and $\mathsf{dom}(\rho) = \mathsf{dom}(\Gamma_2)$ (7). We can apply Lemma 2.3 to (3) and all y in (6) (note that $\mathsf{dom}(\rho) = \mathsf{dom}(\Gamma_2) \subseteq \mathsf{dom}(\Gamma_2')$ from (1) and (7)), with $\vdash \Gamma_2(y) \leq_{\mathsf{d}} \Gamma_2'(y)$ (from (1)), obtaining $\Gamma_1 \vdash E\{\mathcal{T}'(\rho)\} : T'''$ (8), with $\vdash T''' \leq_{\mathsf{s}} T''$ (9). Since $\leq_{\mathsf{s}}$ is a preorder, from (9) and (4) we get $\vdash T''' \leq_{\mathsf{s}} T'$ (10). We can now conclude by applying Lemma 2.3 to (5) and (8), with (1) and (10).

□

3 A case study: lambda calculus with records

A case-study in exchange of mobile code which has been extensively studied [4,3,2,8] is when code to be exchanged has a record-based structure (records, objects, classes, mixins), and type safety is made problematic by conflicts due to components which were not explicitly required. For instance, in MoMi [4,3,2] mobile code consists in mixin classes, and conflicts are avoided by a renaming mechanism which, essentially, hides unexpected components to receiver's code. In [8], we have formalized this kind of solution (on mixin modules rather than classes) as one instantiation of our parametric framework for type safe exchange of mobile code.

However, in this previous work only top-level conflicts were detected and avoided, whereas at nested levels width subtyping was simply not allowed. For instance, given as expected type $\{X:\{Y:\mathsf{int}\}\}$, the type $\{X:\{Y:\mathsf{int}\},Z:\mathsf{int}\}$ was accepted (and Z removed), while $\{X:\{Y:\mathsf{int},Z:\mathsf{int}\}\}$ was rejected.

In this section, we show that a runtime check based on the Penn translation found in the literature allows for simple and nice detection and elimination of conflicts due to arbitrarily nested components. For simplicity, we illustrate the approach on the more foundational example of records, but the same technique could be easily adapted to objects, classes or mixins: in these cases, to take into account mutual recursion, additional fields must be hidden rather than just removed, see [8] for details.

Formally, we present an instantiation of the framework introduced in the previous sections which takes as core calculus a simple λ-calculus with records, as static subtyping depth subtyping, and as dynamic subtyping depth/width subtyping with a coercion function which removes additional fields. We call the instantiation $\mathsf{MoRec}^{\mathsf{del}}$ (for "MObile RECords where unexpected fields are DELeted").

The syntax of the core calculus is given in Fig.8. We assume, besides variables, an infinite set Field of *field names*. Terms of the calculus are built by (unspecified) operators of basic types, standard operators of lambda calculus, and *records* with three operators: *sum*, *delete* and *selection*. A record is a map from field names to expressions.

The reduction relation is given in Fig.9, where we omit standard contextual closure.

Reduction rules are straightforward: rule (APP) is standard application (we are not interested in fixing an evaluation strategy here), rule (SEL) allows selection of an existing field, rule (SUM) performs the union of two records if their sets of field names are disjoint, rule (DEL) removes a field from a record (if present).

$X, Y, Z, \ldots \in \mathsf{Field}$			**field name**
$x, y, z, \ldots \in \mathsf{Var}$			**variable**
$e \in \mathsf{Exp}^{\mathsf{c}}$	$::=$		**expression**
	$\ldots$		basic operators
	$\mid$	$x \mid \lambda x.\, e \mid e_1\, e_2$	lambda calculus operators
	$\mid$	$\{fs\}$	record
	$\mid$	$e_1 + e_2$	sum
	$\mid$	$e \setminus X$	delete
	$\mid$	$e.X$	selection
fs	$:=$	$X_i \overset{i \in I}{\mapsto} e_i$	**fields**

Fig. 8. $\mathsf{MoRec}^{\mathsf{del}}$: syntax

$$(\text{APP})\ \frac{}{(\lambda x.\, e_1)\, e_2 \xrightarrow{\mathsf{c}} e_1\{x \mapsto e_2\}} \qquad (\text{SEL})\ \frac{}{\{fs\}\,.X \xrightarrow{\mathsf{c}} fs(X)}$$

$$(\text{SUM})\ \frac{}{\{fs_1\} + \{fs_2\} \xrightarrow{\mathsf{c}} \{fs_1, fs_2\}}\ \mathsf{dom}(fs_1) \cap \mathsf{dom}(fs_2) = \emptyset \qquad (\text{DEL})\ \frac{}{\{fs\} \setminus X \xrightarrow{\mathsf{c}} \{fs \setminus X\}}$$

Fig. 9. $\mathsf{MoRec}^{\mathsf{del}}$: reduction rules

The λ-calculus with records, with all required ingredients (variables, expressions, substitution application and reduction relation) can be used as a core calculus for the untyped parametric coordination calculus illustrated in Sect.1, since it satisfies the required assumption.

Theorem 3.1 *Assumption 1 of Sect.1 is satisfied, that is:*

If $e \xrightarrow{\mathsf{c}} e'$, *then* $FV(e') \subseteq FV(e)$.

Proof. By induction on reduction rules. □

We give now the static type system and the runtime check for $\mathsf{MoRec}^{\mathsf{del}}$. We assume that the syntax of Fig.8 is enriched with a type annotation for the lambda abstraction binder, as usual in the typed λ-calculus.

Typing rules are in Fig.10. Types include (unspecified) basic types and functional and record types. A record type consists of a *signature* Σ which is a map from field names into types.

In Fig.11 we define the subtyping relations. It is worth to note that, analogously to what happens in [13], in $\mathsf{MoRec}^{\mathsf{del}}$ coercion can be internalized, hence we consider dynamic subtyping judgments having form $\vdash t' \leq_{\mathsf{d}} t \rightsquigarrow f$, with $f \in \mathsf{Exp}^{\mathsf{c}}$ (we use a different metavariable to stress that f will be an expression of a functional type). Both static and dynamic subtyping are the usual subtyping on functional types (that is, contravariant in the input and covariant in the output) and both allow

$$
\begin{array}{rcll}
t \in \mathsf{Type}^{\mathsf{c}} & ::= & & \textbf{type} \\
& & \ldots & \text{basic types} \\
& | & t_1 \to t_2 & \text{functional type} \\
& | & \{\Sigma\} & \text{record type} \\
\Sigma & := & X_i : t_i{}^{i\in I} & \textbf{signature}
\end{array}
$$

$$
(\text{T-VAR})\ \frac{}{\Gamma \vdash_{\mathsf{c}} x : \Gamma(x)} \qquad (\text{T-LAMBDA})\ \frac{\Gamma[x{:}t_1] \vdash_{\mathsf{c}} e : t_2}{\Gamma \vdash_{\mathsf{c}} \lambda x{:}t_1.\, e : t_1 \to t_2} \qquad (\text{T-APP})\ \frac{\Gamma \vdash_{\mathsf{c}} e_1 : t_2 \to t \qquad \Gamma \vdash_{\mathsf{c}} e_2 : t_2' \qquad \vdash t_2' \leq_{\mathsf{s}} t_2}{\Gamma \vdash_{\mathsf{c}} e_1\, e_2 : t}
$$

$$
(\text{T-RECORD})\ \frac{\Gamma \vdash_{\mathsf{c}} e_i : t_i,\ i \in I}{\Gamma \vdash_{\mathsf{c}} \left\{ X_i \overset{i\in I}{\mapsto} e_i \right\} : \left\{ X_i : t_i{}^{i\in I} \right\}}
$$

$$
(\text{T-SUM})\ \frac{\Gamma \vdash_{\mathsf{c}} e_i : \{\Sigma_i\},\ i \in 1,2}{\Gamma \vdash_{\mathsf{c}} e_1 + e_2 : \{\Sigma_1, \Sigma_2\}}\ \mathsf{dom}(\Sigma_1) \cap \mathsf{dom}(\Sigma_2) = \emptyset
$$

$$
(\text{T-DEL})\ \frac{\Gamma \vdash_{\mathsf{c}} e : \{\Sigma\}}{\Gamma \vdash_{\mathsf{c}} e \setminus X : \{\Sigma \setminus X\}} \qquad (\text{T-SEL})\ \frac{\Gamma \vdash_{\mathsf{c}} e : \{\Sigma\}}{\Gamma \vdash_{\mathsf{c}} e.X : \Sigma(X)}
$$

Fig. 10. $\mathsf{MoRec}^{\mathsf{del}}$: type system

basic subtyping rules ...

$$
\frac{\vdash t_1' \leq_{\mathsf{s}} t_1 \quad \vdash t_2 \leq_{\mathsf{s}} t_2'}{\vdash t_1 \to t_2 \leq_{\mathsf{s}} t_1' \to t_2'} \qquad \frac{\vdash t_i \leq_{\mathsf{s}} t_i',\ i \in I}{\vdash \left\{ X_i : t_i{}^{i\in I} \right\} \leq_{\mathsf{s}} \left\{ X_i : t_i'{}^{i\in I} \right\}}
$$

$$
\frac{\vdash t_1' \leq_{\mathsf{d}} t_1 \rightsquigarrow f_1 \qquad \vdash t_2 \leq_{\mathsf{d}} t_2' \rightsquigarrow f_2}{\vdash t_1 \to t_2 \leq_{\mathsf{d}} t_1' \to t_2' \rightsquigarrow \lambda y{:}t_1 \to t_2.\, \lambda x{:}t_1'.\, f_2(y(f_1 x))}
$$

$$
\frac{\vdash t_i \leq_{\mathsf{d}} t_i' \rightsquigarrow f_i, i \in I'}{\vdash \left\{ X_i : t_i{}^{i\in I} \right\} \leq_{\mathsf{d}} \left\{ X_i : t_i'{}^{i\in I'} \right\} \rightsquigarrow \lambda y{:}\left\{ X_i : t_i{}^{i\in I} \right\}.\, \left\{ X_i \overset{i\in I'}{\mapsto} f_i(y.X_i) \right\}}
$$

Fig. 11. $\mathsf{MoRec}^{\mathsf{del}}$: subtyping relations

depth subtyping on record types. Moreover, dynamic subtyping also allows width subtyping on record types. For instance, assuming $\vdash \mathsf{posint} \leq_{\mathsf{d}} \mathsf{int} \rightsquigarrow \lambda z{:}\mathsf{posint}.\, z$, if the expected type is $\{X{:}\{Y{:}\mathsf{int}\}\}$, then $\{X{:}\{Y{:}\mathsf{posint}, Z{:}\mathsf{int}\}, W{:}\mathsf{int}\}$ is accepted and the corresponding coercion is represented by the expression

$$
\lambda x{:}\{X{:}\{Y{:}\mathsf{posint}, Z{:}\mathsf{int}\}, W{:}\mathsf{int}\}.\, \{X \mapsto (\lambda y{:}\{Y{:}\mathsf{posint}, Z{:}\mathsf{int}\}.\, \{Y \mapsto (\lambda z{:}\mathsf{posint}.\, z)\,(y.Y)\}\,)\,(x.X)\}.
$$

Note that, as already mentioned, coercion hierarchically deletes all unexpected fields.

We can now show that $\mathsf{MoRec}^{\mathsf{del}}$, with all required ingredients (types, type judgment, static and dynamic subtyping relations), can be used as a parameter for the typed parametric coordination framework illustrated in Sect.2, since it satisfies all required assumptions.

We first give some useful lemmas.

Lemma 3.2 (Subst) *If* $\Gamma[x{:}t_2] \models_{\mathsf{c}} e_1 : t_1$, $\Gamma \models_{\mathsf{c}} e_2 : t_2$, *then* $\Gamma \models_{\mathsf{c}} e_1\{x \mapsto e_2\} : t_1$. *Moreover, if* $\Gamma[x{:}t_2] \models_{\mathsf{c}} e_1 : t_1$, $\Gamma \vdash e_2 : t_2'$, *with* $t_2' \leq_{\mathsf{s}} t_2$, *then* $\Gamma \vdash e_1\{x \mapsto e_2\} : t_1'$, *with* $t_1' \leq_{\mathsf{s}} t_1$.

Proof. The first part of the lemma is proved by induction on the structure of e_1. For the moreover part, we observe that if $\Gamma[x{:}t_2] \models_{\mathsf{c}} e_1 : t_1$, $\Gamma \vdash e_2 : t_2'$, with $t_2' \leq_{\mathsf{s}} t_2$, then, for the weakening property (see point A3 below), $\Gamma[x{:}t_2'] \models_{\mathsf{c}} e_1 : t_1'$, with $t_1' \leq_{\mathsf{s}} t_1$, and, for the first part of this lemma, we get $\Gamma \vdash e_1\{x \mapsto e_2\} : t_1'$. □

Lemma 3.3 (Coercion type) *If* $\vdash t' \leq_{\mathsf{d}} t \rightsquigarrow f$, *then* $\models_{\mathsf{c}} e : t' \to t''$ *with* $\vdash t'' \leq_{\mathsf{s}} t$.

Proof. Induction on dynamic subtyping rules. □

Theorem 3.4 *All assumptions of Sect.2 are satisfied. In particular:*

A2. *If* $\Gamma \models_{\mathsf{c}} e : t$, $x \notin \mathsf{dom}(\Gamma)$, *then* $e\{x \mapsto e'\} = e$.

A3. *If* $\Gamma \models_{\mathsf{c}} e : t$ *and* $\Gamma' \leq_{\mathsf{s}} \Gamma$, *then* $\Gamma' \models_{\mathsf{c}} e : t'$, *with* $t' \leq_{\mathsf{s}} t$. *Moreover, if* $FV(e) \cap \mathsf{dom}(\Gamma') = \emptyset$, *then* $\Gamma[\Gamma'] \vdash e : t$.

A4. *If* $\Gamma \models_{\mathsf{c}} e : t$ *and* $e \xrightarrow{\mathsf{c}} e'$, *then* $\Gamma \models_{\mathsf{c}} e' : t'$ *for some* $\vdash t' \leq_{\mathsf{s}} t$.

A5. *If* $\Gamma[x{:}t_x] \models_{\mathsf{c}} e : t$, $\Gamma \models_{\mathsf{c}} e' : t_x''$, $\vdash t_x'' \leq_{\mathsf{s}} t_x'$, *and* $\vdash t_x' \leq_{\mathsf{d}} t_x \rightsquigarrow f$, *then* $\Gamma \models_{\mathsf{c}} e\{x \mapsto f\ e'\} : t'$, *for some* $t' \leq_{\mathsf{s}} t$.

Proof.

A2. Induction on the structure of e.

A3. The first part is proved by induction on typing rules. In the case (T-VAR), we have $e = x$, $t = \Gamma(x)$ and from $\Gamma' \leq_{\mathsf{s}} \Gamma$, we get $\Gamma'(x) \leq_{\mathsf{s}} \Gamma(x)$. In cases (T-LAMBDA), (T-DEL) and (T-SEL), we apply the inductive hypothesis to the premise of the rule. In cases (T-APP) and (T-SUM), we apply the inductive hypothesis to both the premises of the rule; moreover, in the case (T-APP), we exploit the transitivity property of $\leq_{\mathsf{s}}$. In the case (T-RECORD), we apply the inductive hypothesis to all premises of the rules (that is, for all $i \in I$). The moreover part is proved by induction on typing rules.

A4. Induction on reduction rules. In the case (APP), we have $(\lambda x : t_2.\, e_1)e_2 \xrightarrow{\mathsf{c}} e_1\{x \mapsto e_2\}$, with $\Gamma \models_{\mathsf{c}} (\lambda x : t_2.\, e_1)e_2 : t$. To derive this last judgment, we must have applied typing rule (T-APP) and (T-LAMBDA), hence, it must be $\Gamma[x{:}t_2] \models_{\mathsf{c}} e_1 : t$, $\Gamma \models_{\mathsf{c}} e_2 : t_2'$, with $t_2' \leq_{\mathsf{s}} t_2$. Thus, we can conclude by using Lemma 3.2.

A5. By applying Lemma 3.3 to the premise $\Gamma \models_{\mathsf{c}} e' : t_x''$, with $\vdash t_x'' \leq_{\mathsf{s}} t_x'$, and $\vdash t_x' \leq_{\mathsf{d}} t_x \rightsquigarrow f$, we get $\Gamma \vdash e''e' : \overline{t}_x$, for some $\vdash \overline{t}_x \leq_{\mathsf{s}} t_x$. Hence, we can apply Lemma 3.2

to this last judgment and the premise $\Gamma[x{:}t_x] \vdash_{\mathsf{c}} e : t$, obtaining $\Gamma \vdash_{\mathsf{c}} e\{x \mapsto e''\, e'\} : t'$, for some $\vdash t' \leq_{\mathsf{s}} t$.

□

4 Conclusion

The contribution of the paper can be summarized as follows. First, we have extended previous work introducing an abstract framework for type-safe exchange of mobile code to the (non trivial) case of open code. The outcome is a parameterized process calculus which allows to express in a simple and clean way rebinding of code in a distributed environment. In this respect, some work which has directly influenced our approach is that on dynamic software updating in, e.g., [5,6,15]. However, here we consider arbitrary core calculi rather than lambda-calculi, and an explicit language for the process layer, whereas in [5,6,15] the basic primitive is an *update* primitive which when performed changes local code in a less controlled way.

Moreover, we have adapted to a different context and to different aims the coercion semantics of subtyping, also called Penn translation [7], showing that it can be used for dynamic retrieval of code and smoothly combined with a classical subset semantics for static subtyping; our work also illustrates how this approach can be generalized to open code.

Finally, we have defined an instantiation of the framework which shows how to use Penn translation to solve the classical problem of interference of names when mobile code has a record structure [3,4,2].

Besides the already mentioned work, an important source of inspiration for the idea of coercion driven by a subtyping relation has been [11].

We plan to investigate other properties besides type safety. For instance, we would like to formalize notions like how often code is rejected and whether the original language semantics is preserved.

On the more applicative side, we would like to develop more practical and realistic examples of the use of the framework, possibly together with a prototype implementation. We plan to propose master theses on this subject. Note that an implementation should likely introduce an explicit marshaling mechanism, for distinguishing between "frozen" code to be exchanged among processes and ordinary code to be executed.

References

[1] Ancona, D., F. Damiani, S. Drossopoulou and E. Zucca, *Polymorphic bytecode: Compositional compilation for Java-like languages*, in: *ACM Symp. on Principles of Programming Languages 2005* (2005).

[2] Bettini, L., V. Bono and S. Likavec, *Safe and flexible objects with subtyping*, Journ. of Object Technology **10** (2005), pp. 5–29, special Issue: OOPS Track at SAC 2005.

[3] Bettini, L., V. Bono and B. Venneri, *Subtyping-inheritance conflicts: The mobile mixin case*, in: J.-J. Lévy, E. W. Mayr and J. C. Mitchell, editors, *TCS'04 - 3rd IFIP Int. Conf. on Theoretical Computer Science 2004* (2004), pp. 451–464.

[4] Bettini, L., B. Venneri and V. Bono, *MOMI: a calculus for mobile mixins*, Acta Informatica **42** (2005), pp. 143–190.

[5] Bierman, G., M. W. Hicks, P. Sewell and G. Stoyle, *Formalizing dynamic software updating (extended abstract)*, in: *USE'03 - the Second International Workshop on Unanticipated Software Evolution*, 2003.

[6] Bierman, G., M. W. Hicks, P. Sewell, G. Stoyle and K. Wansbrough, *Dynamic rebinding for marshalling and update, with destruct-time* λ, in: C. Runciman and O. Shivers, editors, *Intl. Conf. on Functional Programming 2003* (2003), pp. 99–110.

[7] Breazu-Tannen, V., T. Coquand, C. A. Gunter and A. Scedrov, *Inheritance as implicit coercion*, Information and Computation (1991), pp. 172–221.

[8] Fagorzi, S. and E. Zucca, *A framework for type safe exchange of mobile code*, in: U. Montanari, D. Sannella and R. Bruni, editors, *TGC 2006 - 2nd International Symposium on Trustworthy Global Computing 2006*, Lecture Notes in Computer Science 4661 (2007), pp. 319–338.

[9] Felleisen, M. and D. P. Friedman, *Control operators, the SECD-machine, and the lambda-calculus*, in: *3rd Working Conference on the Formal Description of Programming Concepts*, Ebberup, Denmark, 1986, pp. 193–219.

[10] Kobayashi, N., B. C. Pierce and D. N. Turner, *Linearity and the pi-calculus*, in: *ACM Symp. on Principles of Programming Languages 1996* (1996), pp. 358–371.

[11] Meijer, E. and P. Drayton, *Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages*, in: *OOPSLA'04 Workshop on Revival of Dynamic Languages*, 2004.

[12] Necula, G. C., *Proof-carrying code.*, in: *ACM Symp. on Principles of Programming Languages 1997* (1997), pp. 106–119.

[13] Pierce, B. C., "Types and Programming Languages," The MIT Press, 2002.

[14] Pierce, B. C. and D. Sangiorgi, *Typing and subtyping for mobile processes*, in: *Proceedings 8th IEEE Logics in Computer Science*, Montreal, Canada, 1993, pp. 376–385.

[15] Stoyle, G., M. W. Hicks, G. Bierman, P. Sewell and I. Neamtiu, *Mutatis mutandis: safe and predictable dynamic software updating*, in: *ACM Symp. on Principles of Programming Languages 2005* (2005), pp. 183–194.