An interpretation of Typed Objects into Typed π-calculus*

Davide Sangiorgi
INRIA Sophia Antipolis,
B.P. 93, 06902 Sophia Antipolis, France.

Abstract

An interpretation of Abadi and Cardelli's first-order functional *Object Calculus* into a typed π-calculus is presented. The interpretation validates the subtyping relation and the typing judgements of the Object Calculus, and is computationally adequate. This is the first interpretation of a typed Object-Oriented language into a process calculus. The study intends to offer a contribution to understanding on the one hand, the relationship between π-calculus types and conventional types of programming languages and on the other hand, the usefulness of the π-calculus as a metalanguage for the semantics of typed Object-Oriented languages.

The type language for the π-calculus has Pierce and Sangiorgi's I/O annotations, to separate the capabilities of reading and writing on a channel, and *variant types*. Technical contributions of the paper are the presentation of variant types for the π-calculus and their typing and subtying properties, and an analysis of behavioural equivalences in a π-calculus with variant types.

1 Introduction

The π-calculus [MPW92, Mil91] is a calculus of *mobile* processes, i.e., processes with a dynamically changing linkage structure. Central to the π-calculus is the notion of *name*. Two processes with acquaintance of a given name can use it to interact with each other. In the interaction, names may be exchanged and, in this way, a process can acquire the ability to communicate with other processes. A major strength of the π-calculus is its rich algebraic theory.

The notions of name and of mobility are common in many areas of computer science. A relevant example is Object-Oriented programming: Objects refer to each other using names and, during computation, object acquaintances may change and new objects may be created (this is very evident in the case of imperative or concurrent Object-Oriented languages).

Proposals of *types* and *subtypes* for the π-calculus have been recently made. Milner's *sorting* [Mil91] can be considered the first of such type systems. Pierce and Sangiorgi's system [PS96] is a development of sorting, and is the first one to allow subtyping. We now briefly recall this system. Types are assigned to names and force a discipline on what names can carry. A type shows the arity and the directionality of a name and, recursively, of the names carried by that name. For instance, a type $p : \langle S^{\mathsf{r}}, T^{\mathsf{w}}\rangle^{\mathsf{b}}$ (for appropriate type expressions S and T) says that name p can be used *both* to read and to write and that any message at p carries a pair of names; moreover, the first component of the pair can be used by the recipient *only to read*, the second *only to write* (we use "read" and "write" as synonyms for "input"

*Written: April 1996; Revised: October 1997.

and "output" respectively). Thus, process $\overline{p}\,\langle q,r\rangle .\,\mathbf{0} \mid p(x,y)\,.\,(x(z)\,.\,\mathbf{0} \mid \overline{y}\langle v\rangle\,.\,\mathbf{0})$ is well-typed under the type assignment

$$p : \langle S^{\mathsf{r}}, T^{\mathsf{w}}\rangle^{\mathsf{b}}, q : T^{\mathsf{r}}, r : S^{\mathsf{w}}, v : T.$$

(We recall that $\overline{p}\langle r_1 . . r_n\rangle\,.\,P$ is the output at p of names $r_1 . . r_n$ with continuation P, that $p(r_1 . . r_n)\,.\,P$ is an input at p with $r_1 . . r_n$ placeholders for the names received in the input, and that "|" is parallel composition.) In this system, subtyping originates from the r and w tags, which yield covariance and contravariance respectively.

Type systems for the π-calculus are useful both in revealing program errors due to the misuse of names, and in refining the algebraic theory of the π-calculus [PS96, KPT96]. They have been the subject of several recent works. Generalisations and extension of Pierce and Sangiorgi's type system include [Ode95, KPT96, Bor96, Yos96]; in particular, [KPT96] extends it with linear capabilities. Higher-order extensions are also possible, see [Tur96] for the case of parametric polymorphism. Related ideas of types, but without directionality information, can be found, for instance, in [VH93, VT93]. A general framework for these and other type systems is proposed in [Hon96].

The syntactic presentation of these π-calculus type systems is normally easy, following that of familiar type systems for sequential languages, like those for subtyping, linearity and polymorphism. But, in contrast with the latter systems where types are assigned to terms and provide us with an abstract view of their behaviour, types are assigned in the π-calculus to names (i.e., to channels) and hence reveal very little about behavioural properties of the processes. Because of this difference, the semantic relationship between the two forms of types is not obvious. For instance, what happens to the type structure of programming languages when these are translated into the π-calculus? This issue, first addressed in Turner's thesis [Tur96], is especially important on Object-Oriented languages, since certain aspects of the π-calculus, like its stress on naming and mobility and its rich algebraic theory, make it promising for the semantics of these languages. But the π-calculus will remain of little use without solid and well-understood types, because most modern Object-Oriented languages incorporate non-trivial notions of types (and of subtypes).

In short, this paper has these main motivations: (1) We wish to test the usefulness of the above-mentioned π-calculus types and investigate their relationship to familiar types of programming languages; (2) we wish to experiment with the π-calculus as a metalanguage for the semantics of *typed Object-Oriented* languages; (3) we wish to understand what is an object from a π-calculus point of view.

Following [PS96], our type system for the π-calculus has directionality information. This seems necessary in order to have meaningful forms of subtyping. But, in contrast with [PS96], our type language has a *variant* construct, as a destructor for variant values, in place of tupling. Our process operators are those of the untyped monadic π-calculus [MPW92], but with a `case` construct in place of matching. These modifications yield a rich subtype relation while keeping the calculus small and the rules for static detection of run-time errors in communications simple. Enriching the subtype relation was important because that in [PS96] is not powerful enough to describe, for instance, heterogeneous lists as in Milner's encoding of lists [Mil91]. Intuitively, in [PS96] types in the subtype relation may differ on the directionality information but are structurally the same. Our typing and subtyping rules for variants are standard. Similar rules on π-calculus-like languages have been used by Vasconcelos and Tokoro [VT93], who have record types but no subtyping, and by Pierce and Turner in PICT [PT96], who have record types with the standard subtyping rule.

We shall use this typed π-calculus to give semantics to a small but challenging typed Object-Oriented language. The π-calculus semantics validates the subtyping relation and the typing judgements of the Object-Oriented language and is computationally adequate. More precisely, we shall exhibit a translation $[\![\ ]\!]$ of types (A, B), type environments (E) and terms (a, b) of the Object-Oriented language into types, type environments and terms of the π-calculus s.t. (the translation of terms takes a name as a parameter[1]):

1. $A \leq B$ iff $[\![A]\!]^{\mathsf{w}} \leq [\![B]\!]^{\mathsf{w}}$ (*correctness of subtyping*);
2. $E \vdash a : A$ iff $[\![E]\!], p : [\![A]\!]^{\mathsf{w}^{\mathsf{w}}} \vdash [\![a]\!]_p$ (*correctness of typing judgements*);
3. $a \Downarrow$ iff $[\![a]\!]_p \Downarrow$ (*computational adequacy*);

where $\leq$ is the subtype relation and $\Downarrow$ is the convergence predicate, which on a π-calculus process indicates the possibility of performing a visible action with the environment. From these results and the compositionality of the encoding, as an easy corollary we get the soundness of the translation w.r.t. behavioural equivalences like Morris-style contextual equivalence [Bar84] or barbed congruence [MS92].

This is the first interpretation of a typed Object-Oriented language into a process calculus. The novelty of the results on the correctness of the typing and subtyping judgements of the source Object-Oriented language is, more than the results themselves, the way they are proved — via the π-calculus. Computational adequacy and soundness show that the representation of objects as π-calculus processes is also behaviourally accurate.

The proofs of computational adequacy and soundness of the translation rely on algebraic laws whose operational justification depends on types — the laws are not valid in the untyped π-calculus. These proofs are therefore significant examples of the usefulness of types for reasoning on π-calculus processes.

The source Object-Oriented language is Abadi-Cardelli's first-order functional *Object Calculus* ([AC94b]), which we denote by OC in this paper. OC has a minimal number of constructs, which can express, as primitive or derived forms, various major Object-Oriented idioms, and it has simple but interesting typing and subtyping rules. OC has built-in objects, as a collection of methods parametrised on *self*, and operators for method *selection* and method *update*. The latter allows us to replace the method of an object, and it implicitly yields a form of *inheritance*. Self is used within the methods of an object to refer to the object itself. When the object is modified, the value of self changes accordingly.

OC was first presented with direct typing and reduction rules [AC94b]. Finding an interpretation of OC into some form of typed λ-calculus has revealed to be hard. The difficulties are entirely due to types. Interpreting the *untyped* OC into the untyped λ-calculus — as well as into the untyped π-calculus — is straightforward following Kamin's *self-application* semantics [Kam88], where objects are viewed as records and methods as functions. This approach does not work for the typed calculi because the self parameter of methods, occurring in contravariant position, blocks any form of subtyping.

Only very recently a solution has been found by Abadi, Cardelli and Viswanathan [ACV96], using as target language an extension of System F with subtyping and recursive types. The translation is presented using bounded existentials and record types, which are encodable in the polymorphic λ-calculus used. The main features of this interpretation are: The separation between the select and update capabilities in the translation of object methods, the use of type abstraction, the presence of an explicit component for self in the translation of object types. Our interpretation of OC into typed π-calculus follows [ACV96] in maintaining the first of these

[1]The translation in Section 12 will actually have another parameter, a typing environment, but this could be omitted by having, for instance, more type annotations in the syntax of the Object-Oriented language.

ideas, but it does not require the second and the third one. Therefore, our type system for the π-calculus is *first order*; this is an important difference with that needed in [ACV96]; for instance, in our case type checking is decidable, whereas it is undecidable in [ACV96] (there is also a difference in the typing rules for recursive types — ours are more powerful than those in [ACV96] — but this difference is not important for the results on the interpretation of OC we present).[2] We should also stress that our variant values are name values, in the sense that they are built out of variant tags and names only, and cannot contain, for instance, process expressions.

Our encoding of OC, and the proofs of operational correspondence and computational adequacy, owe much to the studies of encodings of various forms of λ-calculi into the π-calculus, in particular [Mil92, San95]. The only previous formal study on the relationship between π-calculus types and conventional types of programming languages has been conducted by Turner [Tur96]. He takes (variants of) Milner's encodings of the λ-calculus into the π-calculus and proves that for some of these encodings there is a correspondence between principal types of the λ-terms and principal types of the encoding π-calculus terms; the π-calculus type system used is (the structural version of) Milner's sorting plus polymorphism.

Jones [Jon93] and Walker [Wal95, LW96] have already used the π-calculus as a target language for translating parallel Object-Oriented languages derived from the POOL family [Ame89] and for proving the validity of certain program transformations on the source languages. Their works show that the π-calculus captures certain Object-Oriented features and that it offers a basis for reasoning on them. The main limitation of these works is that they do not show show to handle *typed* Object-Oriented languages — the source languages have rather simple type systems and the translations do not act on types. Dealing with types is important when the type system of the Object-Oriented language contains non-trivial features like subtyping, otherwise many useful program equalities are lost and the semantics cannot be used to validate the typing rules of the language. (Further, a translation which is correct on untyped calculi may not remain correct when types are taken into account; we shall see an example of this in Section 3.) Other object-oriented features not present in the languages translated by Jones and Walker are inheritance and self.

Structure of the paper. We review the syntax, type system and operational semantics of OC in Section 2. In Section 3 we present a (naive) translation of the untyped OC into the polyadic π-calculus. The translation is correct for untyped OC, but not when types are considered. Showing this translation helps us to motivate the replacement of the ordinary matching of the π-calculus with the `case` construct — we can present the latter as a more disciplined form of conditional; it also helps to understand the final translation in Section 12. In Section 4 to 7 we present the syntax, the reduction, subtyping and typing rules of the typed π-calculus. In Section 8 we prove some basic properties of typing and subtyping, including subject-reduction and narrowing. In Section 9 we define a behavioural equivalence on the typed π-calculus (barbed congruence) and in Section 10 we show some algebraic laws for it. In Section 11 we report some derived type and process constructs. In Section 12 we define the translation of the typed OC. We prove its correctness w.r.t. subtyping and typing judgements in Section 13, and its operational correctness in Section 14. Finally, Section 15 contains some conclusions and directions for future research.

[2] By the time this paper had been written, Ramesh Viswanathan has found an interpretation of OC into a typed lambda-calculus with records and recursive types through which — we strongly believe — our interpretation of OC into π-calculus can be factorised.

2 The Object Calculus

Omitting type annotations, an object a with method names ℓ_j ($j \in 1..n$), and method bodies $\zeta(x_j).\,b_j$ is written

$$\{_{j\in 1..n}\ \ \ell_j = \zeta(x_j).\,b_j\}$$

where x_j's are the self parameters. The selection of method ℓ_i, written $a.\,\ell_i$, results in $b_i\{a/x_i\}$. The update of method ℓ_i with a new body $\zeta(x).\,b$, written $a.\,\ell_i \Leftarrow \zeta(x)b$, results in $\{_{j\in\{1..n\}-\{i\}}\ \ \ell_j = \zeta(x_j).\,b_j,\ \ell_i = \zeta(x).\,b\}$. In the derivative of the select, the self parameter x_i is replaced by the whole object. In the derivative of the update, the self parameter of non-updated methods ℓ_j, $j \neq i$, which before the update referred to the object a, at the end refers to a (possibly) different object. This dynamic behaviour of self, caused by the interplay between self and updates, gives rise to subtle typing issues.

In a method body $\zeta(x).\,b$, letter ζ binds all free occurrences of variable x in b. Alpha conversion, substitutions, free variables are defined in the usual way. A term is *closed* if it does not contain free variables. We use $=$ for syntactic equality up to alpha-conversion between expressions.

The one-step reduction relation ($\longrightarrow$) is defined on the closed terms. As a strategy, it is deterministic and "weak" (reductions underneath binders are forbidden). We write $\longrightarrow^\star$ for the reflexive and transitive closure of $\longrightarrow$, and $a \Downarrow$ to mean that there is a value b s.t. $a \longrightarrow^\star b$.

Subtyping allows an object to be replaced by another with additional methods, but the common methods must have the same type. This invariance is necessary for the soundness of the type system, i.e. avoiding selection or update of method names which do not exist. The variables bound by a type environment E are always taken to be pairwise distinct. We write $E(x)$ for the type assigned to x in E. The order of assignments in E is ignored.

We refer to [AC94b] for more discussions on the syntax, subtyping, typing, and reduction rules of OC.

Syntax

Type Environments	E	$::=$	$\emptyset \quad \mid \quad E, x : A$
Types	A, B	$::=$	$\{_{j\in 1..n}\ \ \ell_j : B_j\}$
Method names	$\ell, \mathtt{h}$		
Values	$\{_{j\in 1..n}\ \ \ell_j = \zeta(x_j : A).\,b_j\}$		
Variables	x, y, z		
Terms	a, b	$::=$	$\{_{j\in 1..n}\ \ \ell_j = \zeta(x_j : A).\,b_j\} \quad \mid \quad a.\,\ell$
			$\mid \quad a.\,\ell \Leftarrow \zeta(x : A).\,b \quad \mid \quad x$

Reduction relation

$$\frac{a = \{_{j\in 1..n}\ \ell_j = \varsigma(x_j : A).\, b_j\} \qquad i \in 1..n}{a.\,\ell_i \to b_i\{a/x_i\}} \qquad \text{(R-sel)}$$

$$\frac{a = \{_{j\in 1..n}\ \ell_j = \varsigma(x_j : A).\, b_j\} \qquad i \in 1..n}{a.\,\ell_i \Leftarrow \varsigma(x_i : B).\, b \to \{_{j\in\{1..n\}-\{i\}}\ \ell_j = \varsigma(x_j : A).\, b_j,\ \ell_i = \varsigma(x_i : A).\, b\}} \qquad \text{(R-upd)}$$

$$\frac{a \to a'}{a.\,\ell_j \to a'.\,\ell_j} \qquad \text{(R-eval1)}$$

$$\frac{a \to a'}{a.\,\ell_i \Leftarrow \varsigma(x_i : B).\, b \to a'.\,\ell_i \Leftarrow \varsigma(x_i : B).\, b} \qquad \text{(R-eval2)}$$

Subtyping rules

$$\frac{}{\{_{j\in 1..n+m}\ \ell_j : B_j\} \leq \{_{j\in 1..n}\ \ell_j : B_j\}} \qquad \text{(O-subOb)}$$

Typing rules

$$\frac{E \vdash a : A \qquad A \leq B}{E \vdash a : B} \qquad \text{(OT-subs)}$$

$$\frac{\text{for each } j \in 1..n \quad E, x_j : A \vdash b_j : B_j \qquad A = \{_{j\in 1..n}\ \ell_j : B_j\}}{E \vdash \{_{j\in 1..n}\ \ell_j = \varsigma(x_j : A).\, b_j\} : A} \qquad \text{(OT-obj)}$$

$$\frac{E \vdash a : \{_{j\in 1..n}\ \ell_j : B_j\} \qquad i \in 1..n}{E \vdash a.\,\ell_i : B_i} \qquad \text{(OT-sel)}$$

$$\frac{E \vdash a : A \qquad E, x_i : A \vdash b : B_i \qquad A = \{_{j\in 1..n}\ \ell_j : B_j\} \qquad i \in 1..n}{E \vdash a.\,\ell_i \Leftarrow \varsigma(x_i : A).\, b : A} \qquad \text{(OT-upd)}$$

$$\frac{E(x) = A}{E \vdash x : A} \qquad \text{(OT-var)}$$

2.1 Some simple properties of types

We report some facts about typing of OC terms, which we will need later.

Lemma 2.1 *If $A \leq B$ and $B \leq A$ then $A = B$.*

Lemma 2.2 *Let* $a \stackrel{\text{def}}{=} \{_{j \in 1..n}\ \ell_j = \varsigma(x_j : A).\, b_j\}$.

1. *If* $E \vdash a.\ell_i : B_i$ *then* $E \vdash a : A$ *and* A *has an* ℓ_i*-th component, say* A_i*, with* $A_i \leq B_i$.
2. *If* $E \vdash a.\ell_i \Leftarrow \varsigma(x : B).\, b : B'$ *then* $E \vdash a : A$ *and* $A \leq B \leq B'$ *and* B *has a* ℓ_i*-th component.*

Lemma 2.3 *Suppose that*

- $E \vdash a : A$,
- $E, x : A \vdash b : B$.

Then $E \vdash b\{a/x\} : B$

Lemma 2.4 *If* $E, x : A \vdash a : B$ *and* $A' \leq A$ *then also* $E, x : A' \vdash a : B$.

Lemma 2.5 (minimum types for OC) *If* $E \vdash a : A$ *, then there is* B *s.t.* $E \vdash a : B$ *and, for any* A'*, if* $E \vdash a : A'$ *then* $B \leq A'$.

3 A translation of the untyped OC into the polyadic π-calculus

To motivate the presence of variant types in the π-calculus, we first show a naive and (in our opinion) most natural encoding of the untyped OC into the polyadic π-calculus, and explain why the translation does not work at the level of types. (A translation of the untyped OC similar to that in this section has been given by Hüttel and Kleist [HK96].)

We briefly recall the operators of the polyadic π-calculus. We use P, Q to range over processes and $p, q, r, .., x, y, z$ to range over names. With some abuse of notation we let symbols $x, y, z..$ be both OC variables and π-calculus names, and ℓ be both a method name and a π-calculus name. By convention, we assume that p, q, r, ℓ are not OC variables. $\mathbf{0}$ is the inactive process; $P \mid Q$ is the parallel composition of processes P and Q; the restriction $(\boldsymbol{\nu} x\,)P$ makes name x local to process P; $x(y_1..y_n).P$ is an input at x with $y_1..y_n$ as placeholders for the names received, and P as continuation; $\overline{x}\langle y_1..y_n\rangle.P$ is the output of names $y_1..y_n$ at x with continuation P; the matching and mismatching constructs $[x = y]P$ and $[x \neq y]P$ release process P if x and y are the same (matching) or different (mismatching) names; $!P$ is the replication of P, therefore it represents infinite many copies of P in parallel. We abbreviate a term $\overline{x}\langle y_1..y_n\rangle.\mathbf{0}$ as $\overline{x}\langle y_1..y_n\rangle$; a term $\overline{x}\langle y\rangle.P$ as $\overline{x}\,y.P$; a term $(\boldsymbol{\nu} y\,)\overline{x}\,y.P$ as $\overline{x}(y).P$; a term $P_1 \mid .. \mid P_n$ as $\prod_{j\in 1..n} P_j$. We assign parallel composition the lowest syntactic precedence among the operators. Input and restriction are binding constructs and give rise to the expected definitions of alpha conversion, name substitutions, free and bound names of a process.

The translation of the untyped OC is defined structurally using the rules below. By alpha conversion we assume that the same symbol is used for the self variables of the methods of an object.

$$
\begin{aligned}
[\![\{_{j\in 1..n}\ \ell_j = \varsigma(y).\, b_j\}]\!]_p &\stackrel{\text{def}}{=} \overline{p}(x).\,!x(\ell, r, y).\Big(\textstyle\prod_{j\in 1..n}[\ell = \ell_j][\![b_j]\!]_r\Big)\\
[\![a.\ell_j]\!]_p &\stackrel{\text{def}}{=} (\boldsymbol{\nu} q\,)\Big([\![a]\!]_q \mid q(x).\,\overline{x}\langle \ell_j, p, x\rangle\Big)\\
[\![a.\ell_j \Leftarrow \varsigma(y)b]\!]_p &\stackrel{\text{def}}{=} (\boldsymbol{\nu} q\,)\Big([\![a]\!]_q \mid q(x).\,\overline{p}(x_{\texttt{new}}).\,!x_{\texttt{new}}(\ell, r, y).\\
&\qquad\qquad \Big([\ell = \ell_j][\![b]\!]_r \mid [\ell \neq \ell_j]\overline{x}\langle \ell, r, y\rangle\Big)\Big)\\
[\![x]\!]_p &\stackrel{\text{def}}{=} \overline{p}\,x
\end{aligned}
$$

As usual, by convention we assume that in each clause of the translation different name symbols stand for different names. Moreover, the symbol names introduced by the translation are supposed to be fresh; these names are x, ℓ, r in the clause for a value, q in the clause for selection, $q, x, x_{\mathtt{new}}, \ell, r$ in the clause for update.

The translation of an object value at p — the *location* of the object — is a process which signals its valuehood by emitting a pointer x to the *value-core* of the object. The value-core is a process which accepts requests for method selection, each request consisting of three parameters: The name ℓ of the method, the location r to be used for the next interaction, and a pointer y to the value-core of the actual self parameter. In the translation of the update of a method ℓ_j of an object a, upon receiving a pointer x to the value-core of a, a new object is created which locally processes every request for method ℓ_j using the new method body, and which forwards any other request along x.

One can prove results of operational correspondence for this encoding similar to those for Milner's encoding of call-by-value λ-calculus [Mil92, San92]. The above translation actually allows *method extension*, in case of updates of methods which do not exist. Method extension is reasonable on untyped calculi; it can be prevented by further elaborating the translation of update. In either case, writing $P \approx Q$ to mean that processes P and Q are behaviourally undistinguishable (formally, $\approx$ is weak barbed congruence, see Section 9), we have, for all closed terms a, b (we write $\longrightarrow\approx$ for the composition of the two relations $\longrightarrow$ and $\approx$):

1. If $a \longrightarrow b$ then $[\![a]\!]_p \longrightarrow\approx [\![b]\!]_p$;
2. conversely, if $[\![a]\!]_p \longrightarrow P$ then there is b s.t. $a \longrightarrow b$ and $P \approx [\![b]\!]_p$.

The proofs are similar to those we shall give in Section 14.

However the translation is unsatisfactory as an interpretation of typed OC because, assuming the type system in [PS96] for the π-calculus, the π-calculus types do not agree with the OC types. The problem appears at the level of subtyping. In a target process $[\![a]\!]_p$, name p would have type $\langle S, T, U\rangle^{\mathsf{w}\mathsf{w}}$, for appropriate types S, T, and U. The outermost tag is w because p may be used only in output position by $[\![a]\!]_p$; then $\langle S, T, U\rangle^{\mathsf{w}}$ is the type of a pointer emitted at p, which may be used by its recipient only in output; and U is the type of (a pointer to) the actual self parameter. In $\langle S, T, U\rangle^{\mathsf{w}}$, type U is underneath an odd number of w tags, hence it is in contravariant position. Moreover, since the value of self may be the object a itself, U must be a subtype of $\langle S, T, U\rangle^{\mathsf{w}}$. These two facts — U occurring in contravariant position and having to be a subtype of $\langle S, T, U\rangle^{\mathsf{w}}$ — prevent any possibility of subtyping between the types of the locations of different objects.

The failure is similar to that of the self-application interpretation of objects into the λ-calculus [Kam88]: The self parameter is completely exposed in a contravariant position. Indeed, the translation can be read as a (functional) encoding of records whose fields are functions with the same argument and where record update is achieved by creating a new record which shares the non-updated components with the old one.

It is worth noticing, in the above translation, the use of the π-calculus constructs of matching and mismatching for modeling objects. This suggests that one might hope to recover the subtyping rule for objects — whereby object types with different sets of methods are related — by allowing forms of subtyping on the conditional constructs, absent in the type system in [PS96]. In the following sections, we shall develop this idea, introducing a more refined form of conditional. Then we shall refine the translation in this section to obtain one which is correct for (*typed*) OC.

4 The syntax of the typed π-calculus

The table below gives the syntax for types and processes of the typed π-calculus that we shall use as the target of the interpretation of OC. The underlying process constructs are those of the monadic π-calculus [MPW92], with matching replaced by a `case` construct. The latter can be thought of as a more disciplined form of matching, in which all tests on a given name are localised to a single place. The syntax chosen for `case` is reminiscent of an analogous construct in [MPW92]. In the untyped calculus, matching and `case` are interderivable, but in the typed calculus `case` allows us simple but powerful typing and subtyping rules with which, moreover, any misuse of variant values in communications is easy to detect (rule R-CASE-WRONG, Section 5). We have omitted summation, since we will not need it in the interpretation of OC. Restriction is explicitly typed to facilitate certain proofs involving the type system (we are not interested in type inference in this paper). *Substitutions*, ranged over by σ, are functions from names to values; for an expression e, which could be a process or a value, $e\sigma$ is the result of applying σ onto e, with the usual renaming to avoid captures of bound names. Substitutions have tighter syntactic precedence than the process operators. In a prefix $p(x) \,.\, P$ or $\overline{p}\, w \,.\, P$ we call p the *subject* of the prefix.

The most important difference w.r.t. the monadic π-calculus is the addition of variant values. Syntactically, this introduces a vertical dimension on values, as opposite to the tupling construct of the polyadic π-calculus, which introduces a horizontal dimension. We should stress that the variant values are rather simple, in that they are constructed out of names and variant tags only and therefore do not contain terms of the language.

The construct `wrong` stands for a process in which a run-time type error has occurred, i.e., a communication in which the variant tag of the transmitted value was unexpected by its recipient or a violation of an I/O restriction. The soundness theorem in Section 13 guarantees that a well-typed process expression cannot reduce to an expression containing `wrong`.

The requirement that, in a `case` statement, tags ℓ_i be pairwise distinct can be removed at the price of allowing non-deterministic choices on the continuation branches. In a `case` branch $\ell_i\_(x_i) \rhd P_i$, name x_i is bound in P_i. We sometimes abbreviate an expression $[\,\ell_1\_(y_1) \rhd P_1 \,;\, ..;\, \ell_n\_(y_n) \rhd P_n\,]$ as $[_{j\in 1..n}\,\ell_j\_(y_j) \rhd P_j]$, and similarly for variant types $[\,\ell_1 : T_1 .. \ell_n : T_n\,]$.

We have chosen different syntactic categories for names and variant tags. This is a matter of taste: Another possibility would be to have a single syntactic category and then to use *distinctions* [MPW92] to make variant tags constants.

Syntax

Names

$$p, q, r\,..\,x, y, z$$

Variant Tags

$$\ell, \mathtt{h}$$

Types

$$\begin{array}{lcll} T & ::= & \mu X\,.\,T & \text{recursive type} \\ & \mid & X & \text{type variable} \\ & \mid & [\,\ell_1\_T_1\,..\,\ell_n\_T_n\,] & \text{variant type} \\ & \mid & T^I & \text{channel type} \end{array}$$

I/O Tags

$$\begin{array}{lcll} I & ::= & \mathsf{r} & \text{input only} \\ & \mid & \mathsf{w} & \text{output only} \\ & \mid & \mathsf{b} & \text{either} \end{array}$$

Values

$$\begin{array}{lcll} v & ::= & x & \text{name} \\ & \mid & \ell\_v & \text{variant value} \end{array}$$

Processes

$$\begin{array}{lcll} P & ::= & \mathbf{0} & \text{nil process} \\ & \mid & P \mid P & \text{parallel} \\ & \mid & (\boldsymbol{\nu} x : T)\,P & \text{restriction} \\ & \mid & p(x)\,.\,P & \text{input} \\ & \mid & \overline{p}\,v\,.\,P & \text{output} \\ & \mid & !P & \text{replication} \\ & \mid & \mathtt{case}\ v\ \mathtt{of}\ [\,\ell_1\_(x_1) \,\triangleright\, P_1\,;\,..\,;\,\ell_n\_(x_n) \,\triangleright\, P_n\,] & \text{case} \\ & \mid & \mathtt{wrong} & \text{error} \end{array}$$

where:

- In a recursive type $\mu X\,.\,T$, variable X must be guarded in T, i.e., occur underneath an I/O-tag or underneath a variant tag;
- in the `case` statement, the tags ℓ_i $(i \in 1\,..\,n)$ are pairwise distinct.

5 Reduction semantics

Following Milner [Mil91], the one-step reduction relation $\longrightarrow$ of the calculus exploits the auxiliary relation $\equiv$ of structural congruence to bring the participants of a potential communication into contiguous positions. W.r.t. Milner [Mil91], the new rules are R-Case, which acts as a destructor for variant values, and R-Case-Wrong, which signals a run-time error on the manipulation of variant values.

Structural congruence

The *structural congruence relation* $\equiv$ is the least congruence on processes which is closed under the following rules:

1. $P \mid Q \equiv Q \mid P,\ \ P \mid (Q \mid R) \equiv (P \mid Q) \mid R,\quad P \mid \mathbf{0} \equiv P;$
2. $(\boldsymbol{\nu} p : T)\, \mathbf{0} \equiv \mathbf{0},\quad (\boldsymbol{\nu} p : T)\, (\boldsymbol{\nu} q : S)\, P \equiv (\boldsymbol{\nu} q : S)\, (\boldsymbol{\nu} p : T)\, P;$
3. $((\boldsymbol{\nu} p : T)\, P) \mid Q \equiv (\boldsymbol{\nu} p : T)\, (P \mid Q)$, if p not free in Q;
4. $!P \equiv P \mid !P;$
5. $C[\mathtt{wrong}] \equiv \mathtt{wrong}$, for any context C.

Rule 5 can be split into smaller local rules like $P \mid \mathtt{wrong} \equiv \mathtt{wrong}$ and $(\boldsymbol{\nu} p : S)\, \mathtt{wrong} \equiv \mathtt{wrong}$.

Reduction relation

$$\frac{}{\overline{p}\, v \,.\, P \mid p(x) \,.\, Q \longrightarrow P \mid Q\{v/x\}} \quad \text{(R-Comm)}$$

$$\frac{\ell_j \in \{\ell_1 .. \ell_n\}}{\mathtt{case}\ \ell_j\_v\ \mathtt{of}\ [\, \ell_1\_(x_1) \,\triangleright\, P_1 \,;\, ..\,;\, \ell_n\_(x_n) \,\triangleright\, P_n \,] \longrightarrow P_j\{v/x_j\}} \quad \text{(R-Case)}$$

$$\frac{\ell_j \notin \{\ell_1 .. \ell_n\}}{\mathtt{case}\ \ell_j\_v\ \mathtt{of}\ [\, \ell_1\_(x_1) \,\triangleright\, P_1 \,;\, ..\,;\, \ell_n\_(x_n) \,\triangleright\, P_n \,] \longrightarrow \mathtt{wrong}} \quad \text{(R-Case-Wrong)}$$

$$\frac{P \longrightarrow P'}{P \mid Q \longrightarrow P' \mid Q} \quad \text{(R-Par)}$$

$$\frac{P \longrightarrow P'}{(\boldsymbol{\nu} p : S)\, P \longrightarrow (\boldsymbol{\nu} p : S)\, P'} \quad \text{(R-Restr)}$$

$$\frac{P \equiv P' \longrightarrow Q' \equiv Q}{P \longrightarrow Q} \quad \text{(R-Eqv)}$$

For any name p, the observation predicate $\downarrow_p$ denotes the possibility of a process immediately performing a communication with the external environment along p. Thus, $P \downarrow_p$ holds if P has a prefix $p(x)$ or $\overline{p}\, v$ which is not underneath another prefix or in a `case` construct, and not in the scope of a restriction on p. For example, if $P = (\boldsymbol{\nu} r : T)\, (\overline{r}\, v \,.\! \mid p(x) \,.\, q(y)\,)$, then $P \downarrow_p$, but not $P \downarrow_r$, or $P \downarrow_q$. We write $P \downarrow$ if there is p s.t. $P \downarrow_p$. We write $P \longrightarrow_{\mathtt{d}} P'$ if $P \longrightarrow P'$ is the only reduction that P can perform, i.e., $P \longrightarrow P''$ implies $P' \equiv P''$, and there is no p s.t. $P \downarrow_p$. That is, in any context $P \longrightarrow P'$ is necessarily the first action which P can participate in. We write $\longrightarrow^\star$ and $\longrightarrow_{\mathtt{d}}^\star$ for the reflexive and transitive closures of $\longrightarrow$ and $\longrightarrow_{\mathtt{d}}$, respectively. Finally, we write $P \Downarrow_p$ if there is P' s.t. $P \longrightarrow^\star P' \downarrow_p$, and $P \Downarrow$ if there is p s.t. $P \Downarrow_p$.

6 Subtyping

Subtyping judgements are of the form $\Sigma \vdash S \leq T$, where Σ represents the subtyping assumptions. We often write $S \leq T$ when the subtyping assumptions are empty. The subtyping rule for variant is standard. The remaining rules follow [PS96]. We recall from [PS96] that type annotation r (an input capability) gives covariance, w (an output capability) gives contravariance, and b (both capabilities) gives invariance. Moreover, since a tag b gives more freedom in the use of a name, for each type T we have $T^{\mathsf{b}} \leq T^{\mathsf{r}}$ and $T^{\mathsf{b}} \leq T^{\mathsf{w}}$.

Subtyping assumptions

$$\Sigma \quad ::= \quad \emptyset \quad | \quad \Sigma, S \leq T$$

Subtyping rules

$$\frac{\Sigma \vdash S \leq T \qquad \Sigma \vdash T \leq S}{\Sigma \vdash S^{\mathsf{b}} \leq T^{\mathsf{b}}} \qquad \text{(A-BB)}$$

$$\frac{I \in \{\mathsf{b}, \mathsf{r}\} \qquad \Sigma \vdash S \leq T}{\Sigma \vdash S^{I} \leq T^{\mathsf{r}}} \qquad \text{(A-XI)}$$

$$\frac{I \in \{\mathsf{b}, \mathsf{w}\} \qquad \Sigma \vdash T \leq S}{\Sigma \vdash S^{I} \leq T^{\mathsf{w}}} \qquad \text{(A-XO)}$$

$$\frac{\text{for each } i \in 1..n \qquad \Sigma \vdash S_i \leq T_i}{\Sigma \vdash [\,\ell_1\_S_1 .. \ell_n\_S_n\,] \leq [\,\ell_1\_T_1 .. \ell_{n+m}\_T_{n+m}\,]} \qquad \text{(A-Case)}$$

$$\Sigma, S{\leq}T, \Sigma' \vdash S \leq T \qquad \text{(A-Ass)}$$

$$\frac{\Sigma, \mu X. S \leq T \vdash S\{{}^{\mu X. S}/_{X}\} \leq T}{\Sigma \vdash \mu X. S \leq T} \qquad \text{(A-Rec-L)}$$

$$\frac{\Sigma, S \leq \mu X. T \vdash S \leq T\{{}^{\mu X. T}/_{X}\}}{\Sigma \vdash S \leq \mu X. T} \qquad \text{(A-Rec-R)}$$

6.1 Properties of subtyping

The proofs of the results below are similar to analogous results in [PS96], and exploit a notion of tree simulation between unfolded types, along the lines of that introduced by Amadio and Cardelli on λ-calculus with subtyping and recursive types [AC93].

Proposition 6.1 *The relation $\leq$ is reflexive and transitive.*

Proof: The proof of transitivity is along the lines of the proof of an analogous result in [PS96]. First, define, for each type T, a tree $\mathit{Tree}(T)$ as the infinite unfolding of T. Then define a notion of simulation between trees and prove that, for any types T and S, it holds that $T \leq S$ iff $\mathit{Tree}(S)$ simulates $\mathit{Tree}(T)$ (for this, one needs to prove that the subtyping algorithm is terminating). Finally, prove that the simulation relation among trees is transitive. $\square$

Lemma 6.2 *$\Sigma \vdash S \leq T$ implies $\Sigma, \Sigma' \vdash S \leq T$ for any Σ'.*

Lemma 6.3 *If $S \leq T$ and $\Sigma, S \leq T \vdash U_1 \leq U_2$, then $\Sigma \vdash U_1 \leq U_2$.*

Proof: By induction on the depth of the proof of $\Sigma, S \leq T \vdash U_1 \leq U_2$. $\square$

Lemma 6.4 $\mu X . S \leq S\{^{\mu X . S}/_X\} \leq \mu X . S$

Proof: From reflexivity and rule A-Rec-L or A-Rec-R. $\square$

Lemma 6.5 *If $S \leq U^{\mathsf{r}}$ and $S \leq T^{\mathsf{w}}$, then $T \leq U$.*

Proof: By Lemma 6.4 and transitivity (and because of the assumption that, in any recursive type, the recursion variable is guarded in the body), we may assume that S is of the form S_1^I. From rules A-XI and A-XO we deduce $S_1 \leq U$ and $T \leq S_1$. Then $T \leq U$ follows by transitivity of subtyping. $\square$

7 Typing

A *type environment* is a finite assignment of types to names. We follow the same convention of type environments for OC, and therefore write $\Gamma(x)$ for the type assigned to x in Γ.

Type Environments

$$\Delta, \Gamma \quad ::= \quad \emptyset \quad | \quad \Gamma, p : T$$

A typing judgement $\Gamma \vdash P$ asserts that process P is well-typed in Γ, and $\Gamma \vdash v : T$ that value v has type T in Γ. There is one typing rule for each process construct except `wrong`. The interesting rules are those for input and output prefixes and for `case`. In the rules for input and output prefixes, the subject of the prefix is checked to possess the appropriate input or output capability in the type environment. Tv-sub is the only rule which explicitly uses subtyping. $\mathtt{Pr}_\Gamma$ is the class of processes well-typed in Γ.

Value typing

$$\frac{\Gamma(p) = T}{\Gamma \vdash p : T} \quad \text{(Tv-base)}$$

$$\frac{\Gamma \vdash v : S \qquad S \leq T}{\Gamma \vdash v : T} \quad \text{(Tv-sub)}$$

$$\frac{\Gamma \vdash v : T}{\Gamma \vdash \ell\_v : [\,\ell\_T\,]} \quad \text{(Tv-var)}$$

Process typing

$$\frac{}{\Gamma \vdash \mathbf{0}} \quad \text{(T-Nil)}$$

$$\frac{\Gamma \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash P \mid Q} \quad \text{(T-Par)}$$

$$\frac{\Gamma \vdash P}{\Gamma \vdash !P} \quad \text{(T-Repl)}$$

$$\frac{\Gamma, x : S^I \vdash P}{\Gamma \vdash (\boldsymbol{\nu} x : S^I)\, P} \quad \text{(T-Restr)}$$

$$\frac{\Gamma \vdash p : S^{\mathsf{r}} \qquad \Gamma, x : S \vdash P}{\Gamma \vdash p(x) \,.\, P} \quad \text{(T-In)}$$

$$\frac{\Gamma \vdash p : S^{\mathsf{w}} \qquad \Gamma \vdash w : S \qquad \Gamma \vdash P}{\Gamma \vdash \overline{p}\, w \,.\, P} \quad \text{(T-Out)}$$

$$\frac{\Gamma \vdash v : [\, \ell_1\_T_1 \,..\, \ell_n\_T_n \,] \qquad \text{for each } i \in 1..n \quad \Gamma, x_i : T_i \vdash P_i}{\Gamma \vdash \mathtt{case}\; v \;\mathtt{of}\; [\, \ell_1\_(x_1) \rhd P_1 \,;\, ..\,;\, \ell_n\_(x_n) \rhd P_n \,]} \quad \text{(T-Case)}$$

8 Properties of the typing system

We establish some basic fundamental properties of the typing relation, including weakening, contraction, substitution and narrowing. We also show that the type relation is sound w.r.t. the reduction relation of the calculus: Typing is preserved under reductions and a well-typed program can never result in a run-time error.

Lemma 8.1 (Weakening) *If* $\Gamma \vdash P$ *then* $\Gamma, x : S \vdash P$ *for any type* S *and any name* x *on which* Γ *is not defined.*

Lemma 8.2 (Contraction) *If* $\Gamma, x : S \vdash P$ *and* x *is not free in* P*, then* $\Gamma \vdash P$ *.*

Lemma 8.3 (Substitution) *Suppose*

- $\Gamma \vdash P$,
- $\Gamma(x) = T$
- $\Gamma \vdash v : T$.

Then $\Gamma \vdash P\{v/x\}$.

Lemma 8.4 *If* $\Gamma \vdash P$ *and* $P \equiv P'$*, then* $\Gamma \vdash P'$ *.*

Lemma 8.5 *If* $\Gamma \vdash P$ *then* $P \not\equiv \mathtt{wrong}$.

Theorem 8.6 (subject reduction) *If* $\Gamma \vdash P$ *and* $P \longrightarrow P'$ *then* $\Gamma \vdash P'$ *.*

Proof: By induction on the depth of the proof of $P \longrightarrow P'$. For the cases of R-Comm and R-Case, Lemma 8.3 is needed. For the case of R-Eqv, Lemma 8.4 is needed. □

Corollary 8.7 (no run-time errors) *If* $\Gamma \vdash P$ *and* $P \longrightarrow^{\star} P'$ *then* $P' \not\equiv \texttt{wrong}$.

Proof: Follows from Theorem 8.6 and Lemmas 8.4 and 8.5. $\square$

Lemma 8.8 (narrowing on values) *If* $\Gamma, p : S \vdash w : U$ *and* $T \leq S$ *then also* $\Gamma, p : T \vdash w : U$ *.*

Proof: Use definition of typing on values. $\square$

Theorem 8.9 (narrowing on processes) *If* $\Gamma, p : S \vdash P$ *and* $T \leq S$*, then also* $\Gamma, p : T \vdash P$ *.*

Proof: Structural induction on P. For output and input prefixes, and for `case`, use Lemma 8.8. $\square$

Lemma 8.10 *If* $\Gamma \vdash p(x) . P \mid \overline{p} v . Q$ *, then for some* T*, it holds that* $\Gamma(p) = T^{\mathsf{b}}$*,* $\Gamma \vdash v : T$ *and* $\Gamma, x : T \vdash P$ *.*

Lemma 8.11 *If* $\Gamma \vdash \overline{p} x . P$ *and* $\Gamma(p) = T^{\mathsf{w}}$ *, then* $\Gamma(x) \leq T$*.*

Lemma 8.12 *If* $\Gamma, p : T^{\mathsf{b}} \vdash p(y) . P$*, then* $\Gamma, p : T^{\mathsf{b}}, y : T \vdash P$*.*

9 Barbed bisimulation and congruence

We define behavioural equality using the notion of barbed bisimulation [MS92]. The main advantage of barbed bisimulation is that it can be defined uniformly on different calculi, which is useful when studying a new calculus or a refinment of an existing one as we are doing here. Moreover, as a bisimulation, barbed bisimulation comes equipped with the co-induction proof technique.

The definition of barbed bisimulation uses the reduction relation of the calculus along with the observation predicate $\downarrow_p$ (Section 5), for each port p. By itself, barbed bisimulation is a rather coarse relation. Better discriminating power is achieved by considering the induced congruence, called *barbed congruence*. It can be shown [San92] that barbed congruence coincides in both CCS and the π-calculus with the standard bisimilarity congruences.

In a typed calculus, the processes being compared must obey the same typing and the contexts employed must be compatible with this typing. We call a (Γ/Δ)-context a context which, when filled in with a process obeying typing Δ, becomes a process obeying typing Γ. Typing Γ might contain names not in Δ; the converse might be true too, because of binders in the context which embrace the hole.

First, a few technical definitions and lemmas.

Definition 9.1 *We write* $\Gamma <_{\mathrm{E}} \Delta$ *if, for each* x *on which* Δ *is defined,* Γ *also is defined and* $\Gamma(x) \leq \Delta(x)$*.*

Definition 9.2 *A substitution* σ *is* legal *in a type environment* Γ *if for all* x *on which* Γ *is defined, it holds that* $\Gamma \vdash x\sigma : \Gamma(x)$ *.*

Lemma 9.3 *If* $\Gamma <_{\mathrm{E}} \Delta$*,* σ *is legal for* Γ *and* $\Delta \vdash R$ *, then* $\Gamma \vdash R\sigma$ *.*

Proof: By narrowing and substitution lemmas. $\square$

Lemma 9.4 *If* $\Gamma \vdash v : T$ *and* σ *is legal for* Γ*, then* $\Gamma \vdash v\sigma : T$ *.*

Proof: Induction on the depth of the proof of $\Gamma \vdash v : T$. $\square$

Definition 9.5 ((Γ/Δ)-context) *Given type environments Γ and Δ and a process context C, we say that C is a (Γ/Δ)-context if $\Gamma \vdash C$ assuming the following typing rule for the hole $[\cdot]$ of C:*

$$\frac{\Gamma' <_{\mathrm{E}} \Delta}{\Gamma' \vdash [\cdot]}$$

(where Δ is one of the given type environments and Γ' is a metavariable over type environments).

Lemma 9.6 *If C is a (Γ/Δ)-context and $\Gamma' <_{\mathrm{E}} \Gamma$ then C is a (Γ'/Δ)-context.*

Proof: Induction on the structure of C and narrowing. $\square$

Definition 9.7 (barbed bisimulation) *A relation $\mathcal{R} \subseteq \mathtt{Pr}_\Delta \times \mathtt{Pr}_\Delta$ is a* barbed Δ-bisimulation *if $(P, Q) \in \mathcal{R}$ implies:*

1. *if $P \longrightarrow P'$ then there exists Q' such that $Q \longrightarrow Q'$ and $(P', Q') \in \mathcal{R}$;*
2. *if $Q \longrightarrow Q'$ then there exists P' such that $P \longrightarrow P'$ and $(P', Q') \in \mathcal{R}$;*
3. *for each name p, $P \downarrow_p$ iff $Q \downarrow_p$.*

Two processes P and Q are barbed Δ-bisimilar, *written $P \stackrel{\centerdot}{\sim}_\Delta Q$, if $(P, Q) \in \mathcal{R}$, for some barbed Δ-bisimulation $\mathcal{R}$.*

Definition 9.8 (barbed congruence) *Two processes $P, Q \in \mathtt{Pr}_\Delta$ are* barbed Δ-congruent, *written $P \sim_\Delta Q$, if, for each type environment Γ and (Γ/Δ)-context C, we have $C[P] \stackrel{\centerdot}{\sim}_\Gamma C[Q]$.*

In the remainder of the paper, we write $P \sim_\Gamma Q$ without recalling the assumption that P and Q are well-typed in Γ.

Barbed congruence requires a quantification over all contexts. The Context Lemma below shows how to lighten this requirement: It is enough to test processes using parallel composition and legal substitutions.

Definition 9.9 *Two processes $P, Q \in \mathtt{Pr}_\Delta$ are* barbed Δ-equivalent, *written $P \sim^{\mathsf{e}}_\Delta Q$, if for each environment Γ, substitution σ, and process $Q \in \mathtt{Pr}_\Gamma$ such that*

1. *$\Gamma <_{\mathrm{E}} \Delta$,*
2. *σ is legal in Γ,*

it holds that $Q \mid P_1\sigma \stackrel{\centerdot}{\sim}_\Gamma Q \mid P_2\sigma$.

Lemma 9.10 *If $P \sim^{\mathsf{e}}_\Delta Q$ and $\Gamma <_{\mathrm{E}} \Delta$, then $P \sim^{\mathsf{e}}_\Gamma Q$.*

Lemma 9.11 *If $P \sim^{\mathsf{e}}_\Delta Q$ and $S \leq \Delta(p)$, then $(\boldsymbol{\nu} p : S)\, P \sim^{\mathsf{e}}_{\Delta - p} (\boldsymbol{\nu} p : S)\, Q$.*

Lemma 9.12 (Context Lemma for barbed congruence) *Relations $\sim^{\mathsf{e}}_\Delta$ and $\sim_\Delta$ coincide.*

Proof: One proves, by induction on the structure of C, that:

$$P \sim^{\mathsf{e}}_\Delta Q \quad \text{implies} \quad C[P] \sim^{\mathsf{e}}_\Gamma C[Q]$$

for all (Γ/Δ)-context C. The basic case, and the cases of restriction and parallel composition, are immediate using Lemma 9.10 and Lemma 9.11. For the remaining cases one defines appropriate bisimulations. $\square$

The weak version of the equivalences, where one abstracts away from the length of reductions, is obtained in the standard way. *Weak barbed Δ-bisimulation*, written $\stackrel{\centerdot}{\approx}_\Delta$, is defined by replacing in Definition 9.7 the transition $Q \longrightarrow Q'$ with $Q \longrightarrow^\star Q'$ and the predicate $\downarrow_p$ with $\Downarrow_p$. Similarly, *weak barbed Δ-congruence*, written $\approx_\Delta$, and *weak barbed Δ-equivalence* are defined by replacing $\stackrel{\centerdot}{\sim}_\Gamma$ with $\stackrel{\centerdot}{\approx}_\Gamma$ in Definition 9.8 and 9.9. The counterpart of Lemma 9.12 for the weak case is true.

10 Some useful algebraic laws

We report some laws for barbed congruence. Some of the laws crucially rely on the type information, like law **L3** of Lemma 10.2 and the replication theorems 10.5 and 10.6. They show the importance of types for reasoning on processes. We recall that when we write a typed equality like $P \sim_\Gamma Q$ we assume that P and Q are well-typed in Γ.

Lemma 10.1 *If $P \equiv Q$ then $P \sim_\Gamma Q$.*

Lemma 10.2 **L1** $(\boldsymbol{\nu} r : T)\,(\overline{p}\, r \mid !r(x)\,.\,R) \sim_\Gamma (\boldsymbol{\nu} r : T)\,(\overline{p}\, r\,.\,!r(x)\,.\,R)$.

- **L2** $(\boldsymbol{\nu} r_1 : T_1 . . r_n : T_n)(P \mid !r_1(x)\,.\,R_1 \mid . . \mid !r_n(x)\,.\,R_n) \sim_\Gamma P$, *if $r_1 . . r_n$ are not free in P.*

- **L3** $\texttt{case}\; v\; \texttt{of}\; \left[_{j \in 1..n}\, \ell_j\_(y_j) \rhd P_j\right] \sim_\Gamma \texttt{case}\; v\; \texttt{of}\; \left[_{j \in 1..n+m}\, \ell_j\_(y_j) \rhd P_j\right]$.

- **L4** $(\boldsymbol{\nu} x : T)\, P \sim_\Gamma (\boldsymbol{\nu} x : S)\, P$.

- **L5** $(\boldsymbol{\nu} r : T)\,(\alpha.\, P \mid !r(x)\,.\,Q) \sim_\Gamma \alpha.\,(\boldsymbol{\nu} r : T)\,(P \mid !r(x)\,.\,Q)$, *if r does not appear in α and any name bound in α does not appear free in $r(x)\,.\,Q$.*

Proof: For law **L3**: for both processes to be well-typed in Γ it must be $\Gamma \vdash v : [\,_{j \in 1..n}\, \ell_j\_T_j\,]$, for some type T_j's. Now, consider a substitution which is legal for Γ. By Lemma 9.4, $\Gamma \vdash v\sigma : [\,_{j \in 1..n}\, \ell_j\_T_j\,]$. The only closed values of type $[\,_{j \in 1..n}\, \ell_j\_T_j\,]$ have the form $\ell_j\_v_j$ ($j \in 1..n$). Then the thesis follows from the reduction relation of `case` and the Context Lemma 9.12. $\square$

Law **L3** shows that in a case construct we can always add (well-typed) branches.

Next, we report some distributivity laws for private replications, i.e., systems of the form

$$(\boldsymbol{\nu} p : T)\,(P \mid !p(x)\,.\,R)$$

in which process P possesses only the output capability on name p. One should think of R as a private resource of P, for P is the only process who can access R; indeed P can activate as many copies of R as needed.

We have omitted the proofs of Lemma 10.3 and 10.4, which are not immediate but along the lines of the proofs of similar results in [San92] and [PS96].

Lemma 10.3 *Suppose that*

- $\Gamma \vdash Q \mid (\boldsymbol{\nu} m : T^{\mathsf{b}})\,(P_1 \mid P_2 \mid !m(x)\,.\,R)$,
- $\Gamma, m : T^{\mathsf{w}} \vdash P_1 \mid P_2$,
- $\Gamma, m : T^{\mathsf{w}}, x : T \vdash R$,
- $m : T^{\mathsf{b}} \not\vdash \overline{m}\, m\,.\,\mathbf{0}$.

Then $Q \mid (\boldsymbol{\nu} m : T^{\mathsf{b}})\,(P_1 \mid P_2 \mid !m(x)\,.\,R) \overset{\centerdot}{\sim}_\Gamma$
$Q \mid (\boldsymbol{\nu} m : T^{\mathsf{b}})\,(P_1 \mid !m(x)\,.\,R) \mid (\boldsymbol{\nu} m : T^{\mathsf{b}})\,(P_2 \mid !m(x)\,.\,R)$.

Lemma 10.4 *Suppose that*

- $\Gamma \vdash Q \mid (\boldsymbol{\nu} m : T^{\mathsf{b}})\,(!P \mid !m(x)\,.\,R)$,
- $\Gamma, m : T^{\mathsf{w}} \vdash P$,
- $\Gamma, m : T^{\mathsf{w}}, x : T \vdash R$,
- $m : T^{\mathsf{b}} \not\vdash \overline{m}\, m\,.\,\mathbf{0}$.

Then $Q \mid (\boldsymbol{\nu} m : T^{\mathsf{b}})\,(!P \mid !m(x)\,.\,R) \;\dot{\sim}_\Gamma\; Q \mid !\Big((\boldsymbol{\nu} m : T^{\mathsf{b}})\,(P \mid !m(x)\,.\,R)\Big)$.

Theorem 10.5 (distribution of a private replication over parallel composition) *Suppose that*

- $\Gamma \vdash (\boldsymbol{\nu} m : T^{\mathsf{b}})\,(P_1 \mid P_2 \mid !m(x)\,.\,R)$,
- $\Gamma, m : T^{\mathsf{w}} \vdash P_1 \mid P_2$,
- $\Gamma, m : T^{\mathsf{w}}, x : T \vdash R$,
- $m : T^{\mathsf{b}} \not\vdash \overline{m}\, m\,.\,\mathbf{0}$.

Then $(\boldsymbol{\nu} m : T^{\mathsf{b}})\,(P_1 \mid P_2 \mid !m(x)\,.\,R) \sim_\Gamma$
$(\boldsymbol{\nu} m : T^{\mathsf{b}})\,(P_1 \mid !m(x)\,.\,R) \mid (\boldsymbol{\nu} m : T^{\mathsf{b}})\,(P_2 \mid !m(x)\,.\,R)$

Proof: By Lemma 10.3 and the Context Lemma 9.12. □

Theorem 10.6 (distribution of a private replication over replication) *Suppose that*

- $\Gamma \vdash (\boldsymbol{\nu} m : T^{\mathsf{b}})\,(!P \mid !m(x)\,.\,R)$,
- $\Gamma, m : T^{\mathsf{w}} \vdash P$,
- $\Gamma, m : T^{\mathsf{w}}, x : T \vdash R$,
- $m : T^{\mathsf{b}} \not\vdash \overline{m}\, m\,.\,\mathbf{0}$.

Then $(\boldsymbol{\nu} m : T^{\mathsf{b}})\,(!P \mid !m(x)\,.\,R) \sim_\Gamma !\Big((\boldsymbol{\nu} m : T^{\mathsf{b}})\,(P \mid !m(x)\,.\,R)\Big)$.

Proof: By Lemma 10.4 and the Context Lemma 9.12. □

11 Derived process and type expressions

The following expressions can be coded up in the basic calculus:

1. Process definitions. A defining equation has the form

$$K \stackrel{\mathrm{def}}{=} (x_1 : T_1 . . x_n : T_n)P,$$

   where K belongs to some new alphabet of process identifiers, and can be thought of as a procedure declaration with formal parameters $x_1 . . x_n$; to use the definition with actual parameters $p_1 . . p_n$ we write $K\langle p_1 . . p_n\rangle$. For our purposes it suffices to assume that the parameters of constant definitions are names.

2. Binary inputs, like $p(x_1, x_2)\,.\,P$, and outputs, like $\overline{p}\,\langle v_1, v_2\rangle\,.\,P$ and $\overline{p}\,\ell\_\mathtt{upd}\_\langle v_1, v_2\rangle\,.\,P$.

3. Variant inputs like $p\big[_{j\in 1..n}\ \ell_j\_\big[_{i\in 1..m}\ell_{i,j}\_(\widetilde{x_{i,j}})\ \triangleright\ P_{i,j}\big]\big]$, where $\widetilde{x_{i,j}}$ can be a single name or a pair of names. This abbreviation allows us to go down two levels into the structure of a variant value received in an input at p; in fact, this term interacts with output particles of the form $\overline{p}\,\ell_r\_\ell_{s,r}\_\widetilde{w}$ (with $r \in 1 . . n$, $s \in 1 . . m$, and tuple $\widetilde{w}$ of the same length as $\widetilde{x_{r,s}}$) and, in doing so, it reduces to $P_{r,s}\{\widetilde{w}/\widetilde{x_{r,s}}\}$.

Each of these derived constructs has an associated derived typing rule, of the expected shape. The reader who wants to see the "macro expansions" of these expressions, and the associated derived rules for typing, subtyping and reduction can find them below; otherwise the reader may safely move to Section 12.

Abbreviations for the `case` construct

- An expression $z\,[\,\ell_1\_(y_1)\ \triangleright\ P_1\,..\,\ell_n\_(y_n)\ \triangleright\ P_n\,]$ expands to

  $$z(x)\,.\,\texttt{case}\ x\ \texttt{of}\ [\,\ell_1\_(y_1)\ \triangleright\ P_1\ ;\ ..\ ;\ \ell_n\_(y_n)\ \triangleright\ P_n\,]$$

  with x not free in $P_1\,..\,P_n$.

- An expression $\ell\_[\,\ell_1\_(y_1)\ \triangleright\ P_1\ ;\ ..\ ;\ \ell_n\_(y_n)\ \triangleright\ P_n\,]$ expands to

  $$\ell\_(x)\ \triangleright\ \Big(\texttt{case}\ x\ \texttt{of}\ [\,\ell_1\_(y_1)\ \triangleright\ P_1\ ;\ ..\ ;\ \ell_n\_(y_n)\ \triangleright\ P_n\,]\Big)$$

  with x not free in $P_1\,..\,P_n$.

Passing of pairs

We shall need communication of pairs of values, of the form

$$\overline{p}\,\tilde{\mathtt{h}}\_\langle v_1, v_2\rangle \tag{1}$$

where $\tilde{\mathtt{h}}$ represents a sequence of variant tags, like $\mathtt{h}_1\_\,..\,\_\mathtt{h}_n$. Symmetrically, and we shall need a pair destructor

$$\mathsf{let}\ (x_1, x_2) = z\ \mathsf{in}\ P \tag{2}$$

These two expressions have the obvious meaning. We now explain how to code them. A pair type $\langle T_1, T_2\rangle$ is translated into the type $[\,\ell_1\_T_1\ ;\ \ell_2\_T_2\,]$ where ℓ_1 and ℓ_2 are some tags chosen by convention. (Note that this translation schema does not justify the subtyping rule for tuples, because it does not force the same length of tuples. However, this can easily be accommodated by adding a dummy final component in the variant target type; for our purposes, the simple translation above will suffice because pair types will be compared with pair types only.) The encoding of (1) and (2) is this:

$$\begin{array}{rcl}
\overline{p}\,\tilde{\mathtt{h}}_\langle v_1, v_2\rangle\,.\,P & \stackrel{\mathrm{def}}{=} & (\boldsymbol{\nu} z : T)\,\overline{p}\,\tilde{\mathtt{h}}_z\,.\,\overline{z}\,\ell_1_v_1\,.\,\overline{z}\,\ell_2_v_2\,.\,P \\
\mathsf{let}\ (x_1, x_2) = z\ \mathsf{in}\ P & \stackrel{\mathrm{def}}{=} & z\Big[\ell_1_(x_1)\ \triangleright\ z[\ell_1_\star\,;\ \ell_2_(x_2)\ \triangleright\ P]\ ;\ \ell_2_\star\Big]
\end{array}$$

where

- if $v_1 : T_1$ and $v_2 : T_2$ then $T \stackrel{\mathrm{def}}{=} [\,\ell_1\_T_1\ ;\ \ell_2\_T_2\,]$;
- $\ell_i\_\star$ means that the expression at ℓ_i is unimportant.

In the same way, we can code polyadic communications of arbitrary length. This is similar to the encoding of the polyadic π-calculus into the monadic one [Mil91]. Having variants, allows us to have the translation correct on types (in the encoding in [Mil91] type information is lost).

Abbreviations for the `let` construct

1. An expression $p(x_1, x_2)\,.\,P$ expands to $p(y)\,.\,\mathsf{let}\ (x_1, x_2) = y\ \mathsf{in}\ P$, with y not free in P.

2. An expression $\ell\_(u_1, u_2)\ \triangleright\ P$ expands to $\ell\_(y)\ \triangleright\ \big(\mathsf{let}\ (u_1, u_2) = y\ \mathsf{in}\ P\big)$, with y not free in P.

Recursion

Recursive definitions can be defined in terms of replication in the usual way (see Milner [Mil91]).

Derived rules for typing, subtyping and reduction

We are now ready to show the derived typing and reduction rules for the process constructs introduced at the beginning of this section and the derived subtyping rule for the type construct of pairing. The derived subtyping rule is

$$\frac{\Sigma \vdash S_1 \leq T_1 \qquad \Sigma \vdash S_2 \leq T_2}{\Sigma \vdash \langle S_1, S_2 \rangle \leq \langle T_1, T_2 \rangle} \qquad \text{(A-Pair)}$$

Now the derived typing rules. In rule T-InpCase below, if $\widetilde{x}$ is a single name, say x, then $\widetilde{x} : T$ means $x : T$. If $\widetilde{x}$ is a pair of names, say x_1 and x_2, then $\widetilde{x} : T$ means that T is a pair type $\langle T_1, T_2 \rangle$ and that $x_1 : T_1, x_2 : T_2$.

$$\frac{\Gamma \vdash v_1 : T_1 \qquad \Gamma \vdash v_2 : T_2}{\Gamma \vdash \langle v_1, v_2 \rangle : \langle T_1, T_2 \rangle} \qquad \text{(Tv-pair)}$$

$$\frac{\Gamma \vdash p : \langle T_1, T_2 \rangle^{\mathsf{r}} \qquad \Gamma, x_1 : T_1, x_2 : T_2 \vdash P}{\Gamma \vdash p(x_1, x_2) \,.\, P} \qquad \text{(T-InpPair)}$$

$$\frac{\begin{array}{l}\Gamma \vdash p : \left[_{j \in 1..n} \ell_j\_\left[_{i \in 1..m} \ell_{i,j}\_T_{i,j}\right]\right]^{\mathsf{r}} \\ \text{for each } i \in 1..m \text{ and } j \in 1..n \;\; \Gamma, \widetilde{x_{i,j}} : T_{i,j} \vdash P_{i,j}\end{array}}{\Gamma \vdash p\left[_{j \in 1..n}\; \ell_j\_\left[_{i \in 1..m} \ell_{i,j}\_(\widetilde{x_{i,j}}) \triangleright P_{i,j}\right]\right]} \qquad \text{(T-InpCase)}$$

$$\frac{\text{for each } i \in 1..n \;\; \Gamma \vdash p_i : T_i \qquad K \stackrel{\text{def}}{=} (\, x_1 : T_1..x_n : T_n \,)P}{\Gamma \vdash K\langle p_1..p_n \rangle} \qquad \text{(T-Const)}$$

Each constant definition must be well-typed:

Definition 11.1 *A constant* $K \stackrel{\text{def}}{=} (\, x_1 : T_1..x_n : T_n \,)P$ *is* well-typed *if* $x_1 : T_1..x_n : T_n \vdash P$ *(i.e., the body of its definition is well-typed according to the information specified in its parameters).*

Finally, the derived reduction rules:

$$\frac{}{\overline{p}\,\langle q, r \rangle \,.\, P \mid p(x, y) \,.\, Q \longrightarrow\!\!\!\!\longrightarrow^{\star}_{\mathtt{d}} P \mid Q\{^{q,\,r}\!/_{x,\,y}\}} \qquad \text{(R-Pair)}$$

$$\frac{r \in 1..n \qquad s \in 1..m}{\overline{p}\,\ell_r\_\ell_{s,r}\_\widetilde{v} \,.\, P \mid p\left[_{j \in 1..n}\; \ell_j\_\left[_{i \in 1..m} \ell_{i,j}\_(\widetilde{x_{i,j}}) \triangleright Q_{i,j}\right]\right] \longrightarrow\!\!\!\!\longrightarrow^{\star}_{\mathtt{d}} P \mid Q_{s,r}\{^{\widetilde{v}}\!/_{\widetilde{x_{s,r}}}\}} \qquad \text{(R-CC)}$$

$$\frac{K \stackrel{\text{def}}{=} (\, x_1 : T_1..x_n : T_n \,)P}{K\langle p_1..p_n \rangle \longrightarrow\!\!\!\!\longrightarrow^{\star}_{\mathtt{d}} P\{^{p_1..p_n}\!/_{x_1..x_n}\}} \qquad \text{(R-Rec)}$$

12 The interpretation of the typed OC

The translations of types, type environments and terms are defined structurally using the rules in the tables below. We assume that `sel` and `upd` are variant tags.

The translation of types

$$[\![\{_{j \in 1..n}\;\ell_j : B_j\}]\!] \stackrel{\text{def}}{=} \mu X.\left[_{j \in 1..n} \ell_j\_\left[\begin{array}{l}\mathtt{sel}\_[\![B_j]\!]^{\mathsf{ww}} \,; \\ \mathtt{upd}\_\langle X^{\mathsf{ww}}, \langle X^{\mathsf{w}}, [\![B_j]\!]^{\mathsf{ww}} \rangle^{\mathsf{w}} \rangle\end{array}\right]\right]$$

Only w tags appear in the translation of types because the translation of terms will respect the discipline that every name received in an input may be used only in output position. This, and the fact that every offer of an input is persistent, prevents non-deterministic reductions in the target processes of the translation.

The pattern of occurrences of w tags is determined by the protocol which implements select and update operations. What is important, however, is the level of nesting of w tags: An even number of nesting gives covariance, whereas an odd number of nesting gives contravariance. Thus, the component $[\![B_j]\!]$ is in covariant position on selection, and in contravariance position on update: This explains the invariance of object types on the common components, in rule O-SUBOB (the interpretation of OC into the λ-calculus [ACV96] does the same).

The location of the translation of an object of type A will actually have type $[\![A]\!]^{\mathsf{w}^{\mathsf{w}}}$; as far as subtyping is concerned, the two outermost w tags are irrelevant because they cancel one another.

The translation of type environments

$$
\begin{aligned}
[\![\emptyset]\!] &\stackrel{\text{def}}{=} \emptyset \\
[\![E, x : A]\!] &\stackrel{\text{def}}{=} [\![E]\!], x : [\![A]\!]^{\mathsf{w}}
\end{aligned}
$$

We now turn to the translation of terms. We can understand the new translation as a rectification on that in Section 3. Let us look back at that translation in Section 3. The core of the translation of an object value is the replicated process $P \stackrel{\text{def}}{=} !x(\ell, r, y) \,.\, (\prod_{j \in 1..n} [\ell = \ell_j][\![b_j]\!]_r)$, where x plays the role of the self parameter of the object. An update of the object is modeled as the creation of a new replicated process that works by forwarding messages for non-updated methods to P. If there are several operations of update, process P may end up receiving messages from several different replicated processes, each of which represents a different object. In this situation, when one of these replicated processes sends a message to P (i.e., it selects one of P's methods) the message must contain the self parameter, because the sender of the message cannot be determined statically. This exposure of the self-parameter as an argument of select requests is what makes the translation go wrong. To put it briefly, the problem of the translation in Section 3 is that (the translation of) an object value handles only operations of selection, whereas operations of update are handled at the place where the update is made. To avoid this, in the new translation an object value directly handles both select and update requests. This is achieved by separating, in the translation of object values, the method bodies and an object manager $\mathtt{OB}^A\langle s_1 \mathinner{..} s_n, x\rangle$. In the new translation, select and update requests do not mention the self parameter.

We explain the new translation in more detail. As in the previous translation, the first action of (the translation of) an object value is to signal its valuehood by providing an access x to its value-core. This value-core is administered by process $\mathtt{OB}^A\langle s_1 \mathinner{..} s_n, x\rangle$; this "owns" the object methods, in the sense that it is the only process which can reach them, via names s_i's. The manager $\mathtt{OB}^A\langle s_1 \mathinner{..} s_n, x\rangle$ can receive a request of select or update on any method ℓ_j, $1 \leq j \leq n$. In the former case, the request is of the form $\overline{x}\,\ell_j\_\mathtt{sel}\_p$, where p is the location for the new object. Using channel s_j, the manager activates a copy of the body of method ℓ_j; in doing so, it also supplies the pointer x to itself, which represents the self parameter. Thus the communication of the self parameter occurs in an *internal* action of the object and, as such, it does not affect its typing. In the case of update at ℓ_j, a request has the form $\overline{x}\,\ell_j\_\mathtt{upd}\_\langle p, s\rangle$ where p is the location for the new object and s a pointer to the new method body; in this case, the manager $\mathtt{OB}^A\langle s_1 \mathinner{\ldotp\ldotp} s_n, x\rangle$ spawns off the interface for a new object, which will be located at p and will use channel s, in place of s_j, to process any request for method ℓ_j. The functional nature of OC is reflected into the functional nature of the object manager $\mathtt{OB}^A\langle s_1 \mathinner{\ldotp\ldotp} s_n, x\rangle$ — it is a replicated process, hence it handles all requests in the same way.

The translations of OC into λ-calculus [ACV96] and into π-calculus have comparable lengths: In a few places, the π-calculus gains by having only first-order types, in other places it suffers by the lack of term substitutions in the syntax.

As in the interpretation into the λ-calculus, our translation of terms uses an environment E as parameter to put the necessary type annotations in the translation of method selection. This parameter could be avoided by having, for instance, more type information on the syntax of method selection. We assume that $p, q, r, s \ldots$ are not OC variables.

The translation of terms

$$\llbracket \{_{j\in 1..n}\ \ell_j = \zeta(x_j : A).\, b_j\} \rrbracket_p^E \stackrel{\text{def}}{=} (\boldsymbol{\nu} x : \llbracket A \rrbracket^{\mathsf{b}})\, \overline{p}\, x \,.\, (_{j\in 1..n}\, \boldsymbol{\nu}\, s_j : T^{\mathsf{b}}_{A,j}) \Big(\mathtt{OB}^A\langle s_1 \mathinner{\ldotp\ldotp} s_n, x\rangle \mid \Pi_{j\in 1..n} ! s_j(x_j, r_j) \,.\, \llbracket b_j \rrbracket_{r_j}^{E, x_j : A} \Big)$$

$$\llbracket a.\, \ell_j \rrbracket_p^E \stackrel{\text{def}}{=} (\boldsymbol{\nu} q : \llbracket \{\, \ell_j : B_j \,\} \rrbracket^{\mathsf{wb}})\, (\llbracket a \rrbracket_q^E \mid q(x) \,.\, \overline{x}\, \ell_j\_\mathtt{sel}\_p)$$

$$\llbracket a.\, \ell_j \Leftarrow \zeta(x_j : A).\, b \rrbracket_p^E \stackrel{\text{def}}{=} (\boldsymbol{\nu} q : \llbracket A \rrbracket^{\mathsf{wb}}) \Big(\llbracket a \rrbracket_q^E \mid q(x) \,.\, (\boldsymbol{\nu} s : T^{\mathsf{b}}_{A,j})\, \overline{x}\, \ell_j\_\mathtt{upd}\_\langle p, s\rangle \,.\, !s(x_j, r_j) \,.\, \llbracket b \rrbracket_{r_j}^{E, x_j : A} \Big)$$

$$\llbracket x \rrbracket_p^E \stackrel{\text{def}}{=} \overline{p}\, x$$

where the process identifier $\mathtt{OB}^A$ is

$$\begin{aligned} \mathtt{OB}^A \stackrel{\text{def}}{=}\ & (\, s_1 : T^{\mathsf{w}}_{A,1} \mathinner{\ldotp\ldotp} s_n : T^{\mathsf{w}}_{A,n}, x : \llbracket A \rrbracket^{\mathsf{b}}\,) \\ & !x \Big[_{j\in 1..n}\ \ell_j\_\big[\ \mathtt{sel}\_(r_j) \rhd \overline{s_j}\, \langle x, r_j\rangle\ ; \\ & \qquad \mathtt{upd}\_(r, s) \rhd (\boldsymbol{\nu} x_{\mathtt{new}} : \llbracket A \rrbracket^{\mathsf{b}}) \\ & \qquad\qquad \overline{r}\, x_{\mathtt{new}} \,.\, \mathtt{OB}^A\langle s_1 \mathinner{\ldotp\ldotp} s_{j-1}, s, s_{j+1} \mathinner{\ldotp\ldotp} s_n, x_{\mathtt{new}}\rangle\ \big]\ \Big] \end{aligned}$$

and where

- $A \stackrel{\text{def}}{=} \{_{j\in 1..n}\ \ell_j : B_j\}$;
- $T_{A,j} \stackrel{\text{def}}{=} \langle \llbracket A \rrbracket^{\mathsf{w}}, \llbracket B_j \rrbracket^{\mathsf{ww}} \rangle$;
- in the encoding of selection, B_j is the unique type s.t. $E \vdash a : [\ldots, \ell_j : B_j, \ldots]$ holds, if one such judgement exists (the unicity of this type is a consequence of the minimum-type property of OC, Lemma 2.5), B_j can be any type otherwise;
- in the rule for value, x is not free in $b_1 \mathinner{\ldotp\ldotp} b_n$; in the rule for update, x is not free in b.

For using the translation, it is useful to define the following abbreviation, for a value $a \stackrel{\text{def}}{=} \{_{j\in 1..n}\ \ell_j = \zeta(x_j : A).\, b_j\}$:

$$\llbracket x := a \rrbracket^E \stackrel{\text{def}}{=} (_{j\in 1..n}\, \boldsymbol{\nu}\, s_j : T^{\mathsf{b}}_{A,j}) \Big(\mathtt{OB}^A\langle s_1 \mathinner{\ldotp\ldotp} s_n, x\rangle \mid \Pi_{j\in 1..n} ! s_j(x_j, r_j) \,.\, \llbracket b_j \rrbracket_{r_j}^{E, x_j : A} \Big).$$

Then

$$[\![a]\!]_p^E = (\boldsymbol{\nu} x : [\![A]\!]^{\mathsf{b}})\,\overline{p}\,x\,.\,[\![x := a]\!]^E.$$

Omitting type information, if $i \in 1..n$, a select operation $a.\,\ell_i \longrightarrow b_i\{a/x_i\}$ and an update operation

$$a.\,\ell_i \Leftarrow \varsigma(y)b \longrightarrow \{_{j\in\{1..n\}-\{i\}}\ \ell_j = \varsigma(x_j).\,b_j,\ \ell_i = \varsigma(x).\,b\} \stackrel{\text{def}}{=} a'$$

are simulated thus:

$$\begin{array}{rcll}
[\![a.\,\ell_i]\!]_p & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} q\,)\Big((\boldsymbol{\nu} x\,)\overline{q}\,x\,.\,[\![x := a]\!] \mid q(x)\,.\,\overline{x}\,\ell_i\texttt{_sel_}p\Big) & \\
& \longrightarrow_{\mathsf{d}} & (\boldsymbol{\nu} x\,)([\![x := a]\!] \mid \overline{x}\,\ell_i\texttt{_sel_}p\,) & \\
& \longrightarrow_{\mathsf{d}}^{\star} & (\boldsymbol{\nu} x\,)([\![x := a]\!] \mid [\![b_i\{x/x_i\}]\!]_p) & \\
& \sim & [\![b_i\{a/x_i\}]\!]_p & (3)
\end{array}$$

$$\begin{array}{rcl}
[\![a.\,\ell_i \Leftarrow \varsigma(y)b]\!]_p & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} q\,)\Big((\boldsymbol{\nu} x\,)\overline{q}\,x\,.\,[\![x := a]\!] \mid \\
& & \qquad q(x)\,.\,(\boldsymbol{\nu} s\,)\overline{x}\,\ell_i\texttt{_upd_}\langle p, s\rangle\,.\,!s(y,r)\,.\,[\![b]\!]_r\Big) \\
& \longrightarrow_{\mathsf{d}} & (\boldsymbol{\nu} x, s\,)([\![x := a]\!] \mid \overline{x}\,\ell_i\texttt{_upd_}\langle p, s\rangle\,.\,!s(y,r)\,.\,[\![b]\!]_r) \\
& \longrightarrow_{\mathsf{d}}^{\star} & (\boldsymbol{\nu} x, s\,)(_{j\in 1..n}\,\boldsymbol{\nu}\,s_j\,) \\
& & \Big(\,\texttt{OB}\langle s_1..s_n, x\rangle \mid \Pi_{j\in 1..n}!s_j(x_j, r_j)\,.\,[\![b_j]\!]_{r_j} \\
& & \quad \mid (\boldsymbol{\nu} x_{\texttt{new}}\,)\overline{p}\,x_{\texttt{new}}\,.\,\texttt{OB}\langle s_1..s_{j-1}, s, s_{j+1}..s_n, x_{\texttt{new}}\rangle \\
& & \quad \mid !s(y,r)\,.\,[\![b]\!]_r\Big) \\
& \sim & (\boldsymbol{\nu} s\,)(_{j\in\{1..n\}-\{i\}}\,\boldsymbol{\nu}\,s_j\,) \\
& & \Big(\Pi_{j\in\{1..n\}-\{i\}}!s_j(x_j, r_j)\,.\,[\![b_j]\!]_{r_j} \mid !s(y,r)\,.\,[\![b]\!]_r \\
& & \quad \mid (\boldsymbol{\nu} x_{\texttt{new}}\,)\overline{p}\,x_{\texttt{new}}\,.\,\texttt{OB}\langle s_1..s_{j-1}, s, s_{j+1}..s_n, x_{\texttt{new}}\rangle\Big) \\
& \equiv & (\boldsymbol{\nu} x_{\texttt{new}}\,)\overline{p}\,x_{\texttt{new}}\,.\,[\![x_{\texttt{new}} := a']\!]\Big) \\
& = & [\![a']\!]
\end{array}$$

Above, $\sim$ stands for strong barbed congruence. All appearances of $\sim$ except (3) represent garbage collection steps where deadlocked processes and restrictions binding nothing are removed. Equality (3) is proved in Lemma 14.1 and uses the algebraic laws of Section 10, in particular the distribution theorems for private replications.

13 Type correctness

In this section, we establish the correctness of the translation at the level of types: The translation validates the typing and subtyping rules of OC.

Lemma 13.1 *If $A \leq B$ and $E, x : B \vdash a : B_j$, then $[\![a]\!]_p^{E,x:B} = [\![a]\!]_p^{E,x:A}$.*

Lemma 13.2 *If $E \vdash a : A$ and x does not appear in E, then for all B we have $[\![a]\!]_p^E = [\![a]\!]_p^{E,x:B}$.*

Theorem 13.3 (correctness for subtyping) *For all A, B, it holds that $A \leq B$ iff $[\![A]\!]^{\mathsf{w}} \leq [\![B]\!]^{\mathsf{w}}$.*

Proof: By hypothesis $A \leq B$ hence by rule O-SubOb, it must be

$$A = \{_{j\in 1..n+m}\ \ell_j : B_j\} \quad \text{and} \quad B = \{_{j\in 1..n}\ \ell_j : B_j\}.$$

By rule A-XO, proving $\emptyset \vdash [\![A]\!]^{\mathsf{w}} \leq [\![B]\!]^{\mathsf{w}}$ reduces to proving $\emptyset \vdash [\![B]\!] \leq [\![A]\!]$ and then, by the rules for recursive types, it reduces to proving

$$\begin{array}{lll}[\![B]\!] \leq [\![A]\!] & \vdash & \left[_{j\in 1..n} \ell_j\_\left[\,\texttt{sel}\_[\![B_j]\!]^{\mathsf{ww}};\ \texttt{upd}\_\langle [\![B]\!]^{\mathsf{ww}}, \langle [\![B]\!]^{\mathsf{w}}, [\![B_j]\!]^{\mathsf{ww}}\rangle^{\mathsf{w}}\rangle\ \right]\ \right] \\ & & \leq \\ & & \left[_{j\in 1..n+m} \ell_j\_\left[\,\texttt{sel}\_[\![B_j]\!]^{\mathsf{ww}};\ \texttt{upd}\_\langle [\![A]\!]^{\mathsf{ww}}, \langle [\![A]\!]^{\mathsf{w}}, [\![B_j]\!]^{\mathsf{ww}}\rangle^{\mathsf{w}}\rangle\ \right]\ \right]\end{array}$$

This, by rule A-Case, gives rise to the following goals, for $j \in 1..n$:

$$\begin{array}{lll}[\![B]\!] \leq [\![A]\!] & \vdash & [\![B_j]\!]^{\mathsf{ww}} \leq [\![B_j]\!]^{\mathsf{ww}} \\ [\![B]\!] \leq [\![A]\!] & \vdash & \langle [\![B]\!]^{\mathsf{ww}}, \langle [\![B]\!]^{\mathsf{w}}, [\![B_j]\!]^{\mathsf{ww}}\rangle^{\mathsf{w}}\rangle \leq \langle [\![A]\!]^{\mathsf{ww}}, \langle [\![A]\!]^{\mathsf{w}}, [\![B_j]\!]^{\mathsf{ww}}\rangle^{\mathsf{w}}\rangle\end{array}$$

The first of these goals is validated by reflexivity. The second goal, by rule A-Pair and A-XO, gives rise to the subgoals:

$$\begin{array}{lll}[\![B]\!] \leq [\![A]\!] & \vdash & [\![B]\!]^{\mathsf{ww}} \leq [\![A]\!]^{\mathsf{ww}} \\ [\![B]\!] \leq [\![A]\!] & \vdash & [\![A]\!]^{\mathsf{w}} \leq [\![B]\!]^{\mathsf{w}} \\ [\![B]\!] \leq [\![A]\!] & \vdash & [\![B_j]\!]^{\mathsf{ww}} \leq [\![B_j]\!]^{\mathsf{ww}}\end{array}$$

which are simple to validate.

Now the opposite implication. We suppose $\emptyset \vdash [\![A]\!]^{\mathsf{w}} \leq [\![B]\!]^{\mathsf{w}}$ and prove $A \leq B$ by induction on the (sum of the) lengths of A and B. Assume

$$A = \{_{j\in 1..m}\ \ell_j : A_j\} \quad \text{and} \quad B = \{_{j\in 1..n}\ \mathtt{h}_j : B_j\}.$$

Then, if

$$S_{A,j} \stackrel{\text{def}}{=} \left[\,\texttt{sel}\_[\![A_j]\!]^{\mathsf{ww}};\ \texttt{upd}\_\langle X^{\mathsf{ww}}, \langle X^{\mathsf{w}}, [\![A_j]\!]^{\mathsf{ww}}\rangle^{\mathsf{w}}\rangle\ \right]$$

and

$$S_{B,j} \stackrel{\text{def}}{=} \left[\,\texttt{sel}\_[\![B_j]\!]^{\mathsf{ww}};\ \texttt{upd}\_\langle X^{\mathsf{ww}}, \langle X^{\mathsf{w}}, [\![B_j]\!]^{\mathsf{ww}}\rangle^{\mathsf{w}}\rangle\ \right]$$

we have

$$[\![A]\!] = \mu X. \left[_{j\in 1..m} \ell_j\_S_{A,j}\right]$$

and

$$[\![B]\!] = \mu X. \left[_{j\in 1..n} \mathtt{h}_j\_S_{B,j}\right]$$

When started with the goal $\emptyset \vdash [\![A]\!]^{\mathsf{w}} \leq [\![B]\!]^{\mathsf{w}}$, the subtyping rules generate the goal $\emptyset \vdash [\![B]\!] \leq [\![A]\!]$, and then (omitting pairs equivalent to $[\![B]\!] \leq [\![A]\!]$ in the assumption) the goal:

$$[\![B]\!] \leq [\![A]\!] \vdash \left[_{j\in 1..n} \mathtt{h}_j\_\left(S_{B,j}\{[\![B]\!]/X\}\right)\right] \leq \left[_{j\in 1..m} \ell_j\_\left(S_{A,j}\{[\![A]\!]/X\}\right)\right]$$

From this, by rule T-Case, we infer

$$n \leq m \text{ and } \mathtt{h}_j = \ell_j \ \ (j \in 1..n) \tag{4}$$

and we reduce ourselves to the goals, for $j \in 1..n$:

$$[\![B]\!] \leq [\![A]\!] \quad \vdash \quad S_{B,j}\{[\![B]\!]/X\} \leq S_{A,j}\{[\![A]\!]/X\}$$

By T-Case again, for each j we get the two goals

$$[\![B]\!] \leq [\![A]\!] \quad \vdash \quad [\![B_j]\!]^{\mathsf{ww}} \leq [\![A_j]\!]^{\mathsf{ww}} \tag{5}$$

$$[\![B]\!] \leq [\![A]\!] \quad \vdash \quad \langle [\![B]\!]^{\mathsf{ww}}, \langle [\![B]\!]^{\mathsf{w}}, [\![B_j]\!]^{\mathsf{ww}}\rangle^{\mathsf{w}}\rangle \leq \langle [\![A]\!]^{\mathsf{ww}}, \langle [\![A]\!]^{\mathsf{w}}, [\![A_j]\!]^{\mathsf{ww}}\rangle^{\mathsf{w}}\rangle \tag{6}$$

From (5), since $\emptyset \vdash [\![B]\!] \leq [\![A]\!]$ holds, applying Lemma 6.3 we infer

$$\emptyset \vdash [\![B_j]\!]^{\mathsf{ww}} \leq [\![A_j]\!]^{\mathsf{ww}}$$

from which we get

$$\emptyset \vdash [\![A_j]\!]^{\mathsf{w}} \leq [\![B_j]\!]^{\mathsf{w}}$$

and then, by induction,

$$A_j \leq B_j.$$

Continuing from (6), similarly we infer

$$B_j \leq A_j$$

Summarising, we have proved that $[\![A]\!]^{\mathsf{w}} \leq [\![B]\!]^{\mathsf{w}}$ implies $n \leq m$ and, for all $j \in 1..n$, $\mathtt{h}_j = \ell_j$ and $A_j \leq B_j \leq A_j$, that is, by Lemma 2.1, $A_j = B_j$. Therefore we can use rule O-subOb, to infer $A \leq B$. $\square$

Theorem 13.4 (completeness of the interpretation on type judgements) *If $E \vdash a : A$ then, for all p, it holds that $[\![E]\!], p : [\![A]\!]^{\mathsf{ww}} \vdash [\![a]\!]^E_p$.*

Proof: By induction on the length of $E \vdash a : A$. We only report two cases; recall however that in the case of rule OT-obj one has also to check the well-typedness of the definition of $\mathtt{OB}^A$.

OT-upd The rule applied is

$$\frac{E \vdash a : A \qquad E, x_i : A \vdash b : B_i \qquad A = \{_{j \in 1..n}\ \ell_j : B_j\} \qquad i \in 1..n}{E \vdash a.\,\ell_i \Leftarrow \zeta(x_i : A).\,b : A}$$

By induction,

$$\text{for all } r, \quad [\![E]\!], r : [\![A]\!]^{\mathsf{ww}} \vdash [\![a]\!]^E_r \tag{7}$$

$$\text{for all } r, \quad [\![E]\!], x_i : [\![A]\!]^{\mathsf{w}}, r : [\![B_i]\!]^{\mathsf{ww}} \vdash [\![b]\!]^{E, x_i : A}_r \tag{8}$$

The thesis to prove is that

$$\begin{aligned}&[\![E]\!], p : [\![A]\!]^{\mathsf{ww}} \vdash \\ &(\boldsymbol{\nu} q : [\![A]\!]^{\mathsf{wb}}) \Big([\![a]\!]^E_q \mid q(x)\,.\,(\boldsymbol{\nu} s : T^{\mathsf{b}}_{A,i})\, \overline{x}\, \ell_i\_\mathtt{upd}\_\langle p, s\rangle\,.\, !s(x_i, r_i)\,.\,[\![b]\!]^{E, x_i : A}_{r_i} \Big)\end{aligned} \tag{9}$$

where $T_{A,i} = \langle [\![A]\!]^{\mathsf{w}}, [\![B_i]\!]^{\mathsf{ww}}\rangle$.

By definition of typing, weakening and narrowing, we can prove (9) from the following equations:

$$[\![E]\!], q : [\![A]\!]^{\mathsf{wb}} \vdash [\![a]\!]^E_q \tag{10}$$

$$p : [\![A]\!]^{\mathsf{ww}}, x : [\![A]\!]^{\mathsf{w}}, s : \langle [\![A]\!]^{\mathsf{w}}, [\![B_i]\!]^{\mathsf{ww}}\rangle^{\mathsf{b}} \vdash \overline{x}\, \ell_i\_\mathtt{upd}\_\langle p, s\rangle \tag{11}$$

$$[\![E]\!], s : \langle [\![A]\!]^{\mathsf{w}}, [\![B_i]\!]^{\mathsf{ww}}\rangle^{\mathsf{r}} \vdash s(x_i, r_i)\,.\,[\![b]\!]^{E, x_i : A}_{r_i} \tag{12}$$

Now, equation (10) follows from (7) and narrowing; (12) follows from

$$[\![E]\!], x_i : [\![A]\!]^{\mathsf{w}}, r_i : [\![B_i]\!]^{\mathsf{ww}} \vdash [\![b]\!]^{E, x_i : A}_{r_i}$$

which is true by (8). Finally, (11) follows from

$$p : [\![A]\!]^{\mathsf{ww}}, s : \langle [\![A]\!]^{\mathsf{w}}, [\![B_i]\!]^{\mathsf{ww}} \rangle^{\mathsf{b}} \vdash \ell_i\_\mathtt{upd}\_\langle p, s\rangle : [\![A]\!]$$

which is inferred using subsumption from

$$p : [\![A]\!]^{\mathsf{ww}}, s : \langle [\![A]\!]^{\mathsf{w}}, [\![B_i]\!]^{\mathsf{ww}} \rangle^{\mathsf{b}} \vdash \ell_i\_\mathtt{upd}\_\langle p, s\rangle : [\,\ell_i\_[\,\mathtt{upd}\_\langle [\![A]\!]^{\mathsf{ww}}, \langle [\![A]\!]^{\mathsf{w}}, [\![B_i]\!]^{\mathsf{ww}} \rangle^{\mathsf{b}} \rangle\,]\,]$$

and

$$\begin{array}{rcl} [\,\ell_i\_[\,\mathtt{upd}\_\langle [\![A]\!]^{\mathsf{ww}}, \langle [\![A]\!]^{\mathsf{w}}, [\![B_i]\!]^{\mathsf{ww}} \rangle^{\mathsf{b}} \rangle\,]\,] & \leq & [\,\ell_i\_[\,\mathtt{upd}\_\langle [\![A]\!]^{\mathsf{ww}}, \langle [\![A]\!]^{\mathsf{w}}, [\![B_i]\!]^{\mathsf{ww}} \rangle^{\mathsf{w}} \rangle\,]\,] \\ & \leq & [\![A]\!] \end{array}$$

exploiting the definitions of A and of $[\![A]\!]$ and the rules of typing for values (including TV-SUB), Lemma 6.4 and transitivity of $\leq$.

OT-SUBS The rule applied is

$$\frac{E \vdash a : A \qquad A \leq B}{E \vdash a : B}$$

By induction,

$$\text{for all } r, \quad [\![E]\!], r : [\![A]\!]^{\mathsf{ww}} \vdash [\![a]\!]^{E}_{r}\ . \tag{13}$$

By Theorem 13.3, $[\![A]\!]^{\mathsf{w}} \leq [\![B]\!]^{\mathsf{w}}$, hence

$$[\![B]\!]^{\mathsf{ww}} \leq [\![A]\!]^{\mathsf{ww}}. \tag{14}$$

Now,

$$[\![E]\!], p : [\![B]\!]^{\mathsf{ww}} \vdash [\![a]\!]^{E}_{p}$$

follows from (13) and (14) by narrowing.

□

Lemma 13.5 *If* $[\![E]\!], p : [\![A]\!]^{\mathsf{wb}} \vdash [\![a]\!]^{E}_{p}$ *, then also* $[\![E]\!], p : [\![A]\!]^{\mathsf{ww}} \vdash [\![a]\!]^{E}_{p}$ *.*

Proof: Easy case inspection on the definition of the encoding. □

Theorem 13.6 (soundness of the translation on type judgements)
If $[\![E]\!], p : [\![A]\!]^{\mathsf{ww}} \vdash [\![a]\!]^{E}_{p}$ *, then* $E \vdash a : A$ *.*

Proof: By induction on the structure of a. First, suppose a is a variable x. By hypothesis

$$[\![E]\!], p : [\![A]\!]^{\mathsf{ww}} \vdash \overline{p}\, x$$

hence x must appear in E. This means that, for some B, $[\![E]\!](x) = [\![B]\!]^{\mathsf{w}}$. By Lemma 8.11, $[\![B]\!]^{\mathsf{w}} \leq [\![A]\!]^{\mathsf{w}}$; hence by Theorem 13.3, $B \leq A$. Therefore $E \vdash x : A$ is derivable, using the rules for variables and subsumption.
Suppose a is a value, say $\{_{j \in 1..n}\ \ell_j = \zeta(x_j : B).\, b_j\}$, with $B = \{_{j \in 1..n}\ \ell_j : B_j\}$, and

$$[\![E]\!], p : [\![A]\!]^{\mathsf{ww}} \vdash [\![a]\!]^{E}_{p}\ . \tag{15}$$

From this and the definition of the encoding, for some R,

$$[\![E]\!], p : [\![A]\!]^{\mathsf{ww}}, x : [\![B]\!]^{\mathsf{b}} \vdash \overline{p}\, x\, .\, R$$

For this to hold, by Lemma 8.11 it must be $[\![B]\!]^{\mathsf{b}} \leq [\![A]\!]^{\mathsf{w}}$, which implies $[\![A]\!] \leq [\![B]\!]$. By Theorem 13.3, this implies

$$B \leq A. \tag{16}$$

From (15), by definition of the encoding and contraction, we infer, for all $j \in 1..n$,

$$[\![E]\!], s_j : T^{\mathsf{b}}_{B,j} \vdash !s_j(x_j, r_j) \,.\, [\![b_j]\!]^{E, x_j : B}_{r_j}$$

from which we get

$$[\![E]\!], x_j : [\![B]\!]^{\mathsf{w}}, r_j : [\![B_j]\!]^{\mathsf{ww}} \vdash [\![b_j]\!]^{E, x_j : B}_{r_j} \,.$$

This, by the inductive hypothesis, gives us

$$E, x_j : B \vdash b_j : B_j \,. \tag{17}$$

Finally, (16) and (17) allow us to use rule OT-OBJ and subsumption to infer $E \vdash a : A$.
The case $a = a'.\,\ell_i \Leftarrow \zeta(x_i : A).\, b$ is similar.
Finally, the case when $a = a'.\,\ell_i$. By hypothesis,

$$[\![E]\!], p : [\![B_i]\!]^{\mathsf{ww}} \vdash (\boldsymbol{\nu} q : [\![\{\, \ell_i : B_i \,\}]\!]^{\mathsf{wb}})\, ([\![a']\!]^E_q \mid q(x) \,.\, \overline{x}\, \ell_i\texttt{\_sel\_}p\,)$$

for some B_i. From this, Lemma 13.5 and contraction, we infer

$$[\![E]\!], q : [\![\{\, \ell_i : B_i \,\}]\!]^{\mathsf{ww}} \vdash [\![a']\!]^E_q$$

which by induction gives $E \vdash a' : \{\, \ell_i : B_i \,\}$. From this, by rule OR-SEL we infer $E \vdash a'.\,\ell_i : B_i$. □

14 Operational correspondence and adequacy

We show that our interpretation preserves the computational content of terms. In the proofs of this section, we omit type information when it is unimportant.

Lemma 14.1 *Suppose*

- $a \stackrel{\text{def}}{=} \{_{j\in 1..n}\ \ \ell_j = \zeta(x_j : A).\, b_j\}$,
- $E \vdash a : A$,
- $E, x : A \vdash b : B$,
- $\Gamma \stackrel{\text{def}}{=} [\![E]\!], p : [\![B]\!]^{\mathsf{ww}}$.

Then $(\boldsymbol{\nu} x : [\![A]\!]^{\mathsf{b}})\, ([\![b]\!]^{E,x:A}_p \mid [\![x := a]\!]^E) \sim_\Gamma [\![b\{^a\!/\!_x\}]\!]^E_p$.

Proof: Recall that $[\![x := a]\!]^E = (\boldsymbol{\nu} \tilde{s}\,)(\mathtt{OB}^A\langle \tilde{s}, x\rangle \mid \prod_j !s_j(x_j, r_j) \,.\, [\![b_j]\!]^{E, x_j : A}_{r_j})$. Moreover, using $E \vdash a : A$ and Theorem 13.4, we can infer

$$[\![E]\!], x : [\![A]\!]^{\mathsf{b}} \ \vdash\ [\![x := a]\!]^E$$

Hence, using the laws for replication in Section 10 and expanding the definitions of the abbreviations,

$$[\![x := a]\!]^E \sim_{[\![E]\!], x : [\![A]\!]^{\mathsf{b}}} \ \ !x(r) \,.\, P_a \tag{18}$$

for some process P_a with

$$[\![E]\!], r : [\![A]\!] \quad \vdash \quad P_a \tag{19}$$

$$x : [\![A]\!]^{\mathsf{b}} \quad \not\vdash \quad \overline{x}\, x \,.\, \mathbf{0} \tag{20}$$

Moreover, by Theorem 13.4, it holds that

$$[\![E]\!], x : [\![A]\!]^{\mathsf{w}}, p : [\![B]\!]^{\mathsf{ww}} \vdash [\![b]\!]_p^{E,x:A} \tag{21}$$

We prove the assertion of the lemma by induction on the structure of b. Facts (18-21) allow us to apply the distributivity laws for replication.

1. $b = x$.

   Use law **L1** of Lemma 10.2.

2. $b = y \neq x$

   Use law **L2** of Lemma 10.2.

3. $b = \{_{i \in 1..m}\ \mathtt{h}_i = \zeta(y_i : B').\, b'_i\}$

   We have:

   $$\begin{array}{l}(\boldsymbol{\nu} x\,)([\![b]\!]_p^{E,x:A} \mid [\![x := a]\!]^E) \quad = \\ (\boldsymbol{\nu} x\,)\Big((\boldsymbol{\nu} x'\,)(\overline{p}\, x' \,.\, (\boldsymbol{\nu} \tilde{s'}\,)(\mathtt{OB}^{B'}\langle \tilde{s'}, x'\rangle \mid \prod_i ! s'_i(y_i, r_i) \,.\, [\![b'_i]\!]_{r_i}^{E,y_i:B',x:A})) \mid \\ \hfill [\![x := a]\!]^E\Big)\end{array}$$

   from which, using the laws for replication and some garbage collection,

   $$\begin{array}{ll}\sim_\Gamma & (\boldsymbol{\nu} x'\,)\Big(\overline{p}\, x' \,.\, (\boldsymbol{\nu} \tilde{s'}\,)\Big(\mathtt{OB}^{B'}\langle \tilde{s'}, x'\rangle \mid \prod_i ! s'_i(y_i, r_i) \,. \\ & \hfill (\boldsymbol{\nu} x\,)([\![b'_i]\!]_{r_i}^{E,y_i:B',x:A} \mid [\![x := a]\!]^E)\Big)\Big)\end{array}$$

   By Lemma 13.2, $[\![x := a]\!]^E = [\![x := a]\!]^{E,y_i:B'}$. Then the thesis follows by the inductive assumption.

4. $b = b'.\,\ell_j$ or $b = b'.\,\ell_i \Leftarrow \zeta(y_i : B').\, b''$.

   Similar to above.

□

We recall that, in a case statement, $\ell\_\star$ means that the branch at ℓ is irrelevant. We write $\longrightarrow_{\mathtt{d}}^{\star} \sim_\Gamma$ for the composition of the two relations $\longrightarrow_{\mathtt{d}}^{\star}$ and $\sim_\Gamma$.

Lemma 14.2 *Let*

- $a \stackrel{\text{def}}{=} \{_{j \in 1..n}\ \ell_j = \zeta(x_j : A).\, b_j\}$,
- $A \stackrel{\text{def}}{=} \{_{j \in 1..n}\ \ell_j : A_j\}$,
- $\emptyset \vdash a.\,\ell_i : B_i$ *, for some* $i \in 1..n$.
- $\Gamma \stackrel{\text{def}}{=} p : [\![B_i]\!]^{\mathsf{ww}}$.

Then

1. $a.\,\ell_i \rightarrow b_i\{a/x_i\}$;
2. $[\![a.\,\ell_i]\!]_p^{\emptyset} \longrightarrow_{\mathtt{d}}^{\star} \sim_\Gamma [\![b_i\{a/x_i\}]\!]_p^{\emptyset}$.

Proof: The first assertion is trivial, so we only look at the second. By Theorem 13.4,

$$\Gamma \vdash [\![a.\,\ell_i]\!]_p^{\emptyset} \tag{22}$$

and, using Lemmas 2.2 and 2.3 and Theorem 13.4,

$$\Gamma \vdash [\![b_i\{a/x_i\}]\!]_p^{\emptyset}\,. \tag{23}$$

We have:

$$[\![a.\,\ell_i]\!]^{\emptyset}_{p} \stackrel{\text{def}}{=} (\boldsymbol{\nu} q\,)([\![a]\!]^{\emptyset}_{q} \mid q(x)\,.\,\overline{x}\,\ell_i\texttt{\_sel\_}p\,)$$

and

$$[\![a]\!]^{\emptyset}_{p} = (\boldsymbol{\nu} x : [\![A]\!]^{\mathsf{b}})\,\overline{p}\,x\,.\,[\![x := a]\!]^{\emptyset}$$

We have the following reductions:

$$\begin{array}{rcll} [\![a.\,\ell_i]\!]^{\emptyset}_{p} & \longrightarrow_{\mathtt{d}} & (\boldsymbol{\nu} q\,)(\boldsymbol{\nu} x\,)([\![x := a]\!]^{\emptyset} \mid \overline{x}\,\ell_i\texttt{\_sel\_}p\,) & \\ & \equiv & (\boldsymbol{\nu} x\,)([\![x := a]\!]^{\emptyset} \mid \overline{x}\,\ell_i\texttt{\_sel\_}p\,) & (24) \end{array}$$

Abbreviating $\tilde{s} \stackrel{\text{def}}{=} s_1 \mathinner{..} s_n$ and $V_j \stackrel{\text{def}}{=} \ell_j\texttt{\_}\big[\texttt{sel\_}(r_j) \,\rhd\, \overline{s_j}\,\langle x, r_j\rangle \;;\, \texttt{upd\_}\,\star\big]$, we have

$$[\![x := a]\!]^{\emptyset} = (\boldsymbol{\nu} \tilde{s}\,)\big(!x\big[_{j\in 1..n}\ V_j\,\big] \mid \Pi_{j\in 1..n}!s_j(x_j, r_j)\,.\,[\![b_j]\!]^{x_j:A}_{r_j}\big)$$

and then we can continue from (24) thus:

$$\begin{array}{ll} \longrightarrow^{\star}_{\mathtt{d}} & (\boldsymbol{\nu} x, \tilde{s}\,)\Big(\overline{s_i}\,\langle x, p\rangle \ \mid\, !x\big[_{j\in 1..n}\ V_j\,\big] \mid \Pi_j!s_j(x_j, r_j)\,.\,[\![b_j]\!]^{x_j:A}_{r_j}\Big) \\ \longrightarrow^{\star}_{\mathtt{d}} & (\boldsymbol{\nu} x, \tilde{s}\,)\Big([\![b_i]\!]^{x_i:A}_{p}\{x/x_i\} \mid\, !x\big[_{j\in 1..n}\ V_j\,\big] \mid \Pi_j!s_j(x_j, r_j)\,.\,[\![b_j]\!]^{x_j:A}_{r_j}\Big) \\ \equiv & (\boldsymbol{\nu} x\,)\Big([\![b_i]\!]^{x_i:A}_{p}\{x/x_i\} \mid (\boldsymbol{\nu} \tilde{s}\,)\big(!x\big[_{j\in 1..n}\ V_j\,\big] \mid \Pi_j!s_j(x_j, r_j)\,.\,[\![b_j]\!]^{x_j:A}_{r_j}\big)\Big) \\ = & (\boldsymbol{\nu} x\,)([\![b_i\{x/x_i\}]\!]^{x:A}_{p} \mid [\![x := a]\!]^{\emptyset}) \\ \sim_{\Gamma} & [\![b_i\{x/x_i\}\{a/x\}]\!]^{\emptyset}_{p} \\ = & [\![b_i\{a/x_i\}]\!]^{\emptyset}_{p} \end{array}$$

where the use of $\sim_{\Gamma}$ is due to Lemma 14.1. We can use this lemma because terms $(\boldsymbol{\nu} x\,)([\![b_i\{x/x_i\}]\!]^{x:A}_{p} \mid [\![x := a]\!]^{\emptyset})$ and $[\![b\{a/x_i\}]\!]^{\emptyset}_{p}$ are well-typed in Γ (for this use (22) and (23) and subject reduction) and because

$$\emptyset \vdash a : A \tag{25}$$

and

$$x_i : A \vdash b_i : B_i \tag{26}$$

hold. Equation (25) comes from Lemma 2.2(1); equation (26) comes from (25), $A_i \leq B_i$ (which comes from Lemma 2.2(1)) and definition of typing.

□

Lemma 14.3 *Let*

- $a \stackrel{\text{def}}{=} \{_{j\in 1..n}\ \ \ell_j = \zeta(x_j : A).\, b_j\}$,
- $a' \stackrel{\text{def}}{=} \{_{j\in\{1..n\}-\{i\}}\ \ \ell_j = \zeta(x_j : A).\, b_j,\ \ell_i = \zeta(x_i : A).\, b\}$,
- $\emptyset \vdash a.\,\ell_i \Leftarrow \zeta(x_i : B).\, b : B$
- $\Gamma \stackrel{\text{def}}{=} p : [\![B]\!]^{\mathsf{ww}}$.

Then

1. $a.\,\ell_i \Leftarrow \zeta(x_i : B).\, b \rightarrow a'$;
2. $[\![a.\,\ell_i \Leftarrow \zeta(x_i : B).\, b]\!]^{\emptyset}_{p} \longrightarrow^{\star}_{\mathtt{d}} \sim_{\Gamma} [\![a']\!]^{\emptyset}_{p}$.

Proof: We prove the second assertion. Let $A = \{_{j\in 1..n}\ \ell_j : B_j\}$. By Lemma 2.2(2), we have $A \leq B$ and the ℓ_i-th component of B is B_i. We have:

$$[\![a.\,\ell_i \Leftarrow \varsigma(x_i : B).\,b]\!]^{\emptyset}_p \stackrel{\text{def}}{=} (\boldsymbol{\nu} q\,)\Big([\![a]\!]^{\emptyset}_q \mid q(x)\,.\,(\boldsymbol{\nu} s : T^{\mathsf{b}}_{B,i})\,\overline{x}\,\ell_i\_\mathtt{upd}\_\langle p, s\rangle\,.\,!s(x_i, r_i)\,.\,[\![b]\!]^{x_i:B}_{r_i}\Big)$$

where $T_{B,i} \stackrel{\text{def}}{=} \langle B^{\mathsf{w}}, [\![B_i]\!]^{\mathsf{ww}}\rangle$, and

$$[\![a]\!]^{\emptyset}_p = (\boldsymbol{\nu} x\,)\overline{p}\,x\,.\,[\![x := a]\!]^{\emptyset}$$

We have the following reductions, for $P \stackrel{\text{def}}{=} !s(x_i, r_i)\,.\,[\![b]\!]^{x_i:B}_{r_i}$. First:

$$\begin{aligned}[\![a.\,\ell_i \Leftarrow \varsigma(x_i : B).\,b]\!]^{\emptyset}_p \quad &\longrightarrow_{\mathtt{d}} \\ &(\boldsymbol{\nu} q\,)(\boldsymbol{\nu} x\,)([\![x := a]\!]^{\emptyset} \mid (\boldsymbol{\nu} s : T^{\mathsf{b}}_{B,i})\,\overline{x}\,\ell_i\_\mathtt{upd}\_\langle p, s\rangle\,.\,P)\end{aligned} \tag{27}$$

Since

$$[\![x := a]\!]^{\emptyset} = (_{j\in 1..n}\,\boldsymbol{\nu}\ s_j : T^{\mathsf{b}}_{A,j}\,)\,\big(\,!x\big[_{j\in 1..n}\ V_j\,\big] \mid \Pi_{j\in 1..n}!s_j(x_j, r_j)\,.\,[\![b_j]\!]^{x_j:A}_{r_j}\big)$$

where for $j \in 1..n$,

$$\begin{aligned} T_{A,j} &\stackrel{\text{def}}{=} \langle [\![A]\!]^{\mathsf{w}}, [\![B_j]\!]^{\mathsf{ww}}\rangle \\ V_j &\stackrel{\text{def}}{=} \ell_j\_\big[\mathtt{sel}\_\star\,;\,\mathtt{upd}\_(r, s') \,\rhd\, U_j\ \big] \\ U_j &\stackrel{\text{def}}{=} (\boldsymbol{\nu} x_{\mathtt{new}} : [\![A]\!]^{\mathsf{b}})\,\overline{r}\,x_{\mathtt{new}}\,.\,\mathtt{OB}^A\langle s_1\,..\,s_{j-1}, s', s_{j+1}\,..\,s_n, x_{\mathtt{new}}\rangle \end{aligned}$$

we can continue from (27) thus:

$$\begin{aligned} \equiv \quad & (\boldsymbol{\nu} s : T^{\mathsf{b}}_{B,i})\,(_{j\in 1..n}\,\boldsymbol{\nu}\ s_j : T^{\mathsf{b}}_{A,j}\,)\,(\boldsymbol{\nu} x\,) \\ & \big(!x\big[_{j\in 1..n}\ V_j\,\big] \mid \Pi_j!s_j(x_j, r_j)\,.\,[\![b_j]\!]^{x_j:A}_{r_j} \mid \overline{x}\,\ell_i\_\mathtt{upd}\_\langle p, s\rangle\,.\,P\big) \\ \longrightarrow^{\star}_{\mathtt{d}} \quad & (\boldsymbol{\nu} s : T^{\mathsf{b}}_{B,i})\,(_{j\in 1..n}\,\boldsymbol{\nu}\ s_j : T^{\mathsf{b}}_{A,j}\,)\,(\boldsymbol{\nu} x\,) \\ & \big(U_i\{^{p,\,s}\!/_{r,\,s'}\} \mid !x\big[_{j\in 1..n}\ V_j\,\big] \mid \Pi_j!s_j(x_j, r_j)\,.\,[\![b_j]\!]^{x_j:A}_{r_j} \mid P\big) \\ \equiv \quad & (\boldsymbol{\nu} s : T^{\mathsf{b}}_{B,i})\,(_{j\in\{1..n\}-\{i\}}\,\boldsymbol{\nu}\ s_j : T^{\mathsf{b}}_{A,j}\,) \\ & \Big(\,U_i\{^{p,\,s}\!/_{r,\,s'}\} \mid P \mid \Pi_{j\in\{1..n\}-\{i\}}!s_j(x_j, r_j)\,.\,[\![b_j]\!]^{x_j:A}_{r_j} \mid \\ & \quad (\boldsymbol{\nu} x, s_i\,)(!x\big[_{j\in 1..n}\ V_j\,\big] \mid !s_i(x_i, r_i)\,.\,[\![b_i]\!]^{x_i:A}_{r_i})\,\Big) \\ \sim_{\Gamma} \quad & (\boldsymbol{\nu} s : T^{\mathsf{b}}_{B,i})\,(_{j\in\{1..n\}-\{i\}}\,\boldsymbol{\nu}\ s_j : T^{\mathsf{b}}_{A,j}\,) \\ & \big(U_i\{^{p,\,s}\!/_{r,\,s'}\} \mid P \mid \Pi_{j\in\{1..n\}-\{i\}}!s_j(x_j, r_j)\,.\,[\![b_j]\!]^{x_j:A}_{r_j}\big) \end{aligned}$$

where the use of $\sim_{\Gamma}$ is due to law **L2**. Moreover, by Lemma 13.1, P is the same as $P' \stackrel{\text{def}}{=} !s(x_i, r_i)\,.\,[\![b]\!]^{x_i:A}_{r_i}$:

$$\begin{aligned} = \ & (\boldsymbol{\nu} s : T^{\mathsf{b}}_{B,i})\,(_{j\in\{1..n\}-\{i\}}\,\boldsymbol{\nu}\ s_j : T^{\mathsf{b}}_{A,j}\,) \\ & \big(U_i\{^{p,\,s}\!/_{r,\,s'}\} \mid P' \mid \Pi_{j\in\{1..n\}-\{i\}}!s_j(x_j, r_j)\,.\,[\![b_j]\!]^{x_j:A}_{r_j}\big) \end{aligned} \tag{28}$$

The process that we have obtained is well-typed under Γ. From $\emptyset \vdash a' : B$ and Theorem 13.4, it also holds that

$$\begin{aligned} \Gamma \ \vdash \ & [\![a']\!]^{\emptyset}_p \\ \equiv \ & (\boldsymbol{\nu} s : T^{\mathsf{b}}_{A,i})\,(_{j\in\{1..n\}-\{i\}}\,\boldsymbol{\nu}\ s_j : T^{\mathsf{b}}_{A,j}\,) \\ & \big(U_i\{^{p,\,s}\!/_{r,\,s'}\} \mid P' \mid \Pi_{j\in\{1..n\}-\{i\}}!s_j(x_j, r_j)\,.\,[\![b_j]\!]^{x_j:A}_{r_j}\big) \end{aligned}$$

Because of these facts, we can apply law **L4**; thus continuing from (28) we infer

$$\sim_{\Gamma} [\![a']\!]^{\emptyset}_p$$

which proves the lemma. $\square$

Lemma 14.4 *If a is closed and $[\![a]\!]_p^E \downarrow$, then a is a value.*

Proof: Immediate by structural induction on a (note that the observation predicate is "strong", that is, no silent steps can occur before the visible action). $\square$

Theorem 14.5 (operational correspondence) *Suppose $\emptyset \vdash a : A$, and let $\Gamma \stackrel{\text{def}}{=} p : [\![A]\!]^{\mathsf{ww}}$. It holds that:*

1. *If $a \to a'$, then $[\![a]\!]_p^\emptyset \longrightarrow_{\mathtt{d}}^\star \sim_\Gamma [\![a']\!]_p^\emptyset$;*

2. *if $[\![a]\!]_p^\emptyset \to P$, then there is a' s.t. $a \to a'$, $P \longrightarrow_{\mathtt{d}}^\star \sim_\Gamma [\![a']\!]_p^\emptyset$ and, moreover, the reduction $[\![a]\!]_p^\emptyset \to P$ is deterministic.*

Proof: Assertion (1) is proved by induction on the depth of the proof of $a \to a'$. The most interesting cases are the basic ones; for these use Lemmas 14.2 and 14.3. In the case of R-Upd, Lemma 2.2(2) might be needed so to transform A into some $A' \leq A$ before applying Lemma 14.3. Then Lemmas 9.10 and 9.12 allow us to replace $\sim_{p:[\![A']\!]^{\mathsf{ww}}}$ with $\sim_{p:[\![A]\!]^{\mathsf{ww}}}$. In the cases of R-Eval1-2, Lemmas 9.10, 9.11 and 9.12 are needed in order to manipulate processes inside contexts.
Assertion (2) is proved similarly, proceeding by induction on the structure of a and using Lemma 14.4. For instance, consider the case where a is of the form $b.\ell \Leftarrow \zeta(x : A).b'$. Then $[\![a]\!]_p^\emptyset = (\boldsymbol{\nu} q : [\![A]\!]^{\mathsf{wb}})\,([\![b]\!]_q^\emptyset \mid q(x).Q)$, for some Q. If $[\![a]\!]_p^\emptyset \to P$, then the reduction either comes from $[\![b]\!]_q^\emptyset$ alone, or it comes from an interaction between $[\![b]\!]_q^\emptyset$ and $q(x).Q$. The former case is solved using the inductive assumption as in assertion (1) for rule R-Eval2. In the latter case, it holds that $[\![b]\!]_q^\emptyset \downarrow$ and therefore, by Lemma 14.4, b is a value; then we proceed as in assertion (1) for rule R-Upd. $\square$

Note that the correctness of the interpretation on reductions (Theorem 14.5) and type judgements (Theorem 13.4 and Theorem 13.6) imply the subject reduction property for OC.

The translation captures precisely convergence on the source language.

Corollary 14.6 (computational adequacy) *If a is closed, then $a \Downarrow$ iff $[\![a]\!]_p^\emptyset \Downarrow$.*

Proof: Consequence of Lemma 14.4 and of the operational correspondence Theorem 14.5. $\square$

We can define barbed bisimulation and congruence on OC terms as we did in the π-calculus. However, since the reduction relation of OC is confluent, clause (1) of the definition of barbed bisimulation can be omitted. Therefore in OC barbed congruence coincides with Morris-style contextual equivalence. A context C in OC is a (A/B)-*context* if $\emptyset \vdash C : A$ holds assuming that the hole of C has type B.

Definition 14.7 *For closed OC terms a and b of type B,we write $a \approx_B b$ if for all A and (A/B)-context C, it holds that $C[a] \Downarrow$ iff $C[b] \Downarrow$.*

Using the compositionality of the encoding and adequacy, one can show the soundness of the translation, which tells us that the equalities that can be proved via the translation are valid.

Theorem 14.8 (soundness) *Let a and b be closed terms of type B. If $[\![a]\!]_p^\emptyset \approx_{p:[\![B]\!]^{\mathsf{ww}}} [\![b]\!]_p^\emptyset$ then $a \approx_B b$.*

The converse of Theorem 14.8 does not hold. Consider objects a and b with type $\{\ell_1 : \{\mathtt{h} : \{\ \}\} \,;\, \ell_2 : \{\ \}\}$ and definition

$$
\begin{aligned}
a &\stackrel{\text{def}}{=} \{\ell_1 := e_1;\ \ell_2 = \zeta(x).\, \Big((x.\,\ell_1).\,\mathtt{h} \Leftarrow e_2\Big).\,\mathtt{h} \Leftarrow e_2\} \\
b &\stackrel{\text{def}}{=} \{\ell_1 := e_1;\ \ell_2 = \zeta(x).\, (x.\,\ell_1).\,\mathtt{h} \Leftarrow e_2\}
\end{aligned}
$$

for some expressions e_1 and e_2. The only difference between a and b is that, in a, the same update operation in the body of method ℓ_2 is repeated. The two objects are equivalent in OC. However, they are distinguishable in the π-calculus, and we informally explain the reason. An external observer can update method ℓ_1, thus becoming the owner of the body of this method and then can respond in a non-deterministic way to requests of access to ℓ_1's body. In particular, the observer can decide to diverge after precisely two requests of update on the submethod $\mathtt{h}$. In this case, when method ℓ_2 of object a or b is selected, the system with a reaches divergence, whereas the system with b does not.

This counterexample is similar to the counterexamples to full abstraction for the encodings of the λ-calculus into the π-calculus, see [Mil91, San95]. Note that full abstraction also fails for the translations of OC into the λ-calculus in [ACV96].

15 Conclusions and future work

We have showed that a typed π-calculus can capture the type and subtype structure and the operational content of a core, but challenging, Object-Oriented calculus. The proofs of operational correctness of the interpretation are indicative examples of the usefulness of type information for reasoning on π-calculus process.

The type language for the π-calculus has Pierce and Sangiorgi's I/O annotations, to separate the capabilities of reading and writing on a channel, and *variant types*. Technical contributions of the paper are the presentation of variant types for the π-calculus and their typing and subtying properties, and an analysis of behavioural equivalences in a π-calculus with variant types.

The results of operational correspondence (Theorem 14.5) are stronger than the results for the encodings of the λ-calculus into the π-calculus [Mil91, San95]: The statements of the latter results make use of *weak* behavioural equivalences, whereas in Theorem 14.5 a *strong* behavioural equivalence suffices.

The language we have interpreted is *sequential*. We have chosen a sequential language because type systems for sequential Object-Oriented languages are better understood. We hope that studying these type systems from within the π-calculus will shed some light onto type systems for parallel Object-Oriented languages.

Our translation uses only a subset of the π-calculus language where: All inputs are either persistent or linear; a name appears in input subject position at most once; a name received in an input can only be used in output position. The processes obeying this discipline are "functional", in that they have a confluent reduction relation. Some of these constraints appear in π-calculus-like languages studied by Amadio [Ama97], Boreale [Bor96], Fournet and Gonthier [CG96]. Identifying combinations of the π-calculus operators which are useful for the interpretation of objects might lead to the definition of a higher-level target calculus, capable of yielding more succinct and readable interpretations of objects.

We would like to apply the interpretation and the theory of the π-calculus, in particular its algebraic laws and its proof techniques, to proving behavioural properties on OC terms. Indeed, a major reason for experimenting the π-calculus in the semantics of Object-Oriented languages is the wish to exploit its theory. We would also like to compare proofs of behavioural equalities between OC terms carried out via the π-calculus translation, against direct proofs using applicative bisimulation as defined by Gordon and Rees [GR96]. Obviously, since the π-calculus translation is not fully abstract, there will be proofs for which only applicative bisimulation can be used.

We have interpreted OC, that is a *core* functional Object Calculus, because it already contains most of the basic challenges to interpreting typed functional Object-Oriented languages into the π-calculus and because we wanted to keep our interpretation and proofs shorter and easier to read. Abadi and Cardelli have studied an extension of the type system for OC with variant annotations $\{+,-,o\}$ on the method names so to have a richer subtyping relation (see [AC95]). A tag ℓ^+, ℓ^- or ℓ^o says, respectively, that method ℓ can only be selected, only updated, or both selected and updated. Tag $+$ gives covariance, $-$ gives contravariance and o gives invariance. It is simple to capture this extension of the OC type system in our π-calculus interpretation, playing with the variant types. For instance, the translation of a type $\{\ell_1^+ : B_1, \ell_2^- : B_2\}$ would be

$$\mu X.\left[\ell_1\_[\mathtt{sel}\_[\![B_1]\!]^{\mathsf{w}\mathsf{w}}]\,;\,\ell_2\_[\mathtt{upd}\_\langle X^{\mathsf{w}\mathsf{w}}, \langle X^{\mathsf{w}}, [\![B_2]\!]^{\mathsf{w}\mathsf{w}}\rangle^{\mathsf{w}}\rangle]\qquad\right]$$

It is worth noticing that the tags $\{+,-,o\}$ yield the same form of subtyping on OC types as that induced by the tags $\{\mathsf{r},\mathsf{w},\mathsf{b}\}$ on the π-calculus types. Abadi and Cardelli have also investigated a second-order extension of OC [AC94a]. It should be possible to extend our encoding of the first-order OC to an encoding of the second-order OC, by adding polymorphic types to the typed π-calculus following Turner [Tur96].

By contrast, how to repeat out program onto the imperative version of OC [AC95] is less predictable. We think that an encoding of the imperative calculus can be written which has a similar structure to the encoding of the functional calculus presented in this paper. The main modification should be in the object manager $\mathtt{OB}^A\langle s_1..s_n, x\rangle$: In our translation, this is a functional process; translating the imperative OC, it should be made "imperative", by allowing updates of the names $s_1..s_n$ for accessing the methods. However, it would be good to see the details worked out.[3]

All Object Calculi are sequential. One hopes that studying their type systems from within the π-calculus will help to develop concurrent or distributed versions of the Object Calculi.

Acknowledgements

The core of this research was carried out while the author was visiting the Isaac Newton Institute for Mathematical Sciences, Cambridge, U.K., for the programme on "Semantics of Computation", from September to December 1995.

I am most grateful to Ramesh Viswanathan for several discussions during the set up of the encoding. I also benefited from comments by Martín Abadi, Gérard Boudol, Kim Bruce, Luca Cardelli, Andy Gordon and Benjamin Pierce. The suggestions of the two anonymous referees have improved the technical presentation.

This research has been supported by the CNET project "Modélisation de Systèmes Mobiles".

[3]By the time this paper is reviewed, a π-calculus interpretation of the imperative Object Calculus has been given [KS97]. The two interpretations differ precisely in the definition of the object manager, as suggested above. The same paper contains some examples of behavioural equivalences between OC terms proved using the π-calculus interpretations, and a discussion of the difference with corresponding proofs done using Gordon and Rees's applicative bisimulation.

References

[AC93] R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993. A preliminary version appeared in POPL '91 (pp. 104–118), and as DEC Systems Research Center Research Report number 62, August 1990.

[AC94a] M. Abadi and L. Cardelli. A theory of primitive objects: Second-order systems. In *Proc. ESOP'94*. Springer Verlag, 1994.

[AC94b] M. Abadi and L. Cardelli. A theory of primitive objects: Untyped and first-order systems. In *Proc. Theoretical Aspects of Computer Science*, volume 789 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.

[AC95] M. Abadi and L. Cardelli. An imperative object calculus. In Mosses P. et al. editor, *Proc. of TAPSOFT'95*, volume 915 of *Lecture Notes in Computer Science*. Springer Verlag, 1995.

[ACV96] M. Abadi, L. Cardelli, and R. Viswanathan. An interpretation of Objects and Objects Types. In *Proc. 23th POPL*. ACM Press, 1996.

[Ama97] R. Amadio. An asynchronous model of locality, failure, and process mobility. In Proc. COORDINATION'97, volume 1282 of *Lecture Notes in Computer Science*. Springer Verlag, 1997.

[Ame89] P. America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1(4):366–411, 1989.

[Bar84] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic*. North Holland, 1984. Revised edition.

[Bor96] M. Boreale. On the expresiveness of internal mobility in name-passing calculi. In *Proc. CONCUR '96*, volume 1119 of *Lecture Notes in Computer Science*. Springer Verlag, 1996.

[CG96] Fournet C. and Gonthier G. The Reflexive Chemical Abstract Machine and the Join calculus. In *Proc. 23th POPL*. ACM Press, 1996.

[GR96] A.D. Gordon and G.D. Rees. Bisimilarity for a first-order calculus of objects with subtyping. In *Proc. 23th POPL*. ACM Press, 1996.

[HK96] Hans Hüttel and Josva Kleist. Objects as mobile processes. Technical report RR-96-38, BRICS - Basic Research in Computer Science, 1996.

[Hon96] K. Honda. Composing processes. In *Proc. 23th POPL*. ACM Press, 1996.

[Jon93] C.B. Jones. A π-calculus semantics for an object-based design notation. In E. Best, editor, *Proc. CONCUR '93*, volume 715 of *Lecture Notes in Computer Science*, pages 158–172. Springer Verlag, 1993.

[Kam88] S. Kamin. Inheritance in Smalltalk-80: a denotational definition. In *Proc. 15th POPL*. ACM Press, 1988.

[KPT96] N. Kobayashi, B.C. Pierce, and D.N. Turner. Linearity and the pi-calculus. In *Proc. 23th POPL*. ACM Press, 1996.

[KS97] J. Kleist and D. Sangiorgi. Imperative objects and mobile processes. Draft. Available electronically as `ftp://zenon.inria.fr/meije/theorie-par/davides/iomp.ps.gz`, October 1997.

[LW96] X. Liu and D. Walker. Partial confluence of processes and systems of objects. Submitted for publication, 1996.

[Mil91] R. Milner. The polyadic π-calculus: a tutorial. Technical Report ECS–LFCS–91–180, LFCS, Dept. of Comp. Sci., Edinburgh Univ., October 1991. Also in *Logic and Algebra of Specification*, ed. F.L. Bauer, W. Brauer and H. Schwichtenberg, Springer Verlag, 1993.

[Mil92] R. Milner. Functions as processes. *Journal of Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, (Parts I and II). *Information and Computation*, 100:1–77, 1992.

[MS92] R. Milner and D. Sangiorgi. Barbed bisimulation. In W. Kuich, editor, *19th ICALP*, volume 623 of *Lecture Notes in Computer Science*, pages 685–695. Springer Verlag, 1992.

[Ode95] Martin Odersky. Polarized name passing. In *Proc. FST & TCS*, Lecture Notes in Computer Science. Springer Verlag, 1995.

[PS96] B. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Journal of Mathematical Structures in Computer Science*, 6(5):409–454, 1996. An extended abstract in *Proc. LICS 93*, IEEE Computer Society Press.

[PT96] B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. In preparation, 1996.

[San92] D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms.* PhD thesis CST–99–93, Department of Computer Science, University of Edinburgh, 1992.

[San95] D. Sangiorgi. Lazy functions and mobile processes. Technical Report RR-2515, INRIA-Sophia Antipolis, 1995. To appear in "Festschrift volume in honor of Robin Milner's 60th birthday", MIT Press.

[Tur96] N.D. Turner. *The polymorphic pi-calculus: Theory and Implementation.* PhD thesis, Department of Computer Science, University of Edinburgh, 1996. PhD thesis, University of Edinburgh.

[VH93] V.T. Vasconcelos and K. Honda. Principal typing schemes in a polyadic π-calculus. In E. Best, editor, *Proc. CONCUR '93*, volume 715 of *Lecture Notes in Computer Science*. Springer Verlag, 1993.

[VT93] V.T. Vasconcelos and M. Tokoro. A typing system for a calculus of objects. In *Proc. Object Technologies for Advanced Software '93*, volume 742 of *Lecture Notes in Computer Science*, pages 460–474. Springer Verlag, 1993.

[Wal95] D. Walker. Objects in the π-calculus. *Information and Computation*, 116(2):253–271, 1995.

[Yos96] Nobuko Yoshida. Graph types for monadic mobile processes. In *Proc. FST & TCS*, volume 1180 of *Lecture Notes in Computer Science*, pages 371–386. Springer Verlag, 1996.