Local π-Calculus at Work: Mobile Objects as Mobile Processes[1]

Massimo Merro[2]
COGS, University of Sussex, United Kingdom

Josva Kleist[3]
INRIA, Sophia-Antipolis, France

Uwe Nestmann
EPF, Lausanne, Switzerland

February 7, 2001

[1]An extended abstract has appeared in *Proceedings of IFIP TCS 2000*, volume 1872 of *Lecture Notes in Computer Science*. Springer Verlag, August 2000.
[2]Partly supported by Marie Curie fellowhip, EU-TMR, No. ERBFMBICT983504.
[3]Partly supported by Danish National Research Foundation grant SNF-28808

Abstract

Obliq is a lexically-scoped, distributed, object-based programming language. In Obliq, the *migration* of an object is proposed as creating a clone of the object at the target site, whereafter the original object is turned into an alias for the clone. Obliq has only an informal semantics, so there is no proof that this style of migration is safe, i.e., transparent to object clients. In previous work, we introduced Øjeblik, an abstraction of Obliq, where, by lexical scoping, sites have been abstracted away. We used Øjeblik in order to exhibit how the semantics behind Obliq's implementation renders migration unsafe. We also suggested a modified semantics that we conjectured instead to be safe. In this paper, we rewrite our modified semantics of Øjeblik in terms of π-*calculus*, and we use it to formally prove the correctness of *object surrogation*, the abstraction of object migration in Øjeblik.

1 Introduction

The work presented in this paper is in line with the research activity to use the π-calculus as a toolbox for reasoning about object-based programming languages. Former works on the semantics of objects as processes showed the value of this approach: while [Wal95, HK96, San98, KS98] focused on just providing formal semantics to object-oriented languages and language features, the work of others [PW98, San99b] has been driven by a specific programming problem. Our work tackles a problem in Cardelli's *lexically-scoped distributed* programming language *Obliq* [Car95]. Cardelli proposed to derive *object migration* from two other primitives, *cloning* and *aliasing*, by performing one after the other. In Obliq, immutable values can be freely copied from site to site, whereas mutable values are stationary. Only references to mutable values may be transmitted between different sites. Accordingly, since objects are mutable, the migration of an object does not physically move the object, but instead creates a *clone* of the object at the target site and then turns the original (local) object into an *alias*—sometimes called a *proxy*—for the new (remote) object.

1.1 Previous work

When is object migration correct? In concurrent and distributed programs, it is important that certain state changes, in parts of the running system, may happen transparently from the point of view of the rest of the system. Ensuring that the implementation of such state changes is in fact transparent can be a difficult task since the programmer must in principle anticipate all possible execution scenarios. In Obliq, a natural question is, whether migration of an object is transparent to the object's clients, and how that can be stated formally. Intuitively, migration of an object a to some other site works transparently, or safely, if (i) during migration it is not possible to interact with a in a way that prevents the migration operation from proper completion, and if (ii) after the migration a client of a cannot tell that a is now an alias. In Obliq, mobile objects are therefore required to be *serialised* and *protected*: serialization guarantees atomicity of the two-phase migration operation; protection guarantees that aliases are persistent.

From Migration to Surrogation *Lexical scoping* in distributed settings makes program analysis easier since the binding of variables is completely determined by their location in the program text, and not by the execution site. Since Obliq is lexically-scoped, we can ignore the aspects of distribution, at least when regarding the results of Obliq computations, unless sites fail. Following this idea, we focus on Øjeblik [NHKM00], an object-based language that represents Obliq's concurrent core, but can also be seen as a concurrent extension of the Imperative Object Calculus [AC96]. Øjeblik supports a distribution-free abstraction of migration called *surrogation*. Like migration, the *surrogation* of an object a is described as the creation of a clone b of a and then turning a itself into a proxy for b, which forwards future request for methods of a to b. The main difference with respect to migration is that neither a nor b are attached to any site.

Correctness as an equation In [NHKM00], we gave a formal definition of *correctness* for object surrogation in Øjeblik which can be straightforwardly adapted to object migration in Obliq. The intuition is that, in order to be correct, the surrogation of an object must be transparent to the clients of that object, i.e., the object must behave the same before and after surrogation. We formalized this concept by means of a simple equation:

$$a.\mathsf{ping} \doteq a.\mathsf{surrogate}$$

where the left side represents the object a before surrogation ($a.\mathsf{ping}$ returns the object resulting from the evaluation of a), the right side represents the object a after surrogation ($a.\mathsf{surrogate}$ returns the surrogated object), and $\doteq$ is an appropriate contextual equivalence, based on the possibility of convergence.

Aliasing Semantics In [NHKM00], we gave several proposals of configuration-style semantics for Øjeblik. One of them fits the Obliq implementation [Car94, Car95], but does not guarantee the correctness of object surrogation as defined above. This was formally shown by exhibiting Øjeblik contexts that are able to distinguish the terms $a.\mathsf{ping}$ and $a.\mathsf{surrogate}$. Similar counterexamples apply to object migration in Obliq, as we tested using the Obliq interpreter [Car94]. Roughly, the reason is because, in Obliq, alias nodes support a too strong form of both protection and serialization. As a consequence, in [NHKM00] we proposed a different semantics in which alias nodes have a milder form of both protection and serialization. In that paper, we conjectured that object migration is safe when considering this new semantics, but no proof was given.

1.2 Contribution

In this paper, we present a π-calculus semantics for Øjeblik corresponding to the aforementioned variant proposed in [NHKM00]. We also give a notion of contextual equivalence for objects defined in terms of may convergence on π-processes corresponding to the equivalence $\doteq$. More precisely, our semantics uses *Local* π [MS98, Mer00], in short Lπ, a variant of the asynchronous π-calculus [HT91, Bou92], where, like in Join-calculus [FG96], the recipients of a channel are local to the process that has created the channel. We prove the correctness of surrogation for a wide class of Øjeblik-programs. The proof is in two parts: an *algebraic part* and an *iterative part*. The algebraic part (Theorem 8.1) relates the core component of the translation of a single object after having committed to a ping and a $\mathsf{surrogate}$ request, respectively. We use powerful adaptations of proof techniques, from standard π-calculus and Lπ. The iterative part (Theorem 8.7) relates the may-convergence behavior of the terms $a.\mathsf{ping}$ and $a.\mathsf{surrogate}$ within arbitrary Øjeblik-contexts; note that in these terms the operations have not yet been performed, and will only do so at some point if the context permits. In Theorem 8.7, we constructively simulate arbitrarily long converging sequences "up to" Theorem 8.1. The main difficulty of Theorem 8.7 is that inherently concurrent Øjeblik-contexts may non-deterministically prevent either term from eventually committing to the requested operation.

The proof is non-trivial, and we give (to our knowledge) the first formal proof that object migration can be correctly implemented in terms of cloning and aliasing (apart from a very restrictive and informal sketch of our own [HKMN99], on which we improve substantially, here). Finally, we want to remark that most counterexamples presented in [NHKM00, Mer00] (exhibiting the problems of Obliq's original semantics) were actually discovered while using some π-calculus semantics to understand Obliq programs and trying to prove the correctness of surrogation.

1.3 Related work

The work closest to ours is [KS98] where an interpretation of Abadi and Cardelli's object calculus [AC96] into typed π-calculus is presented. Unlike [KS98], we focus on a concurrent object calculus. Gordon and Hankin [GH98], and Di Blasio and Fisher [DF96] describe two concurrent object calculi, but no account of object migration is given for them. An early version of Emerald [JLHB88] includes a form of object migration similar to that one in Obliq, but little formal work is known about it. Finally, in Distributed Oz [VHB$^+$97], object migration is a primitive notion, so objects are physically mobile and travel according to a provably safe mobile state protocol from site to site, wherever they are needed or intend to go.

2 Local π: An "Object-Oriented" π-Calculus

Local π [MS98, Mer00], in short Lπ, is a variant of the asynchronous π-calculus [HT91, Bou92] where, similar to the Join-calculus [FG96], the recipients of a channel are local to the process that has created the channel. This is achieved by imposing the syntactic constraint that only the output capability of channels may be transmitted, i.e., the recipient of a channel may only use it in output actions. This property makes Lπ particularly suitable for giving the semantics to, and

Channels:	$c \in \mathbf{C}$	*Values*	
Keys:	$k \in \mathbf{K}$	$v \;::=\; x$	variable
Names:	$\in \mathbf{N}$	$\mid\; \ell\_v$	variant
$n \;::=\; c \mid k$		$\mid\; \langle\, v_1 \,..\, v_n \,\rangle$	tuple
Auxiliary:	$u \in \mathbf{U}$	*Types*	
Variables:	$\in \mathbf{X}$	$T \;::=\; \mathbf{C}(T)$	channel type
$x \;::=\; n \mid u$		$\mid\; \mathbf{K}$	key type
		$\mid\; [\, \ell_1{:}T_1 \,;\ldots;\, \ell_m{:}T_m \,]$	variant type
Labels	$\in \mathbf{L}$	$\mid\; \langle\, T_1 \,..\, T_m \,\rangle$	tuple type
$\ell, \ell_1, \ell_2, \ldots$		$\mid\; X$	type variable
		$\mid\; \mu X.T$	recursive type

Processes

$P \;::=\; \mathbf{0}$	nil process
$\mid\; c(x).P$	single *input*
$\mid\; \overline{c}v$	output
$\mid\; P_1 \mid P_2$	parallel
$\mid\; (\boldsymbol{\nu} n{:}T)\, P$	restriction
$\mid\; \mathbf{!}\, c(x).P$	replicated *input*
$\mid\;$ if $[k{=}k_1]$ then P_1 elif $[k{=}k_2]$ then P_2 else P_3	key testing
$\mid\;$ case v of $\ell_1\_(x_1){:}P_1 \,;\ldots;\, \ell_m\_(x_m){:}P_m$	variant *destructor*
$\mid\;$ let $(\, x_1 \,..\, x_m \,) = v$ in P	tuple *destructor*
$\mid\;$ wrong	run time error

The *locality constraint* requires that in (*single* and *replicated*) *inputs* and in (*variant* and *tuple*) *destructors* the bound names $x, x_1, \ldots, x_m$ must not be used in free input position within the respective scope $P, P_1, \ldots, P_m$.

Table 1: The Calculus $\mathrm{L}\pi^+$

reasoning about, concurrent object-oriented languages. In particular, we can easily guarantee the uniqueness of object identities—a fundamental feature of objects: in object-oriented languages, the name of an object may be transmitted; the recipient may use that name to access the methods of the object, but it cannot create a new object with the same name. When representing objects in the π-calculus, this translates directly into the constraint that the process receiving an object name may only use it in output actions—a guarantee in our setting.

2.1 Terms and Types

In Table 1, we introduce the calculus $\mathrm{L}\pi^+$, a typed version of polyadic $\mathrm{L}\pi$ with: (i) labelled values ℓ_v, called *variants* [San98], with case analysis; (ii) tuple values $\langle\, v_1 \,..\, v_n \,\rangle$, with pattern matching, (iii) constants k, called *keys*, with equality; (iv) a wrong construct to model run-time typing errors.

We introduce a few syntactic categories: the set $\mathbf{X}$ of *variables* includes the set $\mathbf{N}$ of *names* (constants and variables) consisting of the two disjoint sets $\mathbf{C}$ of *channels* and $\mathbf{K}$ of *keys*. The auxiliary variables in the set $\mathbf{U}$ are variables for complex values. $\mathbf{L}$ is the set of *labels*. In addition to the metavariables mentioned in the grammar, we let s, p, q, r, m, t range over channels, y over variables, w over values, Q over processes, and i, j, d, h, m over tuple, variant, or other indices. We abbreviate $\ell\_\langle\rangle$ and $\ell\_()$ as ℓ, as well as $\overline{q}\langle\rangle$ and $q().P$ as $\overline{q}$ and $q.P$, respectively, while $\widetilde{v}$ denotes a sequence $v_1 \,..\, v_m$.

Restriction, both inputs, and both destructors are *binders* for the names $x, x_1, \ldots, x_m$ in the respective scopes $P, P_1, \ldots, P_m$. We assume the usual definitions of free and bound occurrences of names, based on these binders; the inductively defined functions $\mathrm{fn}(P)$ and $\mathrm{bn}(P)$ denote those of process P. Similarly, $\mathrm{fc}(P)$ and $\mathrm{bc}(P)$ denote the free and bound channels of process P. Moreover, $\mathrm{n}(P){=}\,\mathrm{fn}(P){\cup}\mathrm{bn}(P)$ and $\mathrm{c}(P){=}\mathrm{fc}(P){\cup}\mathrm{bc}(P)$. *Substitutions*, ranged over by σ, are type-preserving functions from variables to values (types are introduced below). For an expression e, $e\sigma$ is the result of applying σ to e, with the usual renaming to avoid captures. *Relabellings*, ranged over by ρ, permit replacing a label ℓ with another label ℓ'. We denote such a relabelling with $[{}^{\ell'}/_{\ell}]$. The application of a relabelling to a term is defined thus:

- $(\ell\_v)[{}^{\ell'}/_{\ell}] := \ell'\_v[{}^{\ell'}/_{\ell}]$
- $(\ell''\_v)[{}^{\ell'}/_{\ell}] := \ell''\_v[{}^{\ell'}/_{\ell}]$ if $\ell''{\neq}\ell$
- $x\rho := x$
- $\mathsf{wrong}\rho := \mathsf{wrong}$
- $((\boldsymbol{\nu} n{:}T)\, P)\rho := (\boldsymbol{\nu} n{:}T\rho)\, P\rho$
- $(\mathsf{case}\ v\ \mathsf{of}\ \ell_1\_(x_1){:}P_1\,;\ldots;\ell_n\_(x_n){:}P_n)\rho := \mathsf{case}\ v\rho\ \mathsf{of}\ \ell_1\_(x_1){:}(P_1\rho)\,;\ldots;\ell_n\_(x_n){:}(P_n\rho)$.

For the remaining (value and process) constructors, relabellings act as simple homomorphisms. Substitution and relabelling have the highest operator precedence, parallel composition the lowest.

To rearrange processes we use the following notion of *structural equivalence* that is extended to deal with if-, case-, and let-constructs.

Definition 2.1 Structural equivalence, *written $\equiv$, is the smallest relation preserved by parallel composition and restriction, which satisfies the axioms below:*

- $P \equiv Q$, *if P is α-convertible to Q*
- $P \mid \mathbf{0} \equiv P$, $P \mid Q \equiv Q \mid P$, $P \mid (Q \mid R) \equiv (P \mid Q) \mid R$
- $(\boldsymbol{\nu} n{:}T)\,\mathbf{0} \equiv \mathbf{0}$, $(\boldsymbol{\nu} n_1{:}T_1)\,(\boldsymbol{\nu} n_2{:}T_2)\,P \equiv (\boldsymbol{\nu} n_2{:}T_2)\,(\boldsymbol{\nu} n_1{:}T_1)\,P$, *if $n_1 \neq n_2$*
- $(\boldsymbol{\nu} n{:}T)\,(P \mid Q) \equiv P \mid (\boldsymbol{\nu} n{:}T)\,Q$, *if $n \notin \mathrm{fn}(P)$*
- $\mathsf{if}\ [k_1{=}k_1]\ \mathsf{then}\ P_1\ \mathsf{elif}\ [k_1{=}k_2]\ \mathsf{then}\ P_2\ \mathsf{else}\ P_3 \equiv P_1$
- $\mathsf{if}\ [k_2{=}k_1]\ \mathsf{then}\ P_1\ \mathsf{elif}\ [k_2{=}k_2]\ \mathsf{then}\ P_2\ \mathsf{else}\ P_3 \equiv P_2$, *if $k_1{\neq}k_2$*
- $\mathsf{if}\ [k{=}k_1]\ \mathsf{then}\ P_1\ \mathsf{elif}\ [k{=}k_2]\ \mathsf{then}\ P_2\ \mathsf{else}\ P_3 \equiv P_3$, *if $k_1{\neq}k{\neq}k_2$*
- $\mathsf{case}\ \ell_j\_v_j\ \mathsf{of}\ \ell_1\_(x_1){:}P_1\,;\ldots;\ell_j\_(x_j){:}P_j\,;\ldots;\ell_m\_(x_m){:}P_m \equiv P_j\{{}^{v_j}/_{x_j}\}$
- $\mathsf{case}\ v\ \mathsf{of}\ \ell_1\_(x_1){:}P_1\,;\ldots;\ell_m\_(x_m){:}P_m \equiv \mathsf{wrong}$, *if $v{\neq}\ell_j\_v_j$ for any $j{\in}\,1\,..\,m$ and value v_j*
- $\mathsf{let}\ (\,x_1\,..\,x_m\,) = \langle\, v_1\,..\,v_m\,\rangle\ \mathsf{in}\ P \equiv P\{{}^{\widetilde{v}}/_{\widetilde{x}}\}$*;*
- $\mathsf{let}\ (\,x_1\,..\,x_m\,) = v\ \mathsf{in}\ P \equiv \mathsf{wrong}$, *if $v{\neq}\langle\, v_1\,..\,v_m\,\rangle$ for any values $v_1\,..\,v_m$.*

In Table 2 we give *typing rules* for values and processes. Types are introduced for essentially three reasons: (i) they allow us to cleanly define some abbreviations, (ii) we use them to give a typed semantics of Øjeblik, and (iii) they allow us to formally prove the main result of the paper using typed behavioural equivalences. Abusing the notation for sets of names and the corresponding types, we use $\mathbf{K}$ and $\mathbf{C}$ also as type constructors, where channel types are parameterised over the type of value they carry. For variants and tuples we use standard notations (c.f. [San98]). In a recursive type $\mu X.T$, occurrences of variable X in type T must be guarded, i.e., underneath variant, tuple, or channel constructors. We often omit the type annotation of restriction, when it is clear from the context or not important for the discussion.

A *type environment* Γ is a finite mapping from variables to types. 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. We say that a type environment Γ is *closed* if all names mentioned in Γ are of type channel $\mathbf{C}(T)$ or of type key $\mathbf{K}$. We only consider *closed terms*, i.e. terms which are well-typed in some closed typing Γ.

As expected, the typing in Table 2 satisfies all basic fundamental properties of type environments such as: *weakening*, *contraction*, *substitution*, and *narrowing*.

$$\text{(T-BAS)}\ \frac{\Gamma(x) = T}{\Gamma \vdash x : T}$$

$$\text{(T-REC1)}\ \frac{\Gamma \vdash v{:}T\{{}^{\mu X.T}/_{X}\}}{\Gamma \vdash v{:}\mu X.T} \qquad \text{(T-REC2)}\ \frac{\Gamma \vdash v{:}\mu X.T}{\Gamma \vdash v{:}T\{{}^{\mu X.T}/_{X}\}}$$

$$\text{(T-VAR)}\ \frac{\Gamma \vdash v{:}T}{\Gamma \vdash \ell\_v : [\,\ldots ; \ell{:}T\,; \ldots\,]} \qquad \text{(T-TUP)}\ \frac{\Gamma \vdash v_i{:}T_i \quad \forall i \in 1\,..\,m}{\Gamma \vdash \langle\, v_1\,..\,v_m\,\rangle : \langle\, T_1\,..\,T_m\,\rangle}$$

$$\text{(T-Res1)}\ \frac{\Gamma, c{:}\mathbf{C}(T) \vdash P}{\Gamma \vdash (\boldsymbol{\nu} c{:}\mathbf{C}(T))\ P} \qquad \text{(T-Res2)}\ \frac{\Gamma, k{:}\mathbf{K} \vdash P}{\Gamma \vdash (\boldsymbol{\nu} k{:}\mathbf{K})\ P}$$

$$\text{(T-Par)}\ \frac{\Gamma \vdash P_1 \quad \Gamma \vdash P_2}{\Gamma \vdash P_1 \mid P_2} \qquad \text{(T-Rep)}\ \frac{\Gamma \vdash P}{\Gamma \vdash \mathbf{!}\,P} \qquad \text{(T-Nil)}\ \frac{-}{\Gamma \vdash \mathbf{0}}$$

$$\text{(T-Inp)}\ \frac{\Gamma \vdash c{:}\mathbf{C}(T) \quad \Gamma, x{:}T \vdash P}{\Gamma \vdash c(x).P} \qquad \text{(T-Out)}\ \frac{\Gamma \vdash c{:}\mathbf{C}(T) \quad \Gamma \vdash v{:}T}{\Gamma \vdash \overline{c}v}$$

$$\text{(T-If)}\ \frac{\Gamma \vdash k, k_1, k_2 : \mathbf{K} \qquad \Gamma \vdash P_1, P_2, P_3}{\Gamma \vdash \mathsf{if}\ [k{=}k_1]\ \mathsf{then}\ P_1\ \mathsf{elif}\ [k{=}k_2]\ \mathsf{then}\ P_2\ \mathsf{else}\ P_3}$$

$$\text{(T-Let)}\ \frac{\Gamma \vdash v : \langle\, T_1\,..\,T_m\,\rangle \qquad \Gamma, x_1{:}T_1, \ldots, x_m{:}T_m \vdash P}{\Gamma \vdash \mathsf{let}\,(\,x_1\,..\,x_m\,) = v\ \mathsf{in}\ P}$$

$$\text{(T-Case)}\ \frac{\Gamma \vdash v : [\,\ell_1{:}T_1\,; \ldots ; \ell_m{:}T_m\,] \qquad \Gamma, x_i{:}T_i \vdash P_i \quad \forall i \in 1\,..\,m}{\Gamma \vdash \mathsf{case}\ v\ \mathsf{of}\ \ell_1\_(x_1){:}P_1\,; \ldots ; \ell_m\_(x_m){:}P_m}$$

Table 2: Typing for Values and Processes

$$(\textsc{Inp})\ \frac{-}{c(x).P \xrightarrow{cv} P\{^v\!/_x\}} \qquad (\textsc{Rep})\ \frac{-}{!\,c(x).P \xrightarrow{cv} P\{v/x\} \mid !\,c(x).P}$$

$$(\textsc{Out})\ \frac{-}{\overline{c}v \xrightarrow{\overline{c}v} \mathbf{0}} \qquad (\textsc{Open})\ \frac{P \xrightarrow{(\boldsymbol{\nu}\widetilde{q}:\widetilde{T})\,\overline{c}v} P' \quad n \in \mathrm{n}(v)\backslash\{\widetilde{q}, c\}}{(\boldsymbol{\nu}n{:}T)\,P \xrightarrow{(\boldsymbol{\nu}n:T,\widetilde{q}:\widetilde{T})\,\overline{c}v} P'}$$

$$(\textsc{Com})\ \frac{P_1 \xrightarrow{(\boldsymbol{\nu}\widetilde{q}:\widetilde{T})\,\overline{c}v} P_1' \qquad P_2 \xrightarrow{cv} P_2' \qquad \widetilde{q} \cap \mathrm{fn}(P_2) = \emptyset}{P_1 \mid P_2 \xrightarrow{\tau} (\boldsymbol{\nu}\widetilde{q}{:}\widetilde{T})\,(P_1' \mid P_2')}$$

$$(\textsc{Par})\ \frac{P_1 \xrightarrow{\mu} P_1' \qquad \mathrm{bn}(\mu) \cap \mathrm{fn}(P_2) = \emptyset}{P_1 \mid P_2 \xrightarrow{\mu} P_1' \mid P_2}$$

$$(\textsc{Res})\ \frac{P \xrightarrow{\mu} P' \qquad n \notin \mathrm{n}(\mu)}{(\boldsymbol{\nu}n{:}T)\,P \xrightarrow{\mu} (\boldsymbol{\nu}n{:}T)\,P'}$$

$$(\textsc{Test-1})\ \frac{P_1 \xrightarrow{\mu} P_1' \qquad k_1 = k}{\mathsf{if}\ [k{=}k_1]\ \mathsf{then}\ P_1\ \mathsf{elif}\ [k{=}k_2]\ \mathsf{then}\ P_2\ \mathsf{else}\ P_3 \xrightarrow{\mu} P_1'}$$

$$(\textsc{Test-2})\ \frac{P_2 \xrightarrow{\mu} P_2' \qquad k_1 \neq k = k_2}{\mathsf{if}\ [k{=}k_1]\ \mathsf{then}\ P_1\ \mathsf{elif}\ [k{=}k_2]\ \mathsf{then}\ P_2\ \mathsf{else}\ P_3 \xrightarrow{\mu} P_2'}$$

$$(\textsc{Test-3})\ \frac{P_3 \xrightarrow{\mu} P_3' \qquad k_1 \neq k \neq k_2}{\mathsf{if}\ [k{=}k_1]\ \mathsf{then}\ P_1\ \mathsf{elif}\ [k{=}k_2]\ \mathsf{then}\ P_2\ \mathsf{else}\ P_3 \xrightarrow{\mu} P_3'}$$

$$(\textsc{Case})\ \frac{P_j\{^v\!/_{x_j}\} \xrightarrow{\mu} Q \qquad j \in 1\,..\,m}{\mathsf{case}\ \ell_j\_v\ \mathsf{of}\ \ell_1\_(x_1){:}P_1\,;\ldots;\ell_m\_(x_m){:}P_m \xrightarrow{\mu} Q}$$

$$(\textsc{Let})\ \frac{P\{v_1\,..\,v_m/x_1\,..\,x_m\} \xrightarrow{\mu} Q}{\mathsf{let}\,(\,x_1\,..\,x_m\,) = \langle\,v_1\,..\,v_m\,\rangle\ \mathsf{in}\ P \xrightarrow{\mu} Q}$$

Table 3: Labelled Transition System for $\mathrm{L}\pi^+$.

2.2 Operational and Behavioural semantics

Table 3 shows the transition rules for $\mathrm{L}\pi^+$ in an *early* style; the symmetric rules of (Com) and (Par) are omitted. *Labelled transitions* are of the form $P \xrightarrow{\mu} P'$, where *action* μ is: τ (interaction), cv (free input), $(\boldsymbol{\nu}\widetilde{n}{:}\widetilde{T})\,\overline{c}v$ (output at c of value v containing private names $\widetilde{n}$ of type $\widetilde{T}$, which we often omit), where c is the *subject* and v the *object*. The functions $\mathrm{fn}(\cdot)$, $\mathrm{bn}(\cdot)$, $\mathrm{n}(\cdot)$, $\mathrm{fc}(\cdot)$, $\mathrm{bc}(\cdot)$, and $\mathrm{c}(\cdot)$ are extended to actions as usual. Relation $\Rightarrow$ is the reflexive-transitive closure of $\xrightarrow{\tau}$; $\xRightarrow{\mu}$ denotes $\Rightarrow\xrightarrow{\mu}\Rightarrow$; $\xRightarrow{\hat{\mu}}$ denotes $\xRightarrow{\mu}$ if $\mu \neq \tau$, and $\Rightarrow$ if $\mu = \tau$. For any relation $\mathcal{R}$ on processes, $\xrightarrow{\tau}_{\mathcal{R}}$ denotes $\mathcal{R}\xrightarrow{\tau}\mathcal{R}$, and $\Rightarrow_{\mathcal{R}}$ the reflexive-transitive closure of $\xrightarrow{\tau}_{\mathcal{R}}$.

The typing in Table 2 is preserved under τ-actions, which are also called *reductions*.

Theorem 2.2 (Type Soundness) *Let Γ be a closed type environment.*

1. *If $\Gamma \vdash P$ then $P \not\equiv Q$ where Q contains* wrong.
2. *If $\Gamma \vdash P$ and $P \Rightarrow Q$, then $\Gamma \vdash Q$.*

The proof of the above result is standard (see for instance [San98]).

A crucial notion in a process calculus is that of *behavioural equality* between processes. We focus on bisimulation-based behavioural equivalences, precisely on (weak) *barbed bisimulation* [MS92]. Barbed bisimulation can be defined in any calculus possessing: (i) an *interaction relation* (the τ-steps in the π-calculus), modelling the evolution of the system; and (ii) an *observability predicate* $\downarrow_c$ for each channel c, to detect the possibility of a process to accept a communication with the environment at c. We recall that in asynchronous calculi only output actions are observed [ACS98] because the environment has no direct way of knowing if the message it has sent has been received.

Definition 2.3 (Asynchronous observability) *We write $P\downarrow_c$ if there is a derivative P', and an output action μ with subject c, such that $P \xrightarrow{\mu} P'$. We write $P \Downarrow_c$ if there is P' such that $P \Rightarrow P'$ and $P' \downarrow_c$.*

Definition 2.4 (Barbed bisimilarity) *A symmetric relation $\mathcal{S}$ on processes is an* barbed bisimulation *if $P \;\mathcal{S}\; Q$ implies:*

- *If $P \xrightarrow{\tau} P'$ then there exists Q' such that $Q \Longrightarrow Q'$ and $P' \;\mathcal{S}\; Q'$.*
- *If $P\downarrow_c$ then $Q \Downarrow_c$.*

Two processes P and Q are barbed bisimilar*, written $P \dot{\cong} Q$, if $P \;\mathcal{S}\; Q$ for some barbed bisimulation $\mathcal{S}$.*

Barbed bisimilarity equips a global observer with a minimal ability to observe actions and/or process states but it is not a congruence. By closing barbed bisimilarity under *contexts* we obtain a much finer relation. Since $\mathrm{L}\pi^{+}$ is a typed calculus, only well-typed contexts should be considered [PS96, SW01].

Definition 2.5 (Context) *A (monadic)* context *$C[\cdot]$ is a process expression with a single hole in it, written $[\cdot]$. Given a process P, $C[P]$ is the process obtained by plugging the process P into the hole. A context $C[\cdot]$ is* static *if it is structurally equivalent to $(\boldsymbol{\nu}\widetilde{n})\,(P \mid [\cdot])$, for some P and $\widetilde{n}$.*

Definition 2.6 *Let Γ and Δ be two type environments. We say that Γ* extends *Δ if $\mathtt{dom}(\Delta) \subseteq \mathtt{dom}(\Gamma)$ and $\Gamma \vdash n{:}\Delta(n)$ for all names n on which Δ is defined. We say that Γ is a* closed extension *of Δ if Γ is closed and extends Δ.*

Definition 2.7 *Let Γ and Δ be two type environments. We say that $C[\cdot]$ is a (Δ/Γ)-context if $\Delta \vdash C[\cdot]$ is a valid type judgement when the hole $[\cdot]$ is considered as a process and the following typing rule for $[\cdot]$ is added:*

$$(\textsc{t-hole})\ \frac{\Theta \text{ extends } \Gamma}{\Theta \vdash [\cdot]}$$

(in the rule, Γ is one of the given type environments and Θ is a metavariable over type environments).

Definition 2.8 (Typed barbed relations) *Let Γ be a typing, and P and Q two processes such that $\Gamma \vdash P, Q$. We say that P and Q are* barbed Γ-equivalent*, written $P \simeq_\Gamma Q$, if for each closed type environment Δ and static (Δ/Γ)-context $C[\cdot]$, we have $C[P] \dot{\cong} C[Q]$. We say that P and Q are* barbed Γ-congruent*, written $P \cong_\Gamma Q$, if for each closed type environment Δ and (Δ/Γ)-context $C[\cdot]$, we have $C[P] \dot{\cong} C[Q]$.*

Context-based behavioural equalities like barbed congruence suffer from the universal quantification on contexts. Simpler proof techniques are based on *labelled bisimulations* whose definitions do not use context quantification. These bisimulations should imply, or (better) coincide with, barbed congruence. Labelled bisimilarities for typed barbed relations must take into account types. A *typed relation* is a set of triples $(\Delta; P; Q)$ where Δ is a closed typing and $\Delta \vdash P, Q$. Below, we give a typed variant of Amadio, Castellani, and Sangiorgi's *asynchronous bisimilarity* [ACS98].

Definition 2.9 (Typed bisimilarity) Typed bisimilarity, *is the largest typed relation $\mathcal{S}$ such that $(\Delta; P; Q) \in \mathcal{S}$ implies:*

1. *If $P \xrightarrow{\tau} P'$, then there exists Q' s.t. $Q \Rightarrow Q'$ and $(\Delta; P'; Q') \in \mathcal{S}$.*
2. *If $P \xrightarrow{(\boldsymbol{\nu}\widetilde{n}:\widetilde{T})\,\overline{c}v} P'$, with $\widetilde{n} \cap \mathrm{fn}(Q) = \emptyset$, then there exists Q' such that $Q \xRightarrow{(\boldsymbol{\nu}\widetilde{n}:\widetilde{T})\,\overline{c}v} Q'$ and $((\Delta, \widetilde{n}:\widetilde{T}); P'; Q') \in \mathcal{S}$.*
3. *If*

   (i) Γ is a closed extension of Δ,
   (ii) $\Gamma \vdash c:\mathbf{C}(T)$ and $\Gamma \vdash v:T$,
   (iii) $P \xrightarrow{cv} P'$, with $\mathrm{fc}(v) \cap \mathrm{fc}(P \mid Q) = \emptyset$,

   then there exists Q' such that:

   (i) either $Q \xRightarrow{cv} Q'$ and $(\Gamma; P'; Q') \in \mathcal{S}$,
   (ii) or $Q \Rightarrow Q'$ and $(\Gamma; P'; (\,Q' \mid \overline{c}v\,)) \in \mathcal{S}$.

Let Γ be a closed typing with $\Gamma \vdash P, Q$. We say that P and Q are typed bisimilar at Γ, *written $P \approx_\Gamma Q$, if $(\Gamma; P; Q)$ is contained in typed bisimilarity.*

The bisimilarity above is *early* on keys and *ground* on channels. Indeed, in the input clause, there is an implicit universal quantification on the received keys, whereas we always assume to receive fresh channels by requiring $\mathrm{fc}(v) \cap \mathrm{fc}(P \mid Q) = \emptyset$. In asynchronous calculi without name testing, ground and early bisimilarity coincide [San00, Hon92]. Since we only have testing on keys (i) it makes sense to have the simpler ground clause on channels, (ii) our bisimilarity coincides with its (channel) early variant in which the requirement $\mathrm{fc}(v) \cap \mathrm{fc}(P \mid Q) = \emptyset$, in the input clause, is omitted. The proof that this early variant is a congruence (on well-typed contexts) is essentially the same as that for untyped asynchronous early bisimilarity [ACS98]. As a consequence, $\approx_\Gamma$ implies $\simeq_\Gamma$ and therefore $\cong_\Gamma$.

Later on, we will work with processes containing channels which can be used by the environment only in output. We model this constraint as follows:

Definition 2.10 (Barbed $\Gamma;\mathcal{C}$-relations) *Let $\mathcal{C} \subseteq \mathbf{C}$.* Barbed $\mathcal{C}$-bisimilarity, *written $\dot{\cong}_\mathcal{C}$, is the largest symmetric relation on processes, such that $P \dot{\cong}_\mathcal{C} Q$ implies:*

1. *If $P \xrightarrow{\tau} P'$, then there exists Q' such that $Q \Rightarrow Q'$ and $P' \dot{\cong}_\mathcal{C} Q'$*
2. *If $P{\downarrow_c}$, with $c \notin \mathcal{C}$, then $Q{\Downarrow_c}$.*

Let Γ be a typing, and P and Q two processes such that $\Gamma \vdash P, Q$. We say that P and Q are barbed $\Gamma;\mathcal{C}$-equivalent, *written $P \simeq_{\Gamma;\mathcal{C}} Q$, if for each closed type environment Δ and static (Δ/Γ)-context $C[\cdot]$ not containing names in $\mathcal{C}$ in input position, we have $C[P] \dot{\cong}_\mathcal{C} C[Q]$. We say that P and Q are* barbed $\Gamma;\mathcal{C}$-congruent, *written $P \cong_{\Gamma;\mathcal{C}} Q$, if for each closed type environment Δ and (Δ/Γ)-context $C[\cdot]$ not containing names in $\mathcal{C}$ in input position, we have $C[P] \dot{\cong}_\mathcal{C} C[Q]$.*

Roughly, $\mathcal{C}$ denotes the set of channels which cannot be used in input by the environment. In Definition 2.10, when $\mathcal{C}{=}\emptyset$, we get the standard definitions of typed barbed bisimilarity. If $\mathcal{C}{=}\{s\}$, as abbreviations, we write $\cong_{\Gamma;s}$ for $\cong_{\Gamma;\mathcal{C}}$ and $\simeq_{\Gamma;s}$ for $\simeq_{\Gamma;\mathcal{C}}$. Due to the restriction on the contexts, it holds that $\overline{s}v \cong_{\Gamma;s} \mathbf{0}$ and, by asynchrony, $s(x).\mathbf{0} \cong_{\Gamma;s} \mathbf{0}$. Below, we give the labelled counterpart of barbed $\Gamma;\mathcal{C}$-equivalence.

Definition 2.11 (Typed $\mathcal{C}$-bisimilarity) Typed $\mathcal{C}$-bisimilarity *is the largest typed relation $\mathcal{S}$ such that $(\Delta; P; Q) \in \mathcal{S}$ implies:*

1. *If $P \xrightarrow{\tau} P'$, then there exists Q' s.t. $Q \Rightarrow Q'$ and $(\Delta; P'; Q') \in \mathcal{S}$.*
2. *If $P \xrightarrow{(\boldsymbol{\nu}\widetilde{n}:\widetilde{T})\,\overline{c}v} P'$, with $c \notin \mathcal{C}$ and $\widetilde{n} \cap \mathrm{fn}(Q) = \emptyset$, then there exists Q' such that $Q \xRightarrow{(\boldsymbol{\nu}\widetilde{n}:\widetilde{T})\,\overline{c}v} Q'$ and $((\Delta, \widetilde{n}:\widetilde{T}); P'; Q') \in \mathcal{S}$.*
3. *If*

(i) Γ *is a closed extension of* Δ,
(ii) $\Gamma \vdash c{:}\mathbf{C}(T)$ *and* $\Gamma \vdash v{:}T$,
(iii) $P \xrightarrow{cv} P'$, *with* $\mathrm{fc}(v) \cap \mathrm{fc}(P \mid Q) = \emptyset$,

then there exists Q' *such that:*

(i) either $Q \overset{cv}{\Longrightarrow} Q'$ *and* $(\Gamma; P'; Q') \in \mathcal{S}$,
(ii) or $Q \Rightarrow Q'$ *and* $(\Gamma; P'; (Q' \mid \overline{c}v)) \in \mathcal{S}$.

Let Γ *be a closed typing with* $\Gamma \vdash P, Q$. *We say that* P *and* Q *are* typed $\mathcal{C}$-bisimilar at Γ, *written* $P \approx_{\Gamma;\mathcal{C}} Q$, *if* $(\Gamma; P; Q)$ *is contained in typed* $\mathcal{C}$*-bisimilarity.*

When $\mathcal{C}=\{s\}$, for some channel s, we abbreviate $\approx_{\Gamma;\mathcal{C}}$ with $\approx_{\Gamma;s}$.

Theorem 2.12 *Let* Γ *be a type environment,* $\mathcal{C}$ *a set of channels, and* P *and* Q *two processes such that* $\Gamma \vdash P, Q$. *Then,* $P \approx_{\Gamma;\mathcal{C}} Q$ *implies* $P \simeq_{\Gamma;\mathcal{C}} Q$.

PROOF. [Sketch] We have to prove that $\approx_{\Gamma;\mathcal{C}}$ is preserved by well-typed static contexts. Since $\mathrm{L}\pi^+$ is an asynchronous calculus without testing on channels, $\approx_{\Gamma;\mathcal{C}}$ coincides with its early variant where the requirement $\mathrm{fc}(v) \cap \mathrm{fc}(P \mid Q) = \emptyset$, in the input clause, is omitted. The proof that this (early) variant is preserved by parallel composition and restriction is standard (parallel composition require some care because the processes in parallel must not contain input along channels in $\mathcal{C}$). So, also $\approx_{\Gamma;\mathcal{C}}$ is preserved by parallel composition and restriction. Since $\approx_{\Gamma;\mathcal{C}}$ implies $\dot{\cong}$, it follows that $\approx_{\Gamma;\mathcal{C}} \subseteq \simeq_{\Gamma;\mathcal{C}}$. □

It is easy to prove that $\approx_\Gamma$ implies $\approx_{\Gamma;\mathcal{C}}$ and $\simeq_\Gamma$ implies $\simeq_{\Gamma;\mathcal{C}}$.

Finally, in Lemma 2.15 we give an algebraic law which will be used to prove one of the crucial results of the paper (Theorem 8.1). This law is based on special processes called *link* that behave as name buffers receiving values at one end and retransmitting them at the other end (in the π-calculus literature, links are sometimes called forwarders [HY95] or wires [SW01]). A similar law has already been used in a typed π-calculus with the name discipline of *uniform receptiveness* [San99a].

Definition 2.13 (Link) *Given two channels* p *and* q *with* $\Gamma \vdash p, q : \mathbf{C}(T)$, *we call* link *the process* $!\, p(u).\overline{q}u$, *abbreviated* $p \triangleright q$.

In order to prove Lemma 2.15, we need the following technical lemma.

Lemma 2.14 *Let* p *and* q *be two channels,* Q *a process in which* q *may only appear in output position, and* Γ *a type environment such that* $\Gamma \vdash Q$ *and* $\Gamma \vdash p, q{:}\mathbf{C}(T)$. *Then*

$$Q\{p/q\} \dot{\cong} (\boldsymbol{\nu} q{:}\mathbf{C}(T))\,(Q \mid q \triangleright p).$$

PROOF. See the proof in Appendix A.1. □

Lemma 2.15 *Let* $\Gamma \vdash \overline{p}v$, *for some type environment* Γ. *Let* $q \in \mathrm{fc}(v)$ *with* $\Gamma \vdash q{:}\mathbf{C}(T)$. *Let* $r \notin \mathrm{c}(v)$ *and* $w = v\{{}^{r}\!/_{q}\}$. *Then*

$$\overline{p}v \cong_\Gamma (\boldsymbol{\nu} r{:}\mathbf{C}(T))\,(\overline{p}w \mid r \triangleright q).$$

PROOF. We prove that for any well-typed context $C[\cdot]$, it holds that:

$$C[\overline{p}v] \dot{\cong} C[(\boldsymbol{\nu} r{:}\mathbf{C}(T))\,(\overline{p}w \mid r \triangleright q)].$$

The prove is by structural induction on the context $C[\cdot]$. The most interesting case is when $C[\cdot] \equiv [\cdot] \mid R$ for some process R. So, in order to prove that

$$\overline{p}v \mid R \dot{\cong} (\boldsymbol{\nu} r{:}\mathbf{C}(T))\,(\overline{p}w \mid r \triangleright q) \mid R$$

a, b	$::= \mathbb{O}$	object
	$\mid a.\mathrm{l}\langle a_1 \mathinner{..} a_n \rangle$	method invocation
	$\mid a.\mathrm{l}\Leftarrow m$	method update
	$\mid a.\mathsf{clone}$	shallow copy
	$\mid a.\mathsf{alias}\langle b\rangle$	object aliasing
	$\mid a.\mathsf{surrogate}$	object surrogation
	$\mid a.\mathsf{ping}$	object ping
	$\mid s, x, y, z$	variables
	$\mid \mathsf{let}\, x{:}A = a\, \mathsf{in}\, b$	local definition
	$\mid \mathsf{fork}\langle a\rangle$	thread creation
	$\mid \mathsf{join}\langle a\rangle$	thread destruction
$\mathbb{O}$	$::= [\mathrm{l}_j{=}m_j]_{j\in J}$	object record
m_j	$::= \varsigma(s_j{:}A, \tilde{x}_j{:}\widetilde{B}_j)b_j$	method
A, B	$::= [\mathrm{l}_j{:}\widetilde{B}_j{\to}\widehat{B}_j]_{j\in J}$	object record type
	$\mid \mathrm{Thr}(A)$	thread type

Table 4: Øjeblik Syntax and Types

we show that the relation

$$\mathcal{S} = \{ \big(\overline{p}v \mid R \,,\ (\boldsymbol{\nu} r{:}\mathbf{C}(T))\, (\overline{p}w \mid r \triangleright q) \mid R \big) \} \cup \dot{\cong}$$

is a barbed bisimulation up to $\equiv$. The requirements on the barbs are easily satisfied. As for the bisimulation game on silent moves, the only interesting case is when there is a communication along p, that is, when $R \xrightarrow{p(x)} R'$. In this case we get, up to structural equivalence, the pair of processes

$$\big(Q\{q/r\} \,,\ (\boldsymbol{\nu} r{:}\mathbf{C}(T))\, (Q \mid r \triangleright q) \big)$$

where $Q = R'\{w/x\}$. By Lemma 2.14 we can conclude. □

3 Øjeblik: A Concurrent Object Calculus

In this section, we present Øjeblik [NHKM00], a typed abstraction of Obliq designed to study object migration. Øjeblik-expressions and Øjeblik-types are generated by the grammar in Table 4, where a ranges over *Øjeblik-terms*, l over *method labels*, m over *method bodies*, s, x, y, z over *variables*, $\mathbb{O}$ over *object records*, and A, B over types. The type language extends the one of the imperative object calculus [AC96] by thread types $\mathrm{Thr}(A)$. Pairs $\tilde{x}_j{:}\widetilde{B}_j$ denote sequences $x_{1_j}{:}B_{1_j} \mathinner{..} x_{n_j}{:}B_{n_j}$. Function types $A{\to}B$ do only occur in object types $[\mathrm{l}_j{:}\widetilde{B}_j{\to}\widehat{B}_j]_{j\in J}$, so they are not first-class types. Yet, we sometimes abbreviate such object types by $[\mathrm{l}_j{:}A_j]_{j\in J}$ to clarify that a type is not a thread type. Typed terms are defined by adding type annotations to all binding occurrences of variables: in let-expressions and in method declarations.

For the sake of simplicity, compared to Obliq, in Øjeblik we omit ground values (like numbers, booleans, strings, etc.), data operations, and procedures, we restrict field selection to method invocation, we restrict multiple cloning to single cloning, we omit flexibility of object attributes, we replace field aliasing with object aliasing, we omit explicit distribution, and we omit exceptions and advanced synchronisation, so we get a feasible, but still non-trivial language. As in Obliq, computation follows the call-by-value evaluation order. In particular, in the following, whenever we use a term a, we implicitly assume that we have first evaluated a to some actual value, i.e. in most cases to an object reference.

Objects

An object record $[\mathrm{l}_j{=}m_j]_{j\in J}$ is a finite collection of updatable named methods $\mathrm{l}_j{=}m_j$, for pairwise distinct labels l_j. In a method $\varsigma(s,\tilde{x})b$, the letter ς denotes a binder for the self variable s and argument variables $\tilde{x}$ within the body b. Moreover, every object in Øjeblik comes equipped with special methods for *cloning*, *aliasing*, *surrogation*, and *ping*, which cannot be overwritten by the *update* operation.

Method invocation $a.\mathrm{l}\langle\,\tilde{c}\,\rangle$ with field l of the object a containing the method $\varsigma(s,\tilde{x})b$ results in the body b with the self variable s replaced by (a reference to) the enclosing object a, and the formal parameters $\tilde{x}$ replaced by (references to) the actual parameters $\tilde{c}$ of the invocation.

Method update $a.\mathrm{l}{\Leftarrow}m$ overwrites the current content of the named field l in object a with method m and returns a reference to the modified object.

The *clone operation* $a.\mathsf{clone}$ creates a clone a' of a and returns a reference to a'.

The operation $a.\mathsf{alias}\langle b\rangle$ replaces object a with an alias to b, written $a{\gg}b$, regardless of whether a is already an alias or still an object record; if b itself is an alias, e.g. $b{\gg}c$, then we consequently and naturally create an *alias chain* $a{\gg}b{\gg}c$. From the computational point of view, requests arriving at a after the operation $a.\mathsf{alias}\langle b\rangle$ should be forwarded to b. The operation $a.\mathsf{alias}\langle b\rangle$ returns a reference to b.

The operation $a.\mathsf{surrogate}$ represents our *abstraction of migration*: by calling it, object a is turned into a proxy for a copy of itself. Surrogation is implemented by providing a uniform method $\mathsf{surrogate}{=}\varsigma(s)s.\mathsf{alias}\langle s.\mathsf{clone}\rangle$. It returns a reference to the just created clone. Behaving like standard methods, surrogation is forwarded by aliased objects. This is necessary to correctly mimic migration: an object should be surrogatable more than once, so double-surrogation $a.\mathsf{surrogate}; a.\mathsf{surrogate}$ (where ; denotes sequential composition, as defined below) should be equivalent to $a.\mathsf{surrogate}.\mathsf{surrogate}$. Without forwarding, the surrogation of an already surrogated object would mistakenly surrogate the proxy.

The operation $a.\mathsf{ping}$ is implemented by providing a uniform method: $\mathsf{ping}{=}\varsigma(s)s$. Thus, $a.\mathsf{ping}$ returns the "identity" of the object o resulting from the evaluation of a; note that, due to aliasing and forwarding, this could be the "identity" of the current endpoint of an alias chain potentially starting at object o. We add the $a.\mathsf{ping}$ method uniformly to Øjeblik objects because it allows us to conveniently express the safety of surrogation/migration as an algebraic equation. Furthermore, such a method could be used by clients for garbage collection of references to surrogated servers by interrogating the current identity and using it directly instead of the former indirect reference.

Scoping

Apart from the binding of variables in method bodies, Øjeblik also offers explicit scope declarations. An expression $\mathsf{let}\,x = a\,\mathsf{in}\,b$ first evaluates a, binding the result to x, and then evaluates b within the scope of the new binding. We use the standard inductive definition $\mathrm{fv}(a)$ to denote the free variables of term a with respect to our two forms of binding. Øjeblik only admits non-recursive expressions $\mathsf{let}\,x = a\,\mathsf{in}\,b$, i.e., with $x \notin \mathrm{fv}(a)$. Then, $a; b$ denotes $\mathsf{let}\,x = a\,\mathsf{in}\,b$, where $x \notin \mathrm{fv}(b)$.

Concurrency

While objects represent persistent stateful structural entities, computational activity takes place within *threads*. In addition to the main thread that is initially started up with the execution of a term, new separate threads can be created by the fork command. The term $\mathsf{fork}\langle a\rangle$ returns a new thread identifier to denote the thread evaluating a. The result of a fork'ed computation is grabbed by the join command. If a evaluates to a thread identifier, then $\mathsf{join}\langle a\rangle$ potentially blocks until that thread finishes and returns the thread's result, or blocks forever, if a join on thread a was already performed earlier.

Self-Infliction

The *current method* of a thread is the last method invoked in it that has not yet completed. The *current self* of a thread is the self of its current method. An Øjeblik operation is *self-inflicted*, also called *internal*, if it addresses the current self; an operation is *external* if it is not self-inflicted. However, self-inflicted operations can be invoked from within methods not only literally on the self variable s, but also indirectly by an expression that evaluates to the object itself. For instance, in

$$\mathsf{let}\,x = [\,\mathrm{l}{=}\varsigma(s,z)z.\mathsf{clone}\,]\,\mathsf{in}\,x.\mathrm{l}\langle x\rangle$$

the call $z.\mathsf{clone}$ will be self-inflicted when it is finally executed.

Based on the concept of self-infliction, Obliq, and therefore also our abstraction Øjeblik, supports the notions of serialisation and protection of objects.

Serialisation In concurrent object-based settings, the invariant that at most one thread at a time may be active within an object is often called *serialisation*. The simplest way to ensure serialisation is to associate with an object a *mutex* that is locked when a thread enters the object and released when the thread exits the object. However, this approach is too restrictive, for instance, it prevents recursion. Based on the notion of *thread*, so-called *reentrant* mutexes, as in Java, can be used to allow an operation to re-enter an object under the assumption that this operation belongs to the same thread as the operation that is currently active in the object. In Obliq, however, the more cautious idea of *self-serialisation* requires, based on the above notion of self-infliction, that the mutex is always acquired for external operations, but never for self-inflicted ones. Note that this concept allows a method to recursively call its siblings through self, but it excludes the kind of inter-object mutual recursion, where a method in an object a calls a method in another object b, which then tries to 'call back' another method in a.

Protection Based on self-infliction, objects are *protected* against external modifications in a natural way: updates, cloning, and aliasing are only allowed if these operations are self-inflicted.

In Obliq, object migration is supposed to be correct only for both protected and serialised objects. So, since we are interested in proving the safety of object migration, *all objects in (our abstraction) Øjeblik are both protected and serialised.*

Finally, in Table 5, we present the rules for static typing. The typing rules themselves are not surprising. The operations clone, alias, surrogate, ping, and update, all yield a result of the same type as the object that they address. While fork packs a type into a thread type, join unpacks it accordingly. The rules for variables, let, and objects, and invocations are standard. The usual properties hold, e.g., the free variables of a term are all captured by the type environment.

As for our type system for the π-calculus, all the standard properties of *weakening*, *contraction*, *substitution*, and *narrowing* hold for the typings in Table 5.

4 Towards a formal semantics for Øjeblik

Although, at first sight, the informal semantics of Øjeblik is reasonably clear, its formalisation requires one to consider even the slightest detail. In particular, the behavior of alias chains (that is chains of alias nodes), needs to be clearly spelled out. In our previous work [NHKM00, Mer00], we already showed that the semantics of alias nodes, as implemented in Cardelli's Obliq, gives rise to an incorrect behavior of object migration. Roughly, the reason is because, in Obliq, alias nodes support a too strong form of both protection and serialization. As a consequence, in [NHKM00, Mer00] we proposed a variant of Cardelli's semantics in which alias nodes have a milder form of both protection and serialization. In this section, we explain our proposed semantics and prepare the ground for its formal definition in terms of π-calculus. In Sections 4.1 and 4.2 we first explain a few general concepts about alias chains, then in Section 4.3 we show the design choices for our

$$\text{(T-Var)}\ \frac{\Gamma(x)=A}{\Gamma\vdash x{:}A} \qquad \text{(T-Let)}\ \frac{\Gamma\vdash a{:}A \qquad \Gamma,x{:}A\vdash b{:}B}{\Gamma\vdash \mathsf{let}\,x{:}A=a\,\mathsf{in}\,b : B}$$

$$\text{(T-Fork)}\ \frac{\Gamma\vdash a{:}A}{\Gamma\vdash \mathsf{fork}\langle a\rangle : \mathrm{Thr}(A)} \qquad \text{(T-Join)}\ \frac{\Gamma\vdash a : \mathrm{Thr}(A)}{\Gamma\vdash \mathsf{join}\langle a\rangle : A}$$

$$\text{(T-Obj)}\ \frac{\forall j{\in}J \quad \Gamma,s_j{:}A,\tilde{x}_j{:}\widetilde{B}_j\vdash b_j{:}\widehat{B}_j \qquad A=[\mathrm{l}_j{:}\widetilde{B}_j{\rightarrow}\widehat{B}_j]_{j\in J}}{\Gamma\vdash [\mathrm{l}_j{=}\varsigma(s_j{:}A,\tilde{x}_j{:}\widetilde{B}_j)b_j]_{j\in J} : A}$$

$$\text{(T-Inv)}\ \frac{\Gamma\vdash a : [\mathrm{l}_j{:}\widetilde{B}_j{\rightarrow}\widehat{B}_j]_{j\in J} \qquad \Gamma\vdash \widetilde{b}_k{:}\widetilde{B}_k \qquad k{\in}J}{\Gamma\vdash a.\mathrm{l}_k\langle\,\widetilde{b}_k\,\rangle : \widehat{B}_k}$$

$$\text{(T-Upd)}\ \frac{\Gamma\vdash a{:}A \qquad A=[\mathrm{l}_j{:}\widetilde{B}_j{\rightarrow}\widehat{B}_j]_{j\in J} \qquad \Gamma,s{:}A,\tilde{x}{:}\widetilde{B}_k\vdash b{:}\widehat{B}_k \qquad k{\in}J}{\Gamma\vdash a.\mathrm{l}_k{\Leftarrow}\varsigma(s{:}A,\tilde{x}{:}\widetilde{B}_k)b : A}$$

$$\text{(T-Ping)}\ \frac{\Gamma\vdash a{:}A \qquad A=[\mathrm{l}_j{:}A_j]_{j\in J}}{\Gamma\vdash a.\mathsf{ping} : A}$$

$$\text{(T-Clo)}\ \frac{\Gamma\vdash a{:}A \qquad A=[\mathrm{l}_j{:}A_j]_{j\in J}}{\Gamma\vdash a.\mathsf{clone} : A}$$

$$\text{(T-Ali)}\ \frac{\Gamma\vdash a,b{:}A \qquad A=[\mathrm{l}_j{:}A_j]_{j\in J}}{\Gamma\vdash a.\mathsf{alias}\langle b\rangle : A}$$

$$\text{(T-Sur)}\ \frac{\Gamma\vdash a{:}A \qquad A=[\mathrm{l}_j{:}A_j]_{j\in J}}{\Gamma\vdash a.\mathsf{surrogate} : A}$$

Table 5: Typing Rules for Øjeblik

semantics of alias nodes. We address the reader to [NHKM00, Mer00] for a full explanation about the differences between our aliasing semantics and Cardelli's original one.

4.1 On the stability of alias chains

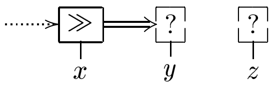
As a matter of fact, according to the operations' character with respect to self-infliction and the intended node of action, a node x in an alias chain can be *unstable*, which means that if it currently points to node y, it may later on point to a different node z. In order to clarify this phenomenon, we distinguish two cases based on the notion of a *task*, which is the run-time entity that is created by method invocation within a single object. A thread may then actually be seen as a stack of tasks connected via invocations. Now, a node can be *active*, in which case it contains running *tasks*, or not. The punchline of this subsection is then that an *alias node* can not become stable before it has terminated its current tasks.

Below, we introduce pictures where we use *single/double* boxes to denote *inactive/active* nodes, respectively, and *single/double* arrows to denote *unstable/stable* aliases, respectively. Furthermore, *dashed* boxes and *dotted* arrows denote unspecified respective entities.

Inactive Nodes: No Tasks By definition, the only way to receive a self-inflicted request is to have already at least one local task running. In other words, if there is no local task, then each incoming request is doomed to be external. Now, let us focus on the example term:

$$\begin{array}{l}\mathsf{let}\ z = [\,\mathrm{l}{=}\ \textit{“bar”}\,]\ \mathsf{in}\\ \quad \mathsf{let}\ y = [\,\mathrm{l}{=}\ \textit{“foo”}\,]\ \mathsf{in}\\ \qquad \mathsf{let}\ x = [\,\mathrm{l}{=}\varsigma(s,w)s.\mathsf{alias}\langle w\rangle\,]\ \mathsf{in}\ x.\mathrm{l}\langle\, y\,\rangle; x.\mathrm{l}\langle\, z\,\rangle\end{array}$$

after it carried out the invocation $x.\mathrm{l}\langle\, y\,\rangle$, that is, when the object referred to by x has turned itself into an alias for y and then terminated its activity. We depict the situation as follows

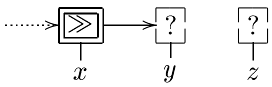


where, in general, the node x may itself be referred to by other aliases, while y and z may be either an alias or an object record. In fact, the alias $x{\gg}y$ is *stable* in the very sense: no re-aliasing operation on x to another node will ever possibly take place since it could only be carried out in a self-inflicted way by one of its own methods, but any request to such a method potentially starting such a self-inflicted operation, e.g., by calling $x.\mathrm{l}\langle\, z\,\rangle$, is itself forwarded to y such that it can never take place in x.

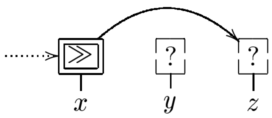
Active nodes: at least one task As an example, let us first consider the term

$$\begin{array}{l}\mathsf{let}\ z = [\,\mathrm{l}{=}\ \textit{“bar”}\,]\ \mathsf{in}\\ \quad \mathsf{let}\ y = [\,\mathrm{l}{=}\ \textit{“foo”}\,]\ \mathsf{in}\\ \qquad \mathsf{let}\ x = [\,\mathrm{l}{=}\varsigma(s,w)s.\mathsf{alias}\langle w\rangle; \underline{\textit{“bla”}}\,]\ \mathsf{in}\ x.\mathrm{l}\langle\, y\,\rangle\end{array}$$

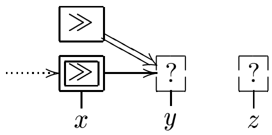
just after object x has accepted the request for method l and turned itself into an alias for y. Since x continues to operate on itself, according to *“bla”* in method l, x is an *active* alias node:



The alias $x{\gg}y$ is marked as *unstable* since *“bla”* may contain further self-inflicted requests, e.g., to perform a re-aliasing or a cloning. Thus, if *“bla”* calls $s.\mathsf{alias}\langle z\rangle$ or $s.\mathsf{clone}$, we get



or



and such changes may continue as long some current task in x is active. Here, the re-aliased x remains active, thus unstable, until all current tasks in x, in our example according to *“bla”*, have terminated. Note that the cloning of an active unstable alias $x{\gg}y$ provides a new inactive stable alias $x'{\gg}y$, because only the state of x is copied, not its tasks.

Generalising the above example, we may consider the case where several tasks of the current thread are running in an alias or an object. However, by the definition of synchronous method invocation, only one of them may be active—namely the one on top of the thread's call-stack, while the others must be blocked. Now, note that it is the active task or any of its ancestors in the call-stack who turned the current node into an alias (in the example it is method l); otherwise, the node would be stable and the current tasks would not exist, but have been created in one of the successors of the stable alias node.

4.2 Cyclic alias chains

Obliq does not prevent the programmer from (either consciously or accidentally) introducing, via substitution, self-aliases or alias chains with cycles. Consider the following example:

$$\mathsf{let}\ x = [\,\mathrm{k}{=}\mathrm{id}\,,\ \mathrm{l}{=}\varsigma(s,z)s.\mathsf{alias}\langle z\rangle\,]\ \mathsf{in}\ x.\mathrm{l}\langle x\rangle; x.\mathrm{k}\,.$$

By calling $x.\mathrm{l}\langle x \rangle$, the aliasing operation $x.\mathsf{alias}\langle x \rangle$ is carried out giving rise to the cyclic alias chain $x \gg x$. As a consequence, the following external method call $x.\mathrm{k}$ will give rise to a diverging computation.

4.3 On forwarding requests within alias nodes

In this section, we describe the behaviour of single alias nodes in Øjeblik by addressing four crucial questions.

1. *What* is the current self of forwarded requests?
2. *Who* is in charge of sending the result of a forwarded external request?
3. *When* does the forwarding take place?
4. *Which* requests are forwarded and which requests fail in an alias node?

Our semantics behaves as follows:

What? Let a be an alias node forwarding requests to b, that is, $a \gg b$. Let c be a third object invoking a method of a. Then, when serving the (external) request, the alias a simply forwards the request to b, and *c is still the current self.* Roughly speaking, it is as if c invokes directly a method of b. The self-inflicted case is trivial because then $a = c$.

Who? As above, let $a \gg b$ and c be a third object invoking a method of a. Since alias nodes simply forward requests unchanged, also *the transmission of the result of the request is delegated to b.* As a consequence: should the request in a have required a mutex, then the mutex can already be released once the request has been forwarded to b.

When? When addressed to stable alias nodes, incoming external requests do not have to wait until previously forwarded requests (there can only be external ones in this case) have successfully signalled termination from their point of action. However, when addressed to unstable alias nodes, incoming external requests must wait for the termination of previous (external and self-inflicted) requests.

Which? Protected external requests are supposed to fail only when addressed to non-aliased nodes, thus only in endpoints of alias chains.

- Method invocations (as well as pings and surrogations) are always forwarded (by transitivity to the endpoint of the chain, if it exists).
- Self-inflicted cloning and self-inflicted aliasing are performed at the alias node; external cloning and external aliasing are forwarded because they can possibly reach another node in the alias chain where they are self-inflicted and therefore executable.
- Self-inflicted update requests are forwarded. External update requests are forwarded because they may reach a (non-aliased) object that serves them.

5 A translational semantics for Øjeblik

In this section we give a *translational semantics* of Øjeblik into $\mathrm{L}\pi^+$ according to the informal semantics given in Sections 3 and 4. In addition to the syntax of $\mathrm{L}\pi^+$ we use standard abbreviations for:

- *polyadic input* $a(x_1 \mathinner{..} x_n).P \stackrel{\text{def}}{=} a(y).\mathsf{let}\,(\,x_1 \mathinner{..} x_n\,) = y\,\mathsf{in}\,P$ where $y \notin \mathrm{fn}(P)$. We will also write $\mathbf{C}(T_1 \mathinner{..} T_n)$ instead of $\mathbf{C}(\langle\, T_1 \mathinner{..} T_n \,\rangle)$ denoting the type of a channel carrying a tuple.
- *polyadic case destructor* $\ell\_(x_1 \mathinner{..} x_n){:}P \stackrel{\text{def}}{=} \ell\_(y){:}\mathsf{let}\,(\,x_1 \mathinner{..} x_n\,) = y\,\mathsf{in}\,P$, where $y \notin \mathrm{fn}(P)$;

$$
\begin{array}{rcl}
[\![\, a.\mathsf{clone} \,]\!]_p^k & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} q) \Big(\, [\![\, a \,]\!]_q^k \mid q(y,k') \,.\, \overline{y}\langle \mathsf{cln\_}p, k' \rangle \, \Big) \\
[\![\, a.\mathsf{alias}\langle b \rangle \,]\!]_p^k & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} q_x q_y) \Big(\, [\![\, a \,]\!]_{q_y}^k \mid q_y(y,k_y).([\![\, b \,]\!]_{q_x}^{k_y} \mid q_x(x,k_x) \,.\, \overline{y}\langle \mathsf{ali\_}\langle x,p \rangle, k_x \rangle) \, \Big) \\
[\![\, a.\mathrm{l}_j \Leftarrow \varsigma(s,\tilde{x})b \,]\!]_p^k & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} q) \Big([\![\, a \,]\!]_q^k \mid q(y,k').(\boldsymbol{\nu} t)\big(\, !\, t(s,\widetilde{x},r,k).[\![\, b \,]\!]_r^k \mid \overline{y}\langle \mathsf{upd}_j\mathsf{\_}\langle t,p \rangle, k' \rangle \big) \Big) \\
[\![\, a.\mathrm{l}_j \langle\, a_1 \,..\, a_n \,\rangle \,]\!]_p^k & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} q q_1 \cdots q_n) \Big(\, [\![\, a \,]\!]_q^k \mid q(y,k_0).([\![\, a_1 \,]\!]_{q_1}^{k_0} \mid q_1(x_1,k_1).([\![\, a_2 \,]\!]_{q_2}^{k_1} \mid \cdots \\
 & & \qquad q_n(x_n,k_n).\overline{y}\langle \mathsf{inv}_j\mathsf{\_}\langle x_1 \,..\, x_n, p \rangle, k_n \rangle \cdots)) \Big) \\
[\![\, a.\mathsf{surrogate} \,]\!]_p^k & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} q) \big(\, [\![\, a \,]\!]_q^k \mid q(y,k') \,.\, \overline{y}\langle \mathsf{sur\_}p, k' \rangle \, \big) \\
[\![\, a.\mathsf{ping} \,]\!]_p^k & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} q) \big(\, [\![\, a \,]\!]_q^k \mid q(y,k') \,.\, \overline{y}\langle \mathsf{png\_}p, k' \rangle \, \big) \\
\hline
[\![\, \mathsf{let}\, x = a \,\mathsf{in}\, b \,]\!]_p^k & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} q) \big(\, [\![\, a \,]\!]_q^k \mid q(x,k') \,.\, [\![\, b \,]\!]_p^{k'} \, \big) \\
[\![\, x \,]\!]_p^k & \stackrel{\text{def}}{=} & \overline{p}\langle x,k \rangle \\
\hline
[\![\, \mathsf{fork}(a) \,]\!]_p^k & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} q t) \big(\, [\![\, a \,]\!]_q^{\boldsymbol{\nu}} \mid \overline{p}\langle t,k \rangle \mid q(x,k').t(r,k'').\overline{r}\langle x,k'' \rangle \, \big) \\
[\![\, \mathsf{join}(b) \,]\!]_p^k & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} q) \big(\, [\![\, b \,]\!]_q^k \mid q(t,k') \,.\, \overline{t}\langle p,k' \rangle \, \big)
\end{array}
$$

Table 6: Translational semantics of Øjeblik — Clients, Scoping, Concurrency

- *parameterised recursive definitions* $\mathsf{A}(x_1 \,..\, x_n) \stackrel{\text{def}}{=} P$ and instantiation $\mathsf{A}\langle\, x_1 \,..\, x_n \,\rangle$, which can be faithfully represented in terms of replication [Mil93]. The typing rule associated with a recursive definition is the standard rule, requering the body to be well-typed under the assumption that the process name is well-typed.

The semantics, as presented in Tables 6 and 7 is a mapping parameterised on two names: in a term $[\![\, a \,]\!]_p^k$, the channel p is used to return the term's result, while the key k represents the term's current self, which is required to deal with self-infliction. In all phases of the translation, whenever we create $\boldsymbol{\nu}$- or input-bindings, we assume that there are no name-clashes. The essence of the semantics is to set up processes representing objects that serve clients' requests. Different requests for operating on objects are distinguished by corresponding labels cln, ali, upd_j, inv_j, png, and sur. We explain the semantics by showing how requests are generated by clients, and then how they are served by objects. Scoping and concurrency are explained along the way.

We present the translation without type annotations in restrictions for sake of readability. However, to make the translation formal such type annotations should be added. In Section 6.1 we present a translation of Øjeblik types to π-calculus types, that can be used to add the necessary type annotations to the translation of an object, based on the type of the object (see [KS98]).

Clients In Table 6, the current self k of encoded terms is 'used' as the current self of the evaluation of the first subterm in left-to-right evaluation order. All the translations in Table 6 follow a common scheme. For example, in the translation of a method invocation $[\![\, a.\mathrm{l}_\mathrm{j} \langle\, a_1 \,..\, a_n \,\rangle \,]\!]_p^k$, the subterms $a, a_1 \,..\, a_n$ have to be evaluated one after the other: the individual evaluations use private return channels $q, q_1 \,..\, q_n$, which are subsequently asked for the respective results $y, x_1 \,..\, x_n$, but also for the respective new current self $k, k_1 \,..\, k_n$ to be used by the next evaluation. After the last subterm a_n has returned its result, the accumulated information is used to send a suitable request with label inv_j on self-channel y of object a, also carrying the overall result channel p and the latest current self k_n. Thus, the responsibility to signal a result on p is passed on to the respective object waiting at y.

Scoping The semantics of let is analogous to [KS98] and represents the core of the call-by-value evaluation order in that first a is evaluated, and then b possibly using the value of a. Here, in addition, the evaluation of a passes on the current self k' to be used afterwards.

Concurrency To fork a thread means to create a new activity running in parallel with the current one(s), which is done using the parallel operator. Upon thread creation, a fresh key is created to become the forked thread's current self. Sometimes, we use $[\![a]\!]_q^{\boldsymbol{\nu}}$ to abbreviate $(\boldsymbol{\nu} k)\,([\![a]\!]_q^k)$. The term $[\![\mathsf{fork}\langle a\rangle]\!]_p^k$ immediately returns on p a private name t, which can be used to retrieve the value of a from the forked thread. Therefore, $[\![\mathsf{join}\langle b\rangle]\!]_p^k$ sends its own result channel p, together with its latest current self k', along the value of b.

Objects The semantics $[\![\mathbb{O}]\!]_p^k$ of an object $\mathbb{O} := [\mathrm{l}_j{=}\varsigma(s_j,\tilde{x}_j)b_j]_{j\in J}$, as shown in Table 7 (again along the style of [KS98]), consists of a message that returns the object's reference s together with the current self k on channel p, a composition of replicated processes that give access to the method bodies $[\![b_j]\!]_r^{k'}$, and a new object process $\mathsf{newO}_{\mathbb{O}}\langle\, s, \widetilde{t}\,\rangle$ that connects invocations on s from the outside to the method bodies, which are invoked by the trigger names $\widetilde{t}$. Correspondingly, new alias processes of the form $\mathsf{newA}_{\mathbb{O}}\langle\, s, s_{\mathrm{a}}\,\rangle$ connect invocations from the outside to a target process listening at s_{a}. Inside $\mathsf{newO}_{\mathbb{O}}\langle\, s, \widetilde{t}\,\rangle$ and $\mathsf{newA}_{\mathbb{O}}\langle\, s, s_{\mathrm{a}}\,\rangle$, several private names are needed: *mutexes* $\widetilde{m} := m_{\mathrm{e}}, m_{\mathrm{i}}$ are used for serialisation; the (*internal*) key k_{i} is used to detect self-infliction; the (*external*) key k_{e} is used to implement serialisation in a concurrent environment (see later on).

Our semantics associates an object manager OM to each object, and an alias manager AM to each alias. Before entering into the details of the translation in Table 7, we provide, in Figure 1, a more abstract overview of the lifetime of an object manager, possibly turning it into an alias manager, by emphasising the relevant states passed. Both object and alias managers listen on their reference channel s for requests. Since objects (resp. aliases) in Øjeblik are serialised, only one request shall be *active* in an object (resp. alias), at any moment. Serialisation is implemented by two mutexes m_{e} and m_{i}: the external one must be grabbed in order to get access to the manager; the internal one precisely alternates with the external one and is used to intermediately save some context information. External requests must grab the external-mutex m_{e} before being served, which in turn brings the object manager from state OM^{f} to state OM^{a}. Then, if the request is protection-critical it is *discarded* (state OM^{n}), otherwise the manager *commits* to it and *serves* it (in state OM^{s}) until explicit *termination* (state OM^{i}). In both cases, the object manager becomes free again by releasing the external-mutex m_{e} (state OM^{f}). Notice that self-inflicted requests can only be served in state OM^{s}. Furthermore, when serving *self-inflicted aliasing* requests, the object becomes an alias and the object manager is replaced by an appropriate alias manager (in state AM^{c}). AM^{c} is a transient state where the alias manager accomplishes all pending self-inflicted requests; note that all of the latter were generated by the external request that is also responsible for creating the alias. When this external request is completed, the manager terminates and goes to state AM^{i}. Afterwards, the mutex m_{e} is released and the alias manager becomes free (state AM^{f}). Only now, external requests addressed to the alias manager are treated again. They must grab the external-mutex m_{e} before being forwarded, bringing the alias manager from state AM^{f} to state AM^{a}. After grabbing m_{e}, external requests will be accepted and forwarded to the alias target (state AM^{s}). The alias manager becomes free again by releasing the external-mutex m_{e} (state AM^{f}). Finally, since alias managers always forward external requests, no self-inflicted requests may be generated anymore. This explains why no self-inflicted requests are taken into account in state AM^{s}.

The following three paragraphs explain in detail how object and alias managers serve requests, referring now directly to the translation semantics Table 7.

Pre-processing $[k_{\mathrm{i}}{\neq}k{\neq}k_{\mathrm{e}}]$

Here, we explain how the serialisation of external requests is implemented. Upon creation of a new object newO (or new alias newA), the fresh mutex channel m_{e} is initialised. According to serialisation, the intended continuation behaviour of an incoming external requests is blocked on m_{e}, once it enters a manager. The manager itself is immediately restarted and remains receptive. Arbitrarily many requests can be blocked this way and compete for the mutex m_{e} once it becomes available. A successfully unblocked request is resent to the same manager, but now carrying the key k_{e}, which allows the manager to detect that the request has grabbed the mutex. We call *pre-processing* the procedure of intermediate blocking of requests. Alongside with the successful

$$\llbracket \mathbb{O} \rrbracket_p^k \stackrel{\text{def}}{=} (\boldsymbol{\nu} s\widetilde{t})\Big(\overline{p}\langle s,k\rangle \mid \mathsf{newO}_{\mathbb{O}}\langle\, s,\widetilde{t}\,\rangle \mid \prod_{j\in J} \mathbf{!}\, t_j(s_j,\widetilde{x}_j,r,k').\llbracket\, b_j \,\rrbracket_r^{k'} \Big)$$

$$\mathsf{newO}_{\mathbb{O}}\langle\, s,\widetilde{t}\,\rangle \stackrel{\text{def}}{=} (\boldsymbol{\nu} m_{\mathrm{e}} m_{\mathrm{i}} k_{\mathrm{e}} k_{\mathrm{i}})\Big(\overline{m_{\mathrm{e}}} \mid \mathsf{OM}_{\mathbb{O}}\langle\, s,m_{\mathrm{e}},m_{\mathrm{i}},k_{\mathrm{e}},k_{\mathrm{i}},\widetilde{t}\,\rangle \Big)$$

$$\mathsf{newA}_{\mathbb{O}}\langle\, s,s_{\mathrm{a}}\,\rangle \stackrel{\text{def}}{=} (\boldsymbol{\nu} m_{\mathrm{e}} m_{\mathrm{i}} k_{\mathrm{e}} k_{\mathrm{i}})\Big(\overline{m_{\mathrm{e}}} \mid \mathsf{AM}_{\mathbb{O}}\langle\, s,m_{\mathrm{e}},m_{\mathrm{i}},k_{\mathrm{e}},k_{\mathrm{i}},s_{\mathrm{a}}\,\rangle \Big)$$

$$\begin{array}{l}
\mathsf{OM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k_{\mathrm{i}},\widetilde{t}\,\rangle \stackrel{\text{def}}{=} s(l,k).(\boldsymbol{\nu} k^*)\Big(\\
\quad \mathsf{if}\ [k{=}k_{\mathrm{i}}]\ \mathsf{then} \\
\qquad \mathsf{case}\ l\ \mathsf{of}\ \mathsf{cln_}(r) : \mathsf{OM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k^*,\widetilde{t}\,\rangle \mid (\boldsymbol{\nu} s^*)\,(\, \overline{r}\langle s^*,k^*\rangle \mid \mathsf{newO}_{\mathbb{O}}\langle\, s^*,\widetilde{t}\,\rangle\,)\ \ ; \\
\qquad\qquad \mathsf{ali_}(s_{\mathrm{a}},r) : \mathsf{AM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k^*,s_{\mathrm{a}}\,\rangle \mid \overline{r}\langle s_{\mathrm{a}},k^*\rangle\ \ ; \\
\qquad\qquad \mathsf{upd}_j\mathsf{_}(t',r) : \mathsf{OM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k^*,t_1\,..\,t_{j-1},t',t_{j+1}\,..\,t_n\,\rangle \mid \overline{r}\langle s,k^*\rangle\ \ ; \\
\qquad\qquad \mathsf{inv}_j\mathsf{_}(\widetilde{x},r) : \mathsf{OM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k^*,\widetilde{t}\,\rangle \mid \overline{t_j}\langle s,\widetilde{x},r,k^*\rangle\ \ ; \\
\qquad\qquad \mathsf{sur_}(r) : \mathsf{OM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k^*,\widetilde{t}\,\rangle \mid \llbracket\, s.\mathsf{alias}\langle s.\mathsf{clone}\rangle \,\rrbracket_r^{k^*}\ \ ; \\
\qquad\qquad \mathsf{png_}(r) : \mathsf{OM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k^*,\widetilde{t}\,\rangle \mid \llbracket\, s \,\rrbracket_r^{k^*} \\
\quad \mathsf{elif}\ [k{=}k_{\mathrm{e}}]\ \mathsf{then} \\
\qquad \mathsf{OM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k^*,\widetilde{t}\,\rangle \mid \mathsf{case}\ l\ \mathsf{of}\ \mathsf{cln_}(r) : m_{\mathrm{i}}(k).\overline{m_{\mathrm{e}}}\ \ ; \\
\qquad\qquad\qquad \mathsf{ali_}(s_{\mathrm{a}},r) : m_{\mathrm{i}}(k).\overline{m_{\mathrm{e}}}\ \ ; \\
\qquad\qquad\qquad \mathsf{upd}_j\mathsf{_}(t',r) : m_{\mathrm{i}}(k).\overline{m_{\mathrm{e}}}\ \ ; \\
\qquad\qquad\qquad \mathsf{inv}_j\mathsf{_}(\widetilde{x},r) : \mathsf{CM}[\, \overline{t_j}\langle s,\widetilde{x},r^*,k^*\rangle \,]\ \ ; \\
\qquad\qquad\qquad \mathsf{sur_}(r) : \mathsf{CM}[\, \llbracket\, s.\mathsf{alias}\langle s.\mathsf{clone}\rangle \,\rrbracket_{r^*}^{k^*} \,]\ \ ; \\
\qquad\qquad\qquad \mathsf{png_}(r) : \mathsf{CM}[\, \llbracket\, s \,\rrbracket_{r^*}^{k^*} \,] \\
\quad \mathsf{else}\ \mathsf{OM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k_{\mathrm{i}},\widetilde{t}\,\rangle \mid m_{\mathrm{e}}.(\ \overline{s}\langle l,k_{\mathrm{e}}\rangle \ \mid\ \overline{m_{\mathrm{i}}}k\ )\Big)
\end{array}$$

$$\mathsf{CM}[\cdot] \stackrel{\text{def}}{=} (\boldsymbol{\nu} r^*)\big(\, [\cdot] \ \mid\ r^*(y,k').m_{\mathrm{i}}(k'').(\,\overline{r}\langle y,k''\rangle \ \mid\ \overline{m_{\mathrm{e}}}\,)\,\big)$$

$$\begin{array}{l}
\mathsf{AM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k_{\mathrm{i}},s_{\mathrm{a}}\,\rangle \stackrel{\text{def}}{=} s(l,k).(\boldsymbol{\nu} k^*)\Big(\\
\quad \mathsf{if}\ [k{=}k_{\mathrm{i}}]\ \mathsf{then} \\
\qquad \mathsf{case}\ l\ \mathsf{of}\ \mathsf{cln_}(r) : \mathsf{AM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k^*,s_{\mathrm{a}}\,\rangle \mid (\boldsymbol{\nu} s^*)\,(\, \overline{r}\langle s^*,k^*\rangle \mid \mathsf{newA}_{\mathbb{O}}\langle\, s^*,s_{\mathrm{a}}\,\rangle\,)\ \ ; \\
\qquad\qquad \mathsf{ali_}(s'_{\mathrm{a}},r) : \mathsf{AM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k^*,s'_{\mathrm{a}}\,\rangle \mid \overline{r}\langle s'_{\mathrm{a}},k^*\rangle\ \ ; \\
\qquad\qquad \mathsf{upd}_j\mathsf{_}(t',r) : \mathsf{AM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k^*,s_{\mathrm{a}}\,\rangle \mid \overline{s_{\mathrm{a}}}\langle l,k\rangle\ \ ; \\
\qquad\qquad \mathsf{inv}_j\mathsf{_}(\widetilde{x},r) : \mathsf{AM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k^*,s_{\mathrm{a}}\,\rangle \mid \overline{s_{\mathrm{a}}}\langle l,k\rangle\ \ ; \\
\qquad\qquad \mathsf{sur_}(r) : \mathsf{AM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k^*,s_{\mathrm{a}}\,\rangle \mid \overline{s_{\mathrm{a}}}\langle l,k\rangle\ \ ; \\
\qquad\qquad \mathsf{png_}(r) : \mathsf{AM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k^*,s_{\mathrm{a}}\,\rangle \mid \overline{s_{\mathrm{a}}}\langle l,k\rangle \\
\quad \mathsf{elif}\ [k{=}k_{\mathrm{e}}]\ \mathsf{then}\ \mathsf{AM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k^*,s_{\mathrm{a}}\,\rangle \ \mid\ m_{\mathrm{i}}(k).(\ \overline{s_{\mathrm{a}}}\langle l,k\rangle \ \mid\ \overline{m_{\mathrm{e}}}\ ) \\
\quad \mathsf{else}\qquad\qquad \mathsf{AM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k_{\mathrm{i}},s_{\mathrm{a}}\,\rangle \ \mid\ m_{\mathrm{e}}.(\ \overline{s}\langle l,k_{\mathrm{e}}\rangle \ \mid\ \overline{m_{\mathrm{i}}}k\ )\Big)
\end{array}$$

Table 7: Translational Semantics of Øjeblik — Objects

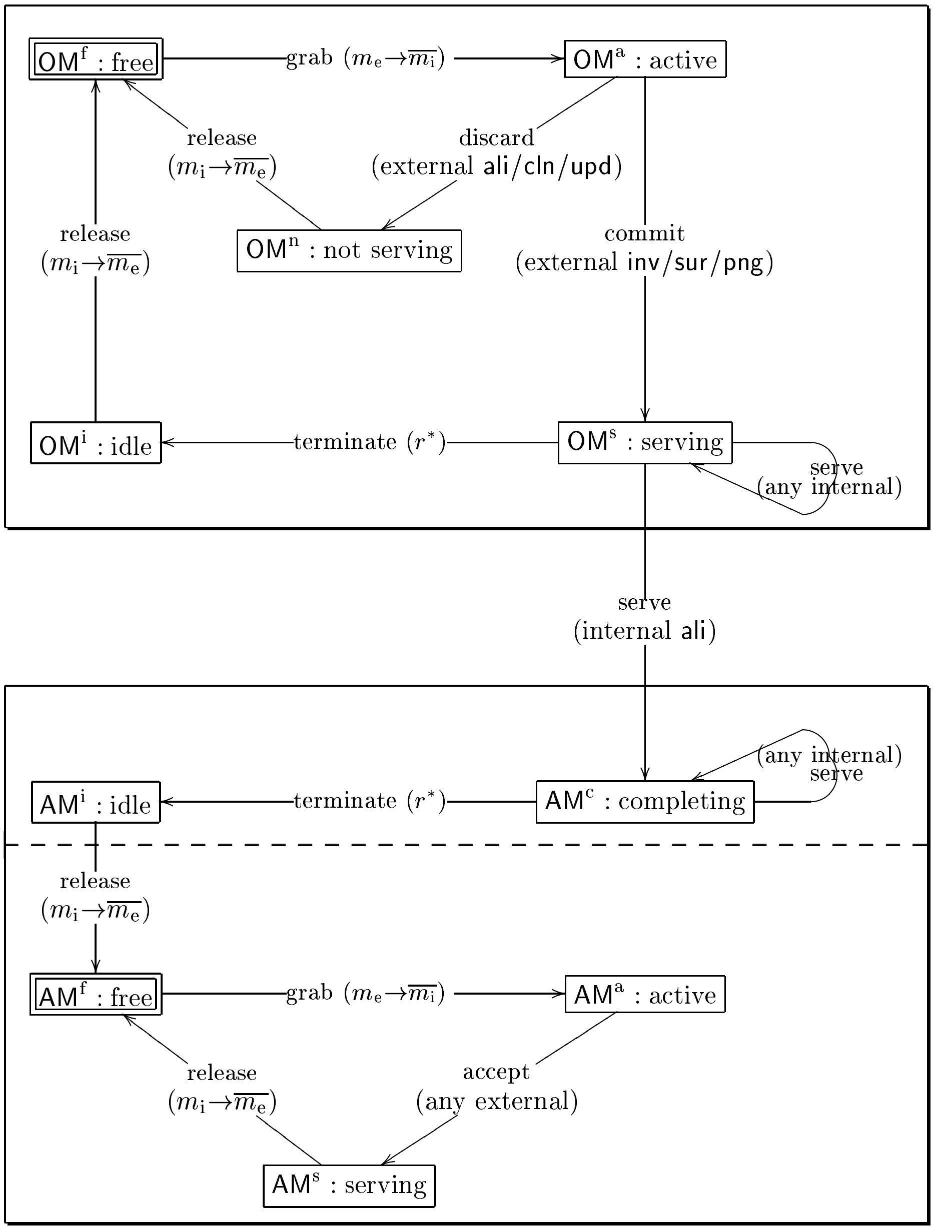


Figure 1: Object and Alias Manager Serving Requests

request, its former current self k is stored on the (internal) mutex m_i for recovery after termination. This recovery is actually necessary since the original current self k is possibly required for use later on by the sender of the request. Note that pre-processing also properly takes care of the fact that competing requests may change the state of an object, and even turn it into an alias by passing from OM^s to AM^c, so pre-processed requests should not be bound too early to some object manager behaviour. By only resending a request once it has grabbed the mutex, it will be handled by the current manager, not by the manager in the state of the moment when the request originally entered the object. Notice that pre-processing in alias managers is not superfluous, because there may be pending requests that have been pre-processed when s was connected to an OM. Finally, pre-processing does not preclude the evolution of the system, that is, external requests can be pre-processed at any moment (in any state) by both alias and object managers without affecting the state of the manager, so these transitions are completely ignored in Figure 1.

Serving external requests $[k{=}k_{\mathrm{e}}]$

Serialization and protection are required. Here, we explain how external requests, which have already been pre-processed and have already grabbed the external mutex m_{e}, are served by both object and alias managers.

Object Managers (OM). When serving an external request, the manager OM is immediately restarted with the same state except for the fresh internal key k^*. The key k^* must subsequently be used as the current self when performing the current request. Later on, we will better explain the use of k^*.

Cloning, aliasing, and update, are critical operations. Once a respective pre-processed request is consumed, the manager evolves from state OM^{a} into state OM^{n}: the request and its former current self k, stored on channel m_{i}, are simply discarded by consuming $\overline{m_{\mathrm{i}}}k$ and releasing $\overline{m_{\mathrm{e}}}$.

Invocation, surrogation, and ping are non-critical operations. Once a respective pre-processed request is consumed, the manager evolves from state OM^{a} into state OM^{s} implying that no other external request shall be served (apart from pre-processing) until the current one has terminated. In order to be notified of that event, we employ a *call manager* protocol, represented by the context $\mathsf{CM}[\cdot]$: instead of delegating to some other process the responsibility of returning a result on r, a fresh return channel r^* is created to be used within $[\cdot]$ in place of r, such that the result will first appear on r^*. Until this event, other external requests remain blocked, while internal request may well be served. After this event, the manager evolves from state OM^{s} into state OM^{i}, where the former current self can be grabbed from m_{i}, the result y be forwarded to the intended result channel r (along with the former current self), and the mutex m_{e} be released. In the case of invocation (case inv_j), the manager activates the method body bound to l_j along trigger name t_j. Note that (externally) triggered method bodies $[\![\, b_j \,]\!]$, and also surrogation and ping bodies $[\![\, s.\mathsf{alias}\langle s.\mathsf{clone}\rangle \,]\!]$ and $[\![\, s \,]\!]$, are all run in the context of the nonce k^* (see below), which is now the new internal key of the OM, so their further calls to s will be self-inflicted. This is essential for surrogation, since cloning and aliasing are only allowed internally.

Alias Managers (AM). When serving external requests, alias managers, like object managers, are immediately restarted with the same state except for the fresh internal key k^*. External requests that arrive at an active alias manager (in state AM^{a}) will be simply forwarded (in state AM^{s}) without modification of the current-self k (obtained by consuming $\overline{m_{\mathrm{i}}}k$) to the aliasing target s_{a}. Finally, when releasing $\overline{m_{\mathrm{e}}}$, the manager will evolve to state AM^{f}.

Serving self-inflicted requests $[k{=}k_{\mathrm{i}}]$

No serialization or protection is required. Here, we explain how self-inflicted requests are served by both object and alias managers.

Object Managers (OM). For each field, the manager may activate appropriate instances of method bodies (case inv_j: the method body bound to l_j along trigger name t_j) and administer updates (case upd_j: install a new trigger name t'). Cloning (case cln) restarts the current object manager in parallel with a new object, which uses the same method bodies $\widetilde{t}$, but is accessible through a fresh reference s^*. In all cases except aliasing, an object manager OM is restarted with a fresh internal key k^*. Aliasing (case ali) starts an appropriate alias manager AM instead of restarting the previous object manager OM. Surrogation and ping (cases sur and png) are modelled according to their uniform method definitions.

Alias Managers (AM). To perform self-inflicted requests the alias manager may only be in the transient state AM^{c}. Cloning and alias requests are allowed and treated as in the respective clauses of object managers, but restarting AM instead of OM. Invocation, surrogation, update, and ping requests are forwarded to the aliasing target s_{a} without modification of the current-self k.

$$
\begin{aligned}
\mathbf{R}(X) &\stackrel{\text{def}}{=} \mathbf{C}(X,\mathbf{K})\\
\mathbf{M}(B_1 \mathinner{..} B_n \to \widehat{B}) &\stackrel{\text{def}}{=} [\![B_1]\!] \mathinner{..} [\![B_n]\!], \mathbf{R}([\![\widehat{B}]\!])
\end{aligned}
$$

$$
[\![[\mathrm{l}_j{:}\widetilde{B}_j \to \widehat{B}_j]_{j\in J}]\!] \stackrel{\text{def}}{=}
\mu X.\, \mathbf{C}(\begin{bmatrix}
\mathsf{cln} : & \mathbf{R}(X)\\
\mathsf{ali} : & \langle X, \mathbf{R}(X) \rangle\\
\mathsf{upd}_j : & \langle \mathbf{C}(X, \mathbf{M}(\widetilde{B}_j \to \widehat{B}_j), \mathbf{K}), \mathbf{R}(X) \rangle\\
\mathsf{inv}_j : & \langle \mathbf{M}(\widetilde{B}_j \to \widehat{B}_j) \rangle\\
\mathsf{sur} : & \mathbf{R}(X)\\
\mathsf{png} : & \mathbf{R}(X)
\end{bmatrix}_{j\in J}, \mathbf{K})
$$

$$
\begin{aligned}
[\![\mathrm{Thr}(A)]\!] &\stackrel{\text{def}}{=} \mathbf{C}(\mathbf{R}([\![A]\!]), \mathbf{K})\\
[\![\Gamma, x{:}A]\!] &\stackrel{\text{def}}{=} [\![\Gamma]\!], x{:}[\![A]\!]
\end{aligned}
$$

Table 8: Translation of Øjeblik-types

Nonces ($\boldsymbol{\nu} k^*$)

We use *nonces* k^* to implement self-serialisation between self-inflicted requests. When serving self-inflicted and external requests, managers OM and AM are always restarted by replacing the current self with a fresh key k^*. According to our semantics, program contexts will never give rise to several competing (external or self-inflicted) requests, but, when reasoning within arbitrary $\mathrm{L}\pi^+$ contexts, as we do in Section 8.1, their existence must be taken into account. Therefore, we add another layer of protection to increase the robustness of serialisation: each time a (self-inflicted or external) request enters a manager, a fresh key k^* is created to be used in the restarted manager; this key must subsequently be used as the current self for all activities enabled by the current request. Thus, the consumption of one of the competing pending requests renders the other competitors external. Notice that *pre-processing must not reinitialise the key* k_{i} of the restarted manager: a currently self-inflicted operation interleaved by pre-processing might be hindered to proceed, because it could unintendedly become external.

6 Properties of the translational semantics

This section is devoted to show two fundamental properties of our translational semantics: (i) the translation preserves well-typedness; (ii) objects (and alias) managers are unique.

6.1 The $\mathrm{L}\pi^+$-translation preserves well-typedness

A translation of the type system of Øjeblik into the type system of the π-calculus has several merits: (i) it strengthens the soundness of our semantics of terms, as in Theorem 6.1; (ii) Øjeblik's type system itself is provided with some more formal underpinning, as demonstrated in Proposition 6.2; (iii) we may employ typeful reasoning about terms, of which we give examples in Proposition 6.3. The translation of types, shown in Table 8, is similar to the ones for the Functional and Imperative Object Calculus found in [San98, KS98]. We use some handy abbreviations to denote (i) the type $\mathbf{R}(X)$ of result channels, which can be used to retrieve results of type X, together with the current key; (ii) the type $\mathbf{M}(B_1 \mathinner{..} B_n \to \widehat{B})$ of methods, which is self-explanatory. The most critical part of the translation is the proper representation in the case of update, but even there, the chosen abbreviations allow us to directly relate the types with the corresponding terms in Tables 6 and 7. The translation of $\mathrm{Thr}(A)$ denotes the type of name t in the semantics of fork and join in Table 6. Note that, because we intended to stay within the constraints of $\mathrm{L}\pi$, we could not use t directly to retrieve the value of a fork'ed term a, but we used it to send the result channel of the join'ing term, together with its current key—this is precisely represented in the translation of $\mathrm{Thr}(A)$.

According to the translation of types, we can add type declarations in a straightforward way to all bindings in the translation of terms, as mentioned, although omitted, in Section 5.

Types witness the clean representation of Øjeblik terms as π-calculus terms.

Theorem 6.1 (Type Soundness) *Let $a \in \mathcal{L}$, let Γ be a type-environment, and let A be a type. Then* $\Gamma \vdash a{:}A$ *if and only if* $[\![\,\Gamma\,]\!]\,, p{:}\mathbf{R}([\![\,A\,]\!])\,, k{:}\mathbf{K} \vdash [\![\,a\,]\!]^k_p$ *for names p and k.*

Proof. The implication from left to right is proved using induction in the depth of the derivation of $\Gamma \vdash a{:}A$ with a case analysis of the last rule used. The implication from right to left is proved by induction in the structure of a. Details can be found in Appendix A.2. □

In addition to the initial correspondence of types in Øjeblik and their π-calculus counterparts, the preservation of types under reduction in the π-calculus provides us for free with preservation of Øjeblik types, thus witnessing the subject reduction theorem based on the operational semantics in [NHKM00].

Proposition 6.2 (Subject Reduction) *Let* $\Gamma \vdash a{:}A$.
If $[\![\,a\,]\!]^k_p \Rightarrow Q$, *then* $[\![\,\Gamma\,]\!]\,, p{:}\mathbf{R}([\![\,A\,]\!])\,, k{:}\mathbf{K} \vdash Q$.

The type system provides some properties of the translation almost for free. Let us fix some terminology. A term P occurs *weakly unguarded* in Q, if there is $Q \Rightarrow Q' \equiv E'[P]$, where $E'[\cdot]$ is a static $\mathrm{L}\pi^+$-context. By means of the type translation, we can show that whenever, at top-level, a request may be directed to some potential object or alias manager, there will always be some manger occurring weakly unguarded and thus being eventually able to serve the request.

Proposition 6.3 *Let* $\Gamma \vdash a{:}A$ *and* $E[\cdot]$ *be a static* $\mathrm{L}\pi^+$*-context.*

1. *If* $[\![\,a\,]\!]^k_p \Rightarrow Q \equiv E[\overline{s}\langle\,.\,.\,\rangle]$,
   then either $\mathsf{AM}_{\mathbb{O}}\langle\, s, \ldots \rangle$ *or* $\mathsf{OM}_{\mathbb{O}}\langle\, s, \ldots \rangle$ *occurs weakly unguarded in Q.*
2. *If* $[\![\,a\,]\!]^k_p \Rightarrow Q \equiv E[\mathsf{AM}_{\mathbb{O}}\langle\, s, \ldots, s_{\mathrm{a}} \rangle]$,
   then either $\mathsf{AM}_{\mathbb{O}}\langle\, s_{\mathrm{a}}, \ldots \rangle$ *or* $\mathsf{OM}_{\mathbb{O}}\langle\, s_{\mathrm{a}}, \ldots \rangle$ *occurs weakly unguarded in Q.*

Proof. [Sketch] Since $\Gamma \vdash a{:}A$, also $[\![\,a\,]\!]^k_p$ is well-typed. By inspection of the encoding, whenever a self-channel is created, also the respective manager is created. The well-typedness of $[\![\,a\,]\!]^k_p$ means that managers cannot disappear: when they receive a message, they can only be guarded by matching, followed by case; by well-typedness, the case can be resolved, resulting in a new manager at the same name. Finally, the creation of requests is always guarded by an input of a self-channel and a key, so the creation of requests always follows the creation of a manager, but never proceeds it. When an object manager changes into an alias manager, it installs as target the self-channel of another manager, which by construction and well-typedness of the translation cannot disappear. □

As a consequence, by transitivity and the finiteness of terms, this proposition tells us that alias chains are either cyclic or end up with an object manager. In other words, when a request is sent to an object it either eventually arrives at an object manager, or it cycles in a loop between alias managers.

6.2 Properties of object managers

A crucial property in object-oriented languages is the uniqueness of objects. The $\mathrm{L}\pi$ constraint on the output capability guarantees this property.

Lemma 6.4 (Uniqueness of objects) *Let a be an Øjeblik term. If* $[\![\,a\,]\!]^k_p \Rightarrow Z$ *with*

$$\text{either } Z \equiv (\boldsymbol{\nu}\widetilde{z})\,(\,M \mid \mathsf{OM}_{\mathbb{O}}\langle\, s, \ldots \rangle\,) \text{ or } Z \equiv (\boldsymbol{\nu}\widetilde{z})\,(\,M \mid \mathsf{AM}_{\mathbb{O}}\langle\, s, \ldots \rangle\,)$$

then $s \in \widetilde{z}$ and s does not appear free in input position within M.

PROOF. By inspection of the encoding. If a manager is present, it must have been created at some point as described in the encoding, because initially, there is none. Upon creation, its name s is bound. Since we only consider reductions, the name remains bound. Finally, the encoding shows that managers are only restarted if the former incarnation disappears. Since there are never two copies restarted, and only the output capability of channels may be transmitted, the uniqueness of the receptor s is preserved. □

We now analyse, referring directly to Figure 1, how the shape of the context around a particular object manager evolves during computation (c.f. Lemma 6.5). Later on, we will need a special case of this result (Lemma 6.6) in the proof of Theorem 8.7.

Observation 1: Pre-processing does not change the state of object managers. At any time, an object/alias manager is ready to receive a request $\overline{s}\langle l, k\rangle$ with $k_e \neq k \neq k_i$. The manager is restarted afterwards, but there will be a process $m_e.(\,\overline{s}\langle l, k_e\rangle \mid \overline{m_i}k\,)$ that replaces the consumed request. Let us assume requests $\overline{s}v_j$, with $v_j := \langle\, l_j, k_j\,\rangle$ for $j \in 1..h$, (and $\widetilde{v}{:=}v_1..v_h$) are pre-processed by the object manager $\mathsf{OM}_{\mathbb{O}}\langle\, s, m_e, m_i, k_e, k_i, \widetilde{t}\,\rangle$, so $k_e \neq k_j \neq k_i$ for all $j \in 1..h$. Then:

$$\mathsf{PP}_{\mathbb{O}}\langle\, s, m_e, m_i, k_e, \widetilde{v}\,\rangle \stackrel{\text{def}}{=} \prod_{j \in 1..h} m_e.(\, \overline{s}\langle l_j, k_e\rangle \mid \overline{m_i}k_j\,)$$

Observation 2: While an object manager evolves, its internal key k_i may be extruded to its object clients, whereas names m_e, m_i, k_e may not. Assume that an inv_j-request (along s) appears at $\mathsf{OM}_{\mathbb{O}}\langle\, s, m_e, m_i, k_e, k_i, \widetilde{t}\,\rangle$, is pre-processed, gets the mutex m_e and re-enters along s with key k_e. At that point, according to the semantics, a fresh internal key k^* is created and extruded to the corresponding method body. The names $\widetilde{n} := m_e, m_i, k_e$ are never extruded; they constitute the proper boundary of a manager during computation. Observation 2 provides the formal basis to understand the evolution of object and alias managers as described in Figure 1. For simplicity, we restrict our analysis to object managers, but a similar argument applies to alias managers.

Lemma 6.5 (Object manager evolution) *Let a be an Øjeblik term. If $[\![\, a \,]\!]_p^k \Rightarrow Z$ and $Z = E[\mathsf{OM}_{\mathbb{O}}\langle\, s, \dots\rangle]$, with $E[\cdot]$ static, then—without α-converting the name s—*

$$Z \equiv \widehat{E}[\ (\boldsymbol{\nu}\widetilde{n})\,(\ M_Z \mid \mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{n}, k_i, \widetilde{t}\,\rangle \mid \mathsf{PP}_{\mathbb{O}}\langle\, s, \widetilde{n}, \widetilde{v}\,\rangle\ )\ ]$$

where $\widehat{E}[\cdot]$ is a static $\mathrm{L}\pi^+$-context, $\widetilde{n} := m_e, m_i, k_e$, and M_Z is either of

Z	M_Z
OM^{f}	$\overline{m_e}$
OM^{a}	$\overline{m_i}k \mid \overline{s}\langle l, k_e\rangle$
OM^{n}	$\overline{m_i}k \mid m_i(k).\overline{m_e}$
OM^{s}	$\overline{m_i}k \mid r^*(y, k').m_i(k'').(\overline{r}\langle y, k''\rangle \mid \overline{m_e})$
OM^{i}	$\overline{m_i}k \mid m_i(k'').(\overline{r}\langle y, k''\rangle \mid \overline{m_e})$.

with Z denoting the states of OM *as in Figure 1.*

PROOF. By induction on the length of $[\![\, a \,]\!]_p^k \Rightarrow Z$ for some fixed s, where we assume that the predecessor state Z' of Z is in one of the five described "states". Details can be found in Appendix A.3. □

In the following two observations, we outline two special cases of Lemma 6.5: *free* object managers in state OM^{f} and *committing* object managers ready to evolve from state OM^{a} to state OM^{s}.

Observation 3: An object manager is free*, if its external-mutex m_e is available.* In our semantics, a manager is willing to grant access to external requests, if its external-mutex m_e occurs unguarded in the term that describes the current state, so the general shape of a *free object* (and analogously *alias*) *manager* is:

$$\begin{aligned}
\mathsf{freeO}_{\mathbb{O}}\langle\, s, k_i, \widetilde{t}, \widetilde{v}\,\rangle &\stackrel{\text{def}}{=} (\boldsymbol{\nu}\widetilde{n})(\ \overline{m_e} \mid \mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{n}, k_i, \widetilde{t}\,\rangle \mid \mathsf{PP}_{\mathbb{O}}\langle\, s, \widetilde{n}, \widetilde{v}\,\rangle\ ) \\
\mathsf{freeA}_{\mathbb{O}}\langle\, s, k_i, s_a, \widetilde{v}\,\rangle &\stackrel{\text{def}}{=} (\boldsymbol{\nu}\widetilde{n})(\ \overline{m_e} \mid \mathsf{AM}_{\mathbb{O}}\langle\, s, \widetilde{n}, k_i, s_a\,\rangle \mid \mathsf{PP}_{\mathbb{O}}\langle\, s, \widetilde{n}, \widetilde{v}\,\rangle\ )
\end{aligned}$$

where the keys mentioned in $\widetilde{v}$ of $\mathsf{PP}_{\mathbb{O}}\langle \dots \rangle$ neither match k_{e} nor k_{i}. Notice that $\mathsf{newO}_{\mathbb{O}}\langle\, s, \widetilde{t}\,\rangle \equiv (\boldsymbol{\nu} k_{\mathrm{i}})\, \mathsf{freeO}_{\mathbb{O}}\langle\, s, k_{\mathrm{i}}, \widetilde{t}, \emptyset\,\rangle$, and analogously for $\mathsf{newA}_{\mathbb{O}}\langle \dots \rangle$.

Observation 4: An object manager is ready to commit, *if it may consume a pre-processed request which has already grabbed* m_{e}. The following lemma derives from the ability to commit to a valid external request—visible as the availability of a valid pre-processed request, i.e., a request carrying k_{e}—the shape of the object manager before and after commitment, including all of its current pre-processed requests.

Lemma 6.6 (Committing object manager) *Let a be an Øjeblik term. If* $[\![\, a \,]\!]_p^k \Rightarrow Z$ *and* $Z \equiv E[\, \overline{s}\langle l, k_{\mathrm{e}}\rangle \mid \mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{n}, k_{\mathrm{i}}, \widetilde{t}\,\rangle \,]$ *with* $E[\cdot]$ *static,* $\widetilde{n} = m_{\mathrm{e}}, m_{\mathrm{i}}, k_{\mathrm{e}}$, *and* $l \in \{\mathsf{inv}_{j}\_\langle \widetilde{x}, r\rangle, \mathsf{png}\_r, \mathsf{sur}\_r\}$, *then* $Z \xrightarrow{\tau} Z'$ *where*

$$\begin{array}{rcl} Z & \equiv & \widehat{E}[\, (\boldsymbol{\nu}\widetilde{n})\big(\overline{m_{\mathrm{i}}}k \mid \mathsf{PP}_{\mathbb{O}}\langle\, s, \widetilde{n}, \widetilde{v}\,\rangle \mid \mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{n}, k_{\mathrm{i}}, \widetilde{t}\,\rangle \mid \overline{s}\langle l, k_{\mathrm{e}}\rangle\big)\,] \\ Z' & \equiv & \widehat{E}[\, (\boldsymbol{\nu}\widetilde{n})\big(\overline{m_{\mathrm{i}}}k \mid \mathsf{PP}_{\mathbb{O}}\langle\, s, \widetilde{n}, \widetilde{v}\,\rangle \mid (\boldsymbol{\nu} k^*)(\, \mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{n}, k^*, \widetilde{t}\,\rangle \mid \mathsf{CM}[\, X_l\langle\, s\,\rangle_{r^*}^{k^*}\,]\,)\big)\,] \end{array}$$

for some static context $\widehat{E}[\cdot]$, *some key* k, *some set* $\widetilde{v}$ *of pre-processed requests, and* $X_l\langle\, s\,\rangle$ *denoting the respective continuation behaviour of Table 7.*

Proof. According to Lemma 6.5, the property holds in state OM^{a} which is the only state that matches the premise. □

As special cases, for $l \in \{\mathsf{png}\_r, \mathsf{sur}\_r\}$, of *committed* object managers, we define

$$\begin{array}{rcl} F[\cdot] & \stackrel{\mathrm{def}}{=} & (\boldsymbol{\nu}\widetilde{n}k^*)\big(\, \overline{m_{\mathrm{i}}}k \big|\, \mathsf{PP}_{\mathbb{O}}\langle\, s, \widetilde{n}, \widetilde{v}\,\rangle \,\big|\, \mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{n}, k^*, \widetilde{t}\,\rangle \,\big|\, [\cdot]\,\big) \\ \mathsf{pingO}_{\mathbb{O}}\langle\, s, r, k, \widetilde{t}, \widetilde{v}\,\rangle & \stackrel{\mathrm{def}}{=} & F[\mathsf{CM}[\, [\![\, s \,]\!]_{r^*}^{k^*}\,]] \\ \mathsf{surO}_{\mathbb{O}}\langle\, s, r, k, \widetilde{t}, \widetilde{v}\,\rangle & \stackrel{\mathrm{def}}{=} & F[\mathsf{CM}[\, [\![\, s.\mathsf{alias}\langle s.\mathsf{clone}\rangle \,]\!]_{r^*}^{k^*}.\,]] \end{array}$$

As we will see in Section 8.1, $\mathsf{pingO}_{\mathbb{O}}\langle\, s, r, k, \widetilde{t}, \widetilde{v}\,\rangle$ and $\mathsf{surO}_{\mathbb{O}}\langle\, s, r, k, \widetilde{t}, \widetilde{v}\,\rangle$ model the object manager before and after surrogation, respectively.

7 Towards a formalization of safe surrogation

In [NHKM00], we motivated an equation on Øjeblik terms to model the safety of object surrogation. In Subsection 7.1, we replay the argument leading to that equation and adapt it to the translational semantics of Øjeblik. In [NHKM00], we also observed that the equation intrinsically can only be true in a restricted sense. The techniques of Subsection 7.2 will allow us precisely formalize this restriction.

7.1 Safety as an Equation

We recall that in order to be safe, object surrogation should be transparent to object clients. In other words *an object should behave the same before and after surrogation*, in all possible contexts. The following equation is a first attempt to model this property:

$$a \doteq a.\mathsf{surrogate} \tag{1}$$

The simplest case of Equation 1 is when a is an object $\mathbb{O}$. In this case the surrogation is surely safe, because (i) the process of surrogation is carried out correctly since, due to serialisation, only the surrogation thread can interact with the object $\mathbb{O}$, i.e., there cannot be any interference with another thread or activity, and (ii) every interaction with $\mathbb{O}$ is mimicked identically by $\mathbb{O}.\mathsf{surrogate}$, which suffices since after surrogation nobody has access to the previous $\mathbb{O}$.

In the general case, however, neither of the two above arguments holds. The reason is because of possible copying of a reference to the former object such that, after surrogation, requests can still be directed to that reference. Observing that $a \doteq \mathsf{let}\, x = a\, \mathsf{in}\, x$ (in all contexts, the let just

$C[\cdot]$ ::= $[\cdot]$		$\mid [\, \mathrm{l}_k {=} \varsigma(s,\tilde{x})C[\cdot]\ ,\ \mathrm{l}_{j\neq k} {=} m_{j\neq k}\ ]_{j\in J}$
	$\mid C[\cdot].\mathrm{l}\langle\, \tilde{a}\,\rangle$	$\mid a.\mathrm{l}\langle\, \tilde{a}, C[\cdot], \tilde{a}\,\rangle$
	$\mid C[\cdot].\mathrm{l}{\Leftarrow}m$	$\mid a.\mathrm{l}{\Leftarrow}\varsigma(s,\tilde{x})C[\cdot]$
	$\mid C[\cdot].\mathsf{alias}\langle b\rangle$	$\mid a.\mathsf{alias}\langle C[\cdot]\rangle$
	$\mid C[\cdot].\mathsf{clone}$	
	$\mid C[\cdot].\mathsf{surrogate}$	$\mid C[\cdot].\mathsf{ping}$
	$\mid \mathsf{let}\ x = C[\cdot]\ \mathsf{in}\ b$	$\mid \mathsf{let}\ x = a\ \mathsf{in}\ C[\cdot]$
	$\mid \mathsf{fork}\langle C[\cdot]\rangle$	$\mid \mathsf{join}\langle C[\cdot]\rangle$

Table 9: Øjeblik contexts

adds one unconditional step after reducing a) and that the notion of equivalence takes all Øjeblik contexts into account, Equation 1 can be reduced to the problem of surrogation on variables:

$$x \doteq x.\mathsf{surrogate} \tag{2}$$

However, there is an inherent problem with Equation 2, which is exhibited by the following context that creates a self-alias via method call:

$$C[\cdot] := \mathsf{let}\ x = [\ \mathrm{l}{=}\varsigma(s)s.\mathsf{alias}\langle s\rangle\ ]\ \mathsf{in}\ x.\mathrm{l};\ [\cdot]$$

It holds that $C[x]\Downarrow$, whereas $C[x.\mathsf{surrogate}]\not\Downarrow$. Indeed, in $C[x]\Downarrow$ the evaluation of x returns immediately, while in $C[x.\mathsf{surrogate}]\not\Downarrow$, the request $x.\mathsf{surrogate}$ is never served because it travels into a loop along the self alias chain $x{\gg}x$. The problem in Equation 2 is that we do not check whether the "object before surrogation" is actually reachable. This can be easily done as follows

$$x.\mathsf{ping} \doteq x.\mathsf{surrogate} \tag{3}$$

The equation 3 detects cyclic chains by means of the ping-request which travels to the endpoint of the alias chain possibly starting at x. For the above context, $C[x.\mathsf{ping}]\not\Downarrow$.

In the remainder of the paper, Equation 3 will be referred to as the *safety equation*. In order to fully specify it, we lack the interpretation of the equivalence $\doteq$. A standard way to define *program equivalences* is to compare the convergence behaviour of programs within arbitrary program contexts, as, for example, shown in previous work on the Imperative Object Calculus (IOC) [AC96, GHL97]. This equivalence is usually referred to as *observational congruence* [Mor68]. In our setting, according to Table 9, an *Øjeblik context* $C[\cdot]$ has a single hole $[\cdot]$ that may be filled with an Øjeblik term. In the remainder of the paper, we assume that Øjeblik-contexts always yield well-typed terms when plugging some Øjeblik-term into the hole.

Since we have given a translational semantics for Øjeblik, our program equivalence is based on the encoding $[\![\,\cdot\,]\!]_p^k$. Roughly, the semantics $[\![\,a\,]\!]_p^k$, of an Øjeblik term a is a $\mathrm{L}\pi^+$-process which returns the result on channel p as soon as it knows it. An Øjeblik term *converges* if its semantics is a process which *may* report its result on the channel p.

Definition 7.1 (Convergence) *Given an Øjeblik term a, we write $a\Downarrow$ if $[\![\,a\,]\!]_p^k\Downarrow_p$.*

Definition 7.2 (Behavioural equivalence) *Two Øjeblik terms a and b are* behaviourally equivalent*, written $a \doteq b$, if*

$$C[a]\Downarrow \quad \text{iff} \quad C[b]\Downarrow$$

for all Øjeblik contexts $C[\cdot]$.

7.2 On the absence of self-inflicted surrogation

One of the main observations in [NHKM00] was that the safety equation can not hold in full generality for Øjeblik-contexts, in which the operation $x.\mathsf{surrogate}$ could occur internally. The reason is that, after *internal* surrogation, an object may misuse by intention the old and new references to *itself*. Actually, the advice to avoid internal surrogation is somehow analogous to the fact that programmers, knowing that $x{=}0$, should never use division by x. "Observable internal-surrogations" should be interpreted as *programming errors* and not as a *semantics fault*. On the other hand, "observable external-surrogations" represents a much more serious problem. In this case, (external) object clients can distinguish whether an object has moved or not. Somehow, it corresponds to the case where a program receives x from some other module, so it should be guaranteed that x will never be 0. In [NHKM00], we conjectured that in our semantics *external surrogation is guaranteed to be safe.* Although this is an undecidable criterion [Car95], we may still formalise it in terms of our π-calculus semantics, which is precisely what we do in this Subsection: we formalise the class of Øjeblik-contexts $C[\cdot]$ that will never lead to self-inflicted occurrences of the term $x.\mathsf{surrogate}$, when plugged into the hole.

In our semantics, the computation $[\![\, a \,]\!]_p^k \Rightarrow Z$ of an Øjeblik term a yields a self-inflicted sur-request if $Z \equiv E[\ \overline{s}\langle \mathsf{sur\_}r, k\rangle \mid \mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k_{\mathrm{i}}, \widetilde{t}\,\rangle\ ]$, for some static context $E[\cdot]$ in $\mathrm{L}\pi^+$, with $k{=}k_{\mathrm{i}}$. Since we must ensure that a sur-request *never* leads to internal surrogation, we must quantify over all derivatives of $[\![\, a \,]\!]_p^k$ and check for self-infliction in each of them.

Note that, starting from the term $[\![\, C[x.\mathsf{surrogate}] \,]\!]_p^k$, we should not be concerned with arbitrary sur-requests that appear at top-level during computation, but only with those that "arise from the request in the hole". However, this property is hard to determine for two different reasons: (1) *All* of the names mentioned in a sur-request may be changed dynamically by instantiation: s (due to forwarding), r (due to a call manager protocol), and k (due to pre-processing). (2) We have to consider arbitrarily many duplications of the request in the case that the hole appears, at the level of Øjeblik terms, within in a method body, which leads to replication in the π-calculus semantics. For both reasons, we need a tool to uniquely identify the various incarnations of the request.

Let $\mathsf{operate} \in \{\mathsf{ping}, \mathsf{surrogate}\}$, and let $\mathsf{op} \in \{\mathsf{png}, \mathsf{sur}\}$ denote the corresponding π-calculus labels (c.f. Table 6). We introduce the *additional* Øjeblik labels $\mathsf{operate}^\star \in \{\mathsf{ping}^\star, \mathsf{surrogate}^\star\}$. The intuition is that tagged labels are semantically treated exactly like their untagged counterparts, but can syntactically be distinguished from them. Consequently, we have to adapt the given semantics to take this into account. Table 10 presents the required straightforward additions, where we use the tagged π-calculus labels $\mathsf{op}^\star \in \{\mathsf{png}^\star, \mathsf{sur}^\star\}$, respectively: the individual clauses of the tagged semantics, written $[\![\![\, \cdot \,]\!]\!]_p^k$, are just copies of the clauses for the untagged requests.

As a result, both tagged and untagged requests can be sent to object and alias managers; object managers ignore the tagging information of requests and treat $\mathsf{op}^\star$-and op-requests identically, but alias managers preserve the tagging information since they simply forward requests. We also add a tag to all parameterised definitions and abbreviations when considering the tagged semantics, for instance, $\mathsf{OM}^\star$, $\mathsf{AM}^\star$, $\mathsf{pingO}^\star$ and $\mathsf{surO}^\star$ are defined as expected. Notice that the semantics is not affected by including tagging information. As a consequence, all results proved for the untagged semantics are valid for the tagged semantics as well.

Lemma 7.3 *Let x be an Øjeblik variable and $C[\cdot]$ an untagged Øjeblik context. Then:*

$$C[x.\mathsf{operate}]\Downarrow \textit{ iff } [\![\![\, C[x.\mathsf{operate}^\star] \,]\!]\!]_p^k \Downarrow_p .$$

PROOF. The proof is in two steps:

$$[\![\, C[x.\mathsf{operate}] \,]\!]_p^k \Downarrow_p \ \text{ iff } \ [\![\![\, C[x.\mathsf{operate}] \,]\!]\!]_p^k \Downarrow_p \ \text{ iff } \ [\![\![\, C[x.\mathsf{operate}^\star] \,]\!]\!]_p^k \Downarrow_p .$$

The first step compares the convergence behaviour of untagged requests—note that $C[x.\mathsf{operate}]$ is untagged by assumption—with respect to the tagged and the untagged semantics. On untagged requests, the tagged and the untagged semantics behave exactly the same. The second step

$$\llbracket\, a.\mathsf{surrogate}^\star \,\rrbracket_p^k \stackrel{\text{def}}{=} (\boldsymbol{\nu} q)\ (\ \llbracket\, a \,\rrbracket_q^k \ \mid\ q(y,i)\,.\,\overline{y}\langle \mathsf{sur}^\star\_p, i\rangle\ )$$

$$\llbracket\, a.\mathsf{ping}^\star \,\rrbracket_p^k \stackrel{\text{def}}{=} (\boldsymbol{\nu} q)\ (\ \llbracket\, a \,\rrbracket_q^k \ \mid\ q(y,i)\,.\,\overline{y}\langle \mathsf{png}^\star\_p, i\rangle\ )$$

$$\begin{array}{l}
\mathsf{OM}^\star_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k_{\mathrm{i}}, \widetilde{t}\,\rangle \stackrel{\text{def}}{=} s(l,k).(\boldsymbol{\nu} k^*)\Big(\\
\quad \mathsf{if}\ [k{=}k_{\mathrm{i}}]\ \mathsf{then} \\
\qquad \mathsf{case}\ l\ \mathsf{of}\quad \ldots\quad :\quad \ldots \\
\qquad\qquad \mathsf{sur}_(r) : \mathsf{OM}^\star_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k^*, \widetilde{t}\,\rangle \mid \llbracket\, s.\mathsf{alias}\langle s.\mathsf{clone}\rangle \,\rrbracket_r^{k^*}\ ; \\
\qquad\qquad \mathsf{png}_(r) : \mathsf{OM}^\star_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k^*, \widetilde{t}\,\rangle \mid \llbracket\, s \,\rrbracket_r^{k^*}\ ; \\
\qquad\qquad \mathsf{sur}^\star_(r) : \mathsf{OM}^\star_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k^*, \widetilde{t}\,\rangle \mid \llbracket\, s.\mathsf{alias}\langle s.\mathsf{clone}\rangle \,\rrbracket_r^{k^*}\ ; \\
\qquad\qquad \mathsf{png}^\star_(r) : \mathsf{OM}^\star_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k^*, \widetilde{t}\,\rangle \mid \llbracket\, s \,\rrbracket_r^{k^*} \\
\quad \mathsf{elif}\ [k{=}k_{\mathrm{e}}]\ \mathsf{then} \\
\qquad \mathsf{OM}^\star_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k^*, \widetilde{t}\,\rangle \,\big|\, \mathsf{case}\ l\ \mathsf{of}\quad \ldots\quad :\quad \ldots \\
\qquad\qquad \mathsf{sur}_(r) : \mathsf{CM}[\, \llbracket\, s.\mathsf{alias}\langle s.\mathsf{clone}\rangle \,\rrbracket_{r^*}^{k^*}\ ]\ ; \\
\qquad\qquad \mathsf{png}_(r) : \mathsf{CM}[\, \llbracket\, s \,\rrbracket_{r^*}^{k^*}\ ]\ ; \\
\qquad\qquad \mathsf{sur}^\star_(r) : \mathsf{CM}[\, \llbracket\, s.\mathsf{alias}\langle s.\mathsf{clone}\rangle \,\rrbracket_{r^*}^{k^*}\ ]\ ; \\
\qquad\qquad \mathsf{png}^\star_(r) : \mathsf{CM}[\, \llbracket\, s \,\rrbracket_{r^*}^{k^*}\ ] \\
\quad \mathsf{else}\ \mathsf{OM}^\star_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k_{\mathrm{i}}, \widetilde{t}\,\rangle \,\big|\, m_{\mathrm{e}}.(\ \overline{s}\langle l, k_{\mathrm{e}}\rangle \ \mid\ \overline{m_{\mathrm{i}}}k\ )\Big)
\end{array}$$

$$\begin{array}{l}
\mathsf{AM}^\star_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k_{\mathrm{i}}, s_{\mathrm{a}}\,\rangle \stackrel{\text{def}}{=} s(l,k).(\boldsymbol{\nu} k^*)\Big(\\
\quad \mathsf{if}\ [k{=}k_{\mathrm{i}}]\ \mathsf{then} \\
\qquad \mathsf{case}\ l\ \mathsf{of}\quad \ldots\quad :\quad \ldots \\
\qquad\qquad \mathsf{sur}_(r) : \mathsf{AM}^\star_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k^*, s_{\mathrm{a}}\,\rangle \mid \overline{s_{\mathrm{a}}}\langle l, k\rangle\ ; \\
\qquad\qquad \mathsf{png}_(r) : \mathsf{AM}^\star_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k^*, s_{\mathrm{a}}\,\rangle \mid \overline{s_{\mathrm{a}}}\langle l, k\rangle\ ; \\
\qquad\qquad \mathsf{sur}^\star_(r) : \mathsf{AM}^\star_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k^*, s_{\mathrm{a}}\,\rangle \mid \overline{s_{\mathrm{a}}}\langle l, k\rangle\ ; \\
\qquad\qquad \mathsf{png}^\star_(r) : \mathsf{AM}^\star_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k^*, s_{\mathrm{a}}\,\rangle \mid \overline{s_{\mathrm{a}}}\langle l, k\rangle \\
\quad \mathsf{elif}\ [k{=}k_{\mathrm{e}}]\ \mathsf{then}\ \mathsf{AM}^\star_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k^*, s_{\mathrm{a}}\,\rangle \ \mid\ m_{\mathrm{i}}(k).(\ \overline{s_{\mathrm{a}}}\langle l, k\rangle \ \mid\ \overline{m_{\mathrm{e}}}\ ) \\
\quad \mathsf{else}\qquad \mathsf{AM}^\star_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k_{\mathrm{i}}, s_{\mathrm{a}}\,\rangle \ \mid\ m_{\mathrm{e}}.(\ \overline{s}\langle l, k_{\mathrm{e}}\rangle \ \mid\ \overline{m_{\mathrm{i}}}k\ )\Big)
\end{array}$$

Table 10: Translational semantics — Additional tagged clauses

compares the convergence behaviour of a tagged term and its untagged counterpart with respect to the tagged semantics. By definition, the tagged semantics treats tagged and untagged requests in exactly the same manner. $\square$

Tagging helps us to detect all "requests arising from the hole".

Definition 7.4 (External Contexts) *Let x be a variable and $C[\cdot]$ an untagged Øjeblik context. Then, $C[\cdot]$ is called* external *for $x.\mathsf{surrogate}$, if whenever*

$$[\![\, C[x.\mathsf{surrogate}^\star]\,]\!]_p^k \;\Rightarrow_\equiv E[\; \overline{s}\langle \mathsf{sur}^\star\_r, k\rangle \mid \mathsf{OM}^\star_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k_{\mathrm{i}}, \widetilde{t}\,\rangle \;]$$

it holds that $k \neq k_{\mathrm{i}}$.

We replay the definition using ping instead of $\mathsf{surrogate}$. By definition of the semantics, an Øjeblik context $C[\cdot]$ is then external for $x.\mathsf{surrogate}$ if and only if it is external for $x.\mathsf{ping}$. For convenience, by abuse, we simply call $C[\cdot]$ to be *external for x.*

8 On the safety of surrogation

In this section, we prove that that

$$C[x.\mathsf{ping}]\Downarrow \;\text{ iff }\; C[x.\mathsf{surrogate}]\Downarrow$$

under the assumption that $C[\cdot]$ will never lead to self-inflicted occurrences of $x.\mathsf{surrogate}$. In Subsection 8.1, we study the behavior of the committed object managers $\mathsf{pingO}_{\mathbb{O}}\langle\, s, \dots\rangle$ and $\mathsf{surO}_{\mathbb{O}}\langle\, s, \dots\rangle$, as defined at the end of Subsection 6.2, and prove them algebraically to be barbed Γ-equivalent. In Subsection 8.2, we then give the formal proof for the safety of external surrogations by iteratively simulating convergence sequences for the proof goal above. Finally in Subsection 8.3, we give a static type system that guarantees that surrogations will always be external.

8.1 On committing external surrogations

By Lemma 6.6, when an object manager commits to either a png or a sur request, we get the processes $\mathsf{pingO}_{\mathbb{O}}\langle\, s, r, k, \widetilde{t}, \widetilde{v}\,\rangle$ or $\mathsf{surO}_{\mathbb{O}}\langle\, s, r, k, \widetilde{t}, \widetilde{v}\,\rangle$, respectively. These processes also represent (the state of) the object manager before and after external surrogation, respectively; recall that $\mathsf{pingO}_{\mathbb{O}}\langle\, s, \dots\rangle$ just tells us that the object manager at the end of the chain was reachable. Notice that due to the use of nonces (c.f. page 21) in the implementation of the object and alias managers, in both processes $\mathsf{pingO}_{\mathbb{O}}\langle\, s, \dots\rangle$ and $\mathsf{surO}_{\mathbb{O}}\langle\, s, \dots\rangle$ the key k^* is fresh and therefore different from any key appearing in the process $\mathsf{PP}_{\mathbb{O}}\langle\, s, \widetilde{n}, \widetilde{v}\,\rangle$ contained in both $\mathsf{pingO}_{\mathbb{O}}\langle\, s, \dots\rangle$ and $\mathsf{surO}_{\mathbb{O}}\langle\, s, \dots\rangle$.

In the following we show that processes $\mathsf{pingO}_{\mathbb{O}}\langle\, s, \dots\rangle$ and $\mathsf{surO}_{\mathbb{O}}\langle\, s, \dots\rangle$ are related by typed barbed equivalence $\simeq_{\Gamma;s}$ (Definition 2.10).

Theorem 8.1 *Let Γ be a type environment with $\Gamma \vdash \mathsf{surO}_{\mathbb{O}}\langle\, s, r, k, \widetilde{t}, \widetilde{v}\,\rangle, \mathsf{pingO}_{\mathbb{O}}\langle\, s, r, k, \widetilde{t}, \widetilde{v}\,\rangle$. Then:*

$$\mathsf{surO}_{\mathbb{O}}\langle\, s, r, k, \widetilde{t}, \widetilde{v}\,\rangle \simeq_{\Gamma;s} \mathsf{pingO}_{\mathbb{O}}\langle\, s, r, k, \widetilde{t}, \widetilde{v}\,\rangle.$$

The proof of Theorem 8.1 requires several strong lemmas. The proofs of the latter can be found in Appendix A. In all the lemmas below the well-typedness requirement is necessary to ensure that (i) the environment sends along the object reference s only values of the right type, (ii) the environment never uses channel s in input.

Lemma 8.2 proves that surrogation results in an alias pointing to a clone of the old object. The proof relies on the nonces used in the implementation of both object and alias managers, which control the interference with the environment.

Lemma 8.2 *If Γ is a suitable type environment for the processes below, then:*

$$\mathsf{surO}_{\mathbb{O}}\langle\, s, r, k, \widetilde{t}, \widetilde{v}\,\rangle \approx_{\Gamma;s} (\boldsymbol{\nu} s^*)\big((\boldsymbol{\nu} k_{\mathrm{i}})\, \mathsf{freeA}_{\mathbb{O}}\langle\, s, k_{\mathrm{i}}, s^*, \widetilde{v}\,\rangle \,\big|\, \mathsf{newO}_{\mathbb{O}}\langle\, s^*, \widetilde{t}\,\rangle \,\big|\, \overline{r}\langle s^*, k\rangle\big).$$

Lemma 8.3 proves that the alias manager appearing in Lemma 8.2 behaves as a forwarder. This will allow us to apply the theory of Lπ.

Lemma 8.3 *Let $\widetilde{v} := v_1 .. v_n$, and $v_j := \langle\, l_j, k_j \,\rangle$ for $1 \leq j \leq n$. If Γ is a suitable type environment for the processes below, then:*

$$(\boldsymbol{\nu} k_{\mathrm{i}})\, \mathsf{freeA}_{\mathbb{O}}\langle\, s, k_{\mathrm{i}}, s^*, \widetilde{v} \,\rangle \approx_{\Gamma;s} s \triangleright s^* \,\Big|\, \prod_{1 \leq j \leq n} \overline{s^*} v_j .$$

Notice that without the well-typedness hypothesis, after having received a wrong value along s, the two processes above would have a different behaviour.

Lemma 8.4 uses the algebraic law of Lπ^+ of Lemma 2.15. Note that the proof of Lemma 8.4 is not a trivial application of Lemma 2.15.

Lemma 8.4 *Let P be a process and s a channel such that $s \notin \mathrm{fc}(P)$. If Γ is a suitable type environment for the processes below, then:*

$$(\boldsymbol{\nu} s^*)\,\big(\, s \triangleright s^* \mid P \,\big) \simeq_{\Gamma;s} P\{^{s}/_{s^*}\}.$$

Lemma 8.5 proves that pre-processing external requests does not preclude other requests.

Lemma 8.5 *Let $\widetilde{v} := v_1 .. v_n$ with $v_j := \langle\, l_j, k_j \,\rangle$ and $k_j \neq k_{\mathrm{i}}$ for $1 \leq j \leq n$. If Γ is a suitable type environment for the processes below, then:*

$$\prod_{1 \leq j \leq n} \overline{s} v_j \,\Big|\, \mathsf{newO}_{\mathbb{O}}\langle\, s, \widetilde{t} \,\rangle \approx_{\Gamma;s} (\boldsymbol{\nu} k_{\mathrm{i}})\, \mathsf{freeO}_{\mathbb{O}}\langle\, s, k_{\mathrm{i}}, \widetilde{t}, \widetilde{v} \,\rangle .$$

Lemma 8.6 is a technical lemma involving two confluent reductions.

Lemma 8.6 *Let $\widetilde{v} := v_1 .. v_n$ with $v_j := \langle\, l_j, k_j \,\rangle$ and $k_j \neq k_{\mathrm{i}}$ for $1 \leq j \leq n$. If Γ is a suitable type environment for the processes below, then:*

$$(\boldsymbol{\nu} k_{\mathrm{i}})\, \mathsf{freeO}_{\mathbb{O}}\langle\, s, k_{\mathrm{i}}, \widetilde{t}, \widetilde{v} \,\rangle \,\big|\, \overline{r}\langle s, k\rangle \approx_{\Gamma} \mathsf{pingO}_{\mathbb{O}}\langle\, s, r, k, \widetilde{t}, \widetilde{v} \,\rangle .$$

Proof of Theorem 8.1 PROOF. We recall that relations $\approx_{\Gamma}$ and $\approx_{\Gamma;s}$ imply $\simeq_{\Gamma;s}$. By subsequently applying Lemmas 8.2, 8.3, 8.4, 8.5, and 8.6 we have:

$$\begin{aligned}
&\mathsf{surO}_{\mathbb{O}}\langle\, s, r, k, \tilde{t}, \widetilde{v} \,\rangle \\
&\quad \simeq_{\Gamma;s} \quad (\boldsymbol{\nu} s^*)\,\big((\boldsymbol{\nu} k_{\mathrm{i}})\,(\mathsf{freeA}_{\mathbb{O}}\langle\, s, k_{\mathrm{i}}, s^*, \widetilde{v} \,\rangle) \;\mid\; \mathsf{newO}_{\mathbb{O}}\langle\, s^*, \tilde{t} \,\rangle \;\mid\; \overline{r}\langle s^*, k\rangle\big) \\
&\quad \simeq_{\Gamma;s} \quad (\boldsymbol{\nu} s^*)\,\big(\, s \triangleright s^* \;\mid\; \prod_{1 \leq j \leq n} \overline{s^*} v_j \;\mid\; \mathsf{newO}_{\mathbb{O}}\langle\, s^*, \tilde{t} \,\rangle \;\mid\; \overline{r}\langle s^*, k\rangle \,\big) \\
&\quad \simeq_{\Gamma;s} \quad \prod_{1 \leq j \leq n} \overline{s} v_j \;\mid\; \mathsf{newO}_{\mathbb{O}}\langle\, s, \tilde{t} \,\rangle \;\mid\; \overline{r}\langle s, k\rangle \\
&\quad \simeq_{\Gamma;s} \quad (\boldsymbol{\nu} k_{\mathrm{i}})\, \mathsf{freeO}_{\mathbb{O}}\langle\, s, k_{\mathrm{i}}, \widetilde{t}, \widetilde{v} \,\rangle \;\mid\; \overline{r}\langle s, k\rangle \\
&\quad \simeq_{\Gamma;s} \quad \mathsf{pingO}_{\mathbb{O}}\langle\, s, r, k, \tilde{t}, \tilde{v} \,\rangle .
\end{aligned}$$

□

8.2 External Surrogation is Safe

Based on the knowledge of Theorem 8.1 that the committed object managers $\mathsf{pingO}_{\mathbb{O}}\langle\, s, \ldots \rangle$ and $\mathsf{surO}_{\mathbb{O}}\langle\, s, \ldots \rangle$ are equivalent, we proceed to construct simulation sequences up to this equivalence. More precisely, whenever needed we may replace one of the managers by the other, because typed barbed equivalence provides us with the same convergence behaviour in all static contexts.

Theorem 8.7 (Safety) *Let x be an object variable and $C[\cdot]$ an (untagged) well-typed context in Øjeblik. If $C[\cdot]$ is external for x, then*

$$C[x.\mathsf{ping}]\Downarrow \text{ iff } C[x.\mathsf{surrogate}]\Downarrow .$$

Proof. By Lemma 7.3 our proof obligation is equivalent to:

$$[\![\, C[x.\mathsf{ping}^\star] \,]\!]_p^k \Downarrow_p \quad \text{iff} \quad [\![\, C[x.\mathsf{surrogate}^\star] \,]\!]_p^k \Downarrow_p .$$

This allows us to make use of the assumption on context $C[\cdot]$.

Since the semantics $[\![\cdot]\!]_p^k$ is compositional, there is an $\mathrm{L}\pi^+$ context $D[\cdot]$ and names y, j, q, such that $[\![\, C[x.\mathsf{operate}^\star] \,]\!]_p^k = D[\overline{y}\langle \mathsf{op}^\star\_q, j\rangle]$, where $D[\cdot]$ itself does not contain any message carrying a tagged request. Since the translation preserves well-typedness (c.f. Proposition 6.2) there is an $\mathrm{L}\pi^+$ typing Γ such that $\Gamma \vdash D[\overline{y}\langle \mathsf{op}^\star\_q, j\rangle]$. We prove that

$$D[\overline{y}\langle \mathsf{png}^\star\_q, j\rangle]\Downarrow_p \quad \text{iff} \quad D[\overline{y}\langle \mathsf{sur}^\star\_q, j\rangle]\Downarrow_p$$

and concentrate on the implication from right to left. The converse is analogous.

Assume that $D[\overline{y}\langle \mathsf{sur}^\star\_q, j\rangle] \Downarrow_p$. If $D[N]\Downarrow_p$ for every process N, then this is also the case for $N = \overline{y}\langle \mathsf{png}^\star\_q, j\rangle$; otherwise, the $\mathsf{sur}^\star$-request must contribute to the barb. Therefore, we assume $D[\overline{y}\langle \mathsf{sur}^\star\_q, j\rangle] \;\Rightarrow\; P \downarrow_p$ and show that there is a corresponding sequence $D[\overline{y}\langle \mathsf{png}^\star\_q, j\rangle] \;\Rightarrow_{\simeq_\Gamma}\; Q \downarrow_p$ where $Q = P[{}^{\mathsf{png}^\star}\!/_{\mathsf{sur}^\star}]$. Since typed barbed equivalence $\simeq_\Gamma$ and relabelling preserve convergence, this suffices.

According to the discussion in Section 6.2, a reduction step due to an external request is *committing*, if it represents the consumption of a pre-processed request by an object manager. Now, we combine this knowledge with the fact that we have to concentrate on surrogation requests arising from the hole within the reduction sequence $D[\overline{y}\langle \mathsf{sur}^\star\_q, j\rangle] \;\Rightarrow\; P \downarrow_p$ and call *significant* ($\rightarrow_{\mathrm{s}}$) precisely those steps that exhibit the commitment to a $\mathsf{sur}^\star$-request. All the other steps can be considered *insignificant* because—as we show during the proof—they can be mimicked in a straightforward way by the $\mathsf{png}^\star$-ed counterpart.

Whenever $P \xrightarrow{\tau} P'$, we know that either

1. $P \equiv (\boldsymbol{\nu}\widetilde{z})\,(\, \overline{w}v \mid w(x).R \mid M \,)$ and $P' \equiv (\boldsymbol{\nu}\widetilde{z})\,(\, R\{{}^{v}\!/_{x}\} \mid M \,)$, or
2. $P \equiv (\boldsymbol{\nu}\widetilde{z})\,(\, \overline{w}v \mid\, !\, w(x).R \mid M \,)$ and
   $P' \equiv (\boldsymbol{\nu}\widetilde{z})\,(\, R\{{}^{v}\!/_{x}\} \mid\, !\, w(x).R \mid M \,)$.

A silent move $P \xrightarrow{\tau} P'$ (decomposed as above) is called

significant if case 1 applies where $\overline{w}v = \overline{s}\langle \mathsf{sur}^\star\_q, k_{\mathrm{e}}\rangle$ and
$w(x).R = \mathsf{OM}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k_{\mathrm{i}}, \widetilde{t}\,\rangle$. We denote these $P \;\rightarrow_{\mathrm{s}}\; P'$.

insignificant if either

- case 2 applies, or
- case 1 applies where v does not carry a $\mathsf{sur}^\star$-request, or
- case 1 applies where $\overline{w}v = \overline{s}\langle \mathsf{sur}^\star\_q, j\rangle$ and
  $w(x).R = \mathsf{AM}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k_{\mathrm{i}}, \widetilde{t}\,\rangle$, or
- case 1 applies where $\overline{w}v = \overline{s}\langle \mathsf{sur}^\star\_q, j\rangle$ and
  $w(x).R = \mathsf{OM}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k_{\mathrm{i}}, \widetilde{t}\,\rangle$ with $k_{\mathrm{i}} \neq j \neq k_{\mathrm{e}}$.

We denote this as $P \;\rightarrow_{\mathrm{i}}\; P'$.

The missing case of $\overline{w}v = \overline{s}\langle \mathsf{sur}^\star\_q, k\rangle$ and $w(x).R = \mathsf{OM}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k_{\mathrm{i}}, \widetilde{t}\,\rangle$ with $k = k_{\mathrm{i}}$ is excluded by the assumption that $C[\cdot]$ is external for x (c.f. Definition 7.4). Note that starting with a $\mathsf{sur}^\star$-request in the hole, we will never encounter $\mathsf{png}^\star$-requests during the computation, and vice versa.

Now, we apply the classification of reduction steps to the given reduction sequence $D[\overline{y}\langle \mathsf{sur}^\star\_q, j\rangle] \;\Rightarrow\; P \downarrow_p$, assuming that it contains $d > 0$ significant steps (if $d = 0$, then $D[N]\Downarrow_p$ for all processes N):

$$D[\overline{y}\langle \mathsf{sur}^\star\_q, j\rangle] =$$

$$\begin{array}{lllllllllll}
P_{1,1} & \rightarrow_{\mathrm{i}} & P_{1,2} & \rightarrow_{\mathrm{i}} & \cdots & \rightarrow_{\mathrm{i}} & P_{1,n_1} & \rightarrow_{\mathrm{s}} & P_1 & = & P_{2,1} \\
P_{2,1} & \rightarrow_{\mathrm{i}} & P_{2,2} & \rightarrow_{\mathrm{i}} & \cdots & \rightarrow_{\mathrm{i}} & P_{2,n_2} & \rightarrow_{\mathrm{s}} & P_2 & = & P_{3,1} \\
\vdots & & \vdots & & & & \vdots & & \vdots & & \\
P_{d,1} & \rightarrow_{\mathrm{i}} & P_{d,2} & \rightarrow_{\mathrm{i}} & \cdots & \rightarrow_{\mathrm{i}} & P_{d,n_d} & \rightarrow_{\mathrm{s}} & P_d & = & P_{d+1,1} \\
P_{d+1,1} & \rightarrow_{\mathrm{i}} & P_{d+1,2} & \rightarrow_{\mathrm{i}} & \cdots & \rightarrow_{\mathrm{i}} & P_{d+1,n_{d+1}} & = & P\!\downarrow_p & &
\end{array}$$

By (the tagged counterpart of) Lemma 6.6 it holds that:

$$P_h \quad \equiv \quad (\boldsymbol{\nu}\widetilde{z}_h)\,(\ M_h \mid \mathsf{surO}^{\star}_{\mathbb{O}}\langle\, s_h, q_h, k_h, \widetilde{t}_h, \widetilde{v}_h \,\rangle\ )$$

for some $\widetilde{z}_h$ and M_h. Now, we simulate the previous reduction sequence, which uses $\mathsf{sur}^{\star}$-requests, but now using $\mathsf{png}^{\star}$-requests and proceeding up to structural equivalence *and* barbed equivalence.

$D[\overline{y}\langle \mathsf{png}^{\star}\_q, j\rangle] =$

$$\begin{array}{lllllllllllll}
Q_{1,1} & \rightarrow_{\mathrm{i}} & Q_{1,2} & \rightarrow_{\mathrm{i}} & \cdots & \rightarrow_{\mathrm{i}} & Q_{1,n_1} & \rightarrow_{\mathrm{s}} & Q_1 & \simeq_\Gamma & \widehat{Q}_1 & \equiv & Q_{2,1} \\
Q_{2,1} & \rightarrow_{\mathrm{i}} & Q_{2,2} & \rightarrow_{\mathrm{i}} & \cdots & \rightarrow_{\mathrm{i}} & Q_{2,n_2} & \rightarrow_{\mathrm{s}} & Q_2 & \simeq_\Gamma & \widehat{Q}_2 & \equiv & Q_{3,1} \\
\vdots & & \vdots & & & & \vdots & & \vdots & & \vdots & & \vdots \\
Q_{d,1} & \rightarrow_{\mathrm{i}} & Q_{d,2} & \rightarrow_{\mathrm{i}} & \cdots & \rightarrow_{\mathrm{i}} & Q_{d,n_d} & \rightarrow_{\mathrm{s}} & Q_d & \simeq_\Gamma & \widehat{Q}_d & \equiv & Q_{d+1,1} \\
Q_{d+1,1} & \rightarrow_{\mathrm{i}} & Q_{d+1,2} & \rightarrow_{\mathrm{i}} & \cdots & \rightarrow_{\mathrm{i}} & Q_{d+1,n_{d+1}} & \stackrel{\mathrm{def}}{=} & Q{\downarrow_p} & & & &
\end{array}$$

where:

$$Q_{h,g} \quad \stackrel{\mathrm{def}}{=} \quad P_{h,g}[^{\mathsf{png}^{\star}}/_{\mathsf{sur}^{\star}}]$$

The insignificant reduction steps $\rightarrow_{\mathrm{i}}$ exist because of Lemma 8.8. The significant reduction steps $Q_{h,n_h} \rightarrow_{\mathrm{s}} Q_h$ are analogous to their counterparts $P_{h,n_h} \rightarrow_{\mathrm{s}} P_h$. Precisely, by (the tagged counterpart of) Lemma 6.6, they give rise (up to structural equivalence) to a $\mathsf{pingO}^{\star}$ instead of a $\mathsf{surO}^{\star}$, that is:

$$Q_h \quad \equiv \quad (\boldsymbol{\nu}\widetilde{z}_h)\,(\ M_h \mid \mathsf{pingO}^{\star}_{\mathbb{O}}\langle\, s_h, q_h, j_h, \widetilde{t}_h, \widetilde{v}_h \,\rangle\ )[^{\mathsf{png}^{\star}}/_{\mathsf{sur}^{\star}}].$$

The processes $\widehat{Q}_h$ are defined as follows:

$$\widehat{Q}_h \quad \stackrel{\mathrm{def}}{=} \quad (\boldsymbol{\nu}\widetilde{z}_h)\,(\ M_h \mid \mathsf{surO}^{\star}_{\mathbb{O}}\langle\, s_h, q_h, j_h, \widetilde{t}_h, \widetilde{v}_h \,\rangle\ )[^{\mathsf{png}^{\star}}/_{\mathsf{sur}^{\star}}]$$

The relations $Q_h \simeq_\Gamma \widehat{Q}_h$ hold by application of (the tagged counterparts of) Theorem 8.1 and Lemma 6.4, and since $\simeq_\Gamma$ is preserved by relabelling $[^{\mathsf{png}^{\star}}/_{\mathsf{sur}^{\star}}]$. The relations $\widehat{Q}_h \equiv Q_{h+1,1}$ hold since

$$\widehat{Q}_h \equiv P_h[^{\mathsf{png}^{\star}}/_{\mathsf{sur}^{\star}}] = P_{h+1,1}[^{\mathsf{png}^{\star}}/_{\mathsf{sur}^{\star}}] \stackrel{\mathrm{def}}{=} Q_{h+1,1}.$$

Lemma 8.8 *Let a be an Øjeblik term possibly containing a tagged request. If* $[\![\, a \,]\!]^k_p \Rightarrow R \rightarrow_{\mathrm{i}} R'$, *then* $R[^{\mathsf{png}^{\star}}/_{\mathsf{sur}^{\star}}] \rightarrow_{\mathrm{i}} R'[^{\mathsf{png}^{\star}}/_{\mathsf{sur}^{\star}}]$ *and* $R[^{\mathsf{sur}^{\star}}/_{\mathsf{png}^{\star}}] \rightarrow_{\mathrm{i}} R'[^{\mathsf{sur}^{\star}}/_{\mathsf{png}^{\star}}]$.

Proof. By case analysis on the four different shapes of insignificant steps. In each of them, the relabelling distributes over the components of R, which allows us afterwards to derive the corresponding reduction step. □

This concludes the proof of Theorem 8.7. □

8.3 Typing for External Surrogation

Since only external surrogations are safe, we look for some way to statically ensure that this is the case. To avoid such unwanted situations, the most obvious case is s.operate, where s is the self-variable of the immediately enclosing method. A less obvious case is a.operate, where a may evaluate to the current self or to the self of a node in an alias chain leading to the current self. In the least obvious case, concurrent threads may render the evaluation of a nondeterministic, such that it may or may not evaluate to the current self.

At first, it might seem hopeless to come up with a good way of ensuring that an operation is external. However, if a evaluates to the current self, or a node in an alias chain leading to the current self, then a *must* have the same type as the type of the current self. This implies, that if we ensure that the type of a is not the same as for the current self, then a.operate cannot result in operate being an internal operation. Such a check can be incorporated into the type system of Table 5. In the new system, judgements are now on the form $\Gamma \vdash_D a{:}A$ where D denotes the type of the self variable for the method enclosing a. In Table 11 we present the modifications of the type system; the rules missing are as the ones in Table 5 with $\vdash$ replaced by $\vdash_D$.

$$\text{(T-Obj)}\ \frac{\forall j{\in}J \quad \Gamma, s_j{:}A, \tilde{x}_j{:}\widetilde{B}_j \vdash_A b_j{:}\widehat{B}_j \qquad A = [\mathrm{l}_j{:}\widetilde{B}_j{\rightarrow}\widehat{B}_j]_{j\in J}}{\Gamma \vdash_D [\mathrm{l}_j : \varsigma(s_j{:}A, \tilde{x}_j{:}\widetilde{B}_j)b_j]_{j\in J} : A}$$

$$\text{(T-Upd)}\ \frac{\Gamma \vdash_D a{:}A \qquad A = [\mathrm{l}_j{:}\widetilde{B}_j{\rightarrow}\widehat{B}_j]_{j\in J} \qquad \Gamma, s_k{:}A, \tilde{x}_k{:}\widetilde{B}_k \vdash_A b_k{:}\widehat{B}_k \qquad k{\in}J}{\Gamma \vdash_D a.\mathrm{l}_k{\Leftarrow}\varsigma(s_k{:}A, \tilde{x}_k{:}\widetilde{B}_k)b_k : A}$$

$$\text{(T-Sur)}\ \frac{\Gamma \vdash_D a{:}A \qquad A = [\mathrm{l}_j{:}A_j]_{j\in J} \qquad D \neq A}{\Gamma \vdash_D a.\mathsf{surrogate} : A}$$

$$\text{(T-Fork)}\ \frac{\Gamma \vdash_{\mathrm{Thr}(A)} a{:}A}{\Gamma \vdash_D \mathsf{fork}\langle a\rangle : \mathrm{Thr}(A)}$$

Table 11: Typing Rules Ensuring External Surrogate Operations

Theorem 8.9 *If* $\Gamma \vdash_{\mathrm{Thr}(A)} C[x.\mathsf{surrogate}] : A$*, then* $C[\cdot]$ *is external for* x*.*

Proof. [Sketch] We proceed in four steps. (1) Refine the typing of keys according to the Øjeblik object (or thread) type that they are used with. When a manager hands out a key k_{i}, the latter is always annotated with the same type as the one carried by the self-channel of the manager. (2) Observe that a request $\overline{s}\langle l, k\rangle$ must be external if the type of k does not match the type of s. (3) Observe, that in a request the types of k and s never change. (4) Prove that if $\Gamma \vdash_D x.\mathsf{surrogate}$, then $[\![\,\Gamma\,]\!] \vdash [\![\,x.\mathsf{surrogate}\,]\!]_p^k$ for $[\![\,\Gamma\,]\!](x){=}[\![\,A\,]\!]$, $[\![\,\Gamma\,]\!](k){=}\mathbf{K}_B$ with $A \neq B$. □

Let us adapt the notion of behavioral equivalence of Definition 7.2 to take into account the proposed type system. This is done in a standard fashion by only considering for a term P only contexts $C[\cdot]$ such that $C[P]$ is typable.

Definition 8.10 (Typed Equivalence) *Two Øjeblik terms* a, b *with* $\Gamma \vdash a, b : A$ *for some* Γ *and* A *are* typed equivalent*, written* $a \doteq^{\vdash}_{\mathrm{ext}} b$*, if* $C[a]\Downarrow$ *iff* $C[b]\Downarrow$ *for all contexts* $C[\cdot]$ *with* $\Delta \vdash_{\mathrm{Thr}(B)} C[a], C[b] : B$ *for some* Δ *and* B*.*

Corollary 8.11 *If* x *is an object variable, then* $x.\mathsf{ping} \doteq^{\vdash}_{\mathrm{ext}} x.\mathsf{surrogate}$*.*

9 Conclusion

In this paper, we have outlined a formal proof of the safety of object surrogation, a distribution-free abstraction of object migration, for a dynamically defined class of program contexts that render surrogations always external. Moreover, for improved feasibility of the use of surrogation in programming, we have provided a simple static type system that guarantees that all well-typed occurrences of surrogation are indeed external.

Since we have carried out this work on an abstraction of migration, it is required to ask for the meaningfulness of our result for migration itself. Since Obliq is a lexically-scoped distributed language, our results tell that any well-typed program—assuming that our type system is lifted to Obliq, and that Obliq is equipped with a forwarder model, as in [NHKM00]—will never observe a difference in the view of may-convergence between an object before and after surrogation, unless one of the involved distribution sites fails, and unless contexts could retrieve (by language primitives) the actual location of an object.

A natural potential criticism on results based on a semantics by translation into another formalism is that it is sometimes hard to evaluate what the results actually say about the original subject. On the one hand, as in our case, where we have also developed several direct operational semantics for Øjeblik, the question for some formal correspondence result among the semantics by translation and the direct semantics arises. On the other hand, one may ask to carry out the proofs on the direct semantics instead of employing some other lower-level formalism. However, we found it very natural and useful to develop two semantics at different abstraction levels hand-in-hand. In fact, most of the examples of unsafe surrogation were discovered by means of the π-calculus semantics, and only then "verified" in the direct semantics. Moreover, since we have developed both levels of semantics in lock-step, we have a good basis for formalizing their interrelation. Finally, in contrast to our abstract configuration-style semantics for closed terms only, the π-calculus provides indeed a very rich set of approved reasoning tools that make the life of a theorem prover much easier, as exemplified by Kleist and Sangiorgi [KS98], and also in this paper.

Other strands of future work are twofold. One is to continue to develop and exploit semantics for the Obliq-style of object migration, and to use our semantics also to prove other equations on Obliq-programs. For example, also equations like $\mathsf{join}\langle\mathsf{fork}\langle a\rangle\rangle{=}a$ do only hold under certain conditions inflicted by self-infliction. Another strand is to try to carry over our results to settings that are not based on the notion of serialization via self-infliction, but rather reentrant mutexes, as in Java.

Acknowledgements

We thank Luca Cardelli for several useful discussions on Obliq. We also thank Giuseppe Castagna, Rocco De Nicola, Joachim Parrow, and Davide Sangiorgi for comments on an early draft.

A Proofs

A.1 Proof of Lemma 2.14

PROOF. We show that the relation

$$\mathcal{S} = \{ \big(Q\{p/q\} , (\boldsymbol{\nu} q{:}\mathbf{C}(T))\, (Q \mid q \triangleright p) \big) \ : q \text{ in } Q \text{ only in output position}\}$$

is a barbed bisimulation up to structural equivalence.

- Let $Q\{p/q\} \xrightarrow{\tau} Q'\{p/q\}$. There are two cases.
  1. $Q \xrightarrow{\tau} Q'$. This case can be easily treated.
  2. Otherwise, since p and q are channels and they never appear in testing, this means that the τ-action is due to a communication along p. More precisely, Q must contain an occurrence of q in output subject position and an occurrence of p in input position which give rise to the communication. Up to structural equivalence, this implies that

     $$(\boldsymbol{\nu} q{:}\mathbf{C}(T))\, (Q \mid q \triangleright p) \xrightarrow{\tau} \xrightarrow{\tau} \equiv (\boldsymbol{\nu} q{:}\mathbf{C}(T))\, (Q' \mid q \triangleright p).$$

     As desired.
- Let $(\boldsymbol{\nu} q{:}\mathbf{C}(T))\, (Q \mid q \triangleright p) \xrightarrow{\tau} R$ for some R. There are two cases.
  1. $R = (\boldsymbol{\nu} q{:}\mathbf{C}(T))\, (Q' \mid q \triangleright p)$ since $Q \xrightarrow{\tau} Q'$. This case can be easily treated.
  2. The τ-action is due to some communication along q between Q and the link $q \triangleright p$. More precisely,

     $$(\boldsymbol{\nu} q{:}\mathbf{C}(T))\, (Q \mid q \triangleright p) \equiv (\boldsymbol{\nu} q{:}\mathbf{C}(T))\, ((\boldsymbol{\nu} \tilde{z})\, (Q' \mid \overline{q}v) \mid q \triangleright p)$$

     and

     $$(\boldsymbol{\nu} q{:}\mathbf{C}(T))\, (Q \mid q \triangleright p) \xrightarrow{\tau} \equiv (\boldsymbol{\nu} q{:}\mathbf{C}(T))\, ((\boldsymbol{\nu} \tilde{z})\, (Q' \mid \overline{p}v) \mid q \triangleright p).$$

     The left side can easily mimic the move as follows:

     $$Q'\{p/q\} \Longrightarrow \equiv (\boldsymbol{\nu} \tilde{z})\, (Q' \mid \overline{q}v)\{p/q\} = (\boldsymbol{\nu} \tilde{z})\, (Q' \mid \overline{p}v)\{p/q\}.$$

     As desired.

□

A.2 Proof of Theorem 6.1

To prove Theorem 6.1 we need the following lemma, allowing us to type object/alias managers using the translation of an object type.

Lemma A.1 *If* $A = [\mathrm{l}_j{:}\widetilde{B}_j{\to}\widehat{B}_j]_{j\in 1..n}$ *and* $\Gamma \vdash \mathbb{O}{:}A$.

$$\begin{aligned}\Gamma = \ & s{:}[\![A]\!], t_1{:}\mathbf{C}([\![A]\!], \mathbf{M}(\widetilde{B}_1{\to}\widehat{B}_1), \mathbf{K}) \dots \\ & t_n{:}\mathbf{C}([\![A]\!], \mathbf{M}(\widetilde{B}_n{\to}\widehat{B}_n), \mathbf{K}), m_{\mathrm{e}}{:}\mathbf{C}(), m_{\mathrm{i}}{:}\mathbf{C}(\mathbf{K}), k_{\mathrm{e}}{:}\mathbf{K}, k_{\mathrm{i}}{:}\mathbf{K}\end{aligned}$$

and

$$\Gamma' = s{:}[\![A]\!], s_{\mathrm{a}}{:}[\![A]\!], m_{\mathrm{e}}{:}\mathbf{C}(), m_{\mathrm{i}}{:}\mathbf{C}(\mathbf{K}), k_{\mathrm{e}}{:}\mathbf{K}, k_{\mathrm{i}}{:}\mathbf{K}$$

then $\Gamma \vdash \mathsf{OM}_{\mathbb{O}}\langle\, s, m_{\mathrm{e}}, m_{\mathrm{i}}, k_{\mathrm{e}}, k_{\mathrm{i}}, t_1 \dots t_n \,\rangle$ *and* $\Gamma' \vdash \mathsf{AM}_{\mathbb{O}}\langle\, s, m_{\mathrm{e}}, m_{\mathrm{i}}, k_{\mathrm{e}}, k_{\mathrm{i}}, s_{\mathrm{a}} \,\rangle$.

PROOF. The proof is in both cases a lengthy type derivation. Here, we only show a part of the derivation of $\Gamma \vdash \mathsf{OM}_{\mathbb{O}}\langle\, s, m_{\mathrm{e}}, m_{\mathrm{i}}, k_{\mathrm{e}}, k_{\mathrm{i}}, t_1 \dots t_n \,\rangle$.

Before we start, let

$$A^*(X) \text{ denote } \left[\begin{array}{lcc} \mathsf{cln} & : & \mathbf{R}(X) \\ \mathsf{ali} & : & \langle X, \mathbf{R}(X) \rangle \\ \mathsf{upd}_j & : & \langle \mathbf{C}(X, \mathbf{M}(\widetilde{B}_j \to \widehat{B}_j), \mathbf{K}), \mathbf{R}(X) \rangle \\ \mathsf{inv}_j & : & \langle \mathbf{M}(\widetilde{B}_j \to \widehat{B}_j) \rangle \\ \mathsf{sur} & : & \mathbf{R}(X) \\ \mathsf{png} & : & \mathbf{R}(X) \end{array}\right]_{j \in 1..n},$$

with this abbreviation $[\![A]\!] = \mu X.\, \mathbf{C}(A^*(X), \mathbf{K})$.

The definition of $\mathsf{OM}_{\mathbb{O}}\langle s, m_\mathrm{e}, m_\mathrm{i}, k_\mathrm{e}, k_\mathrm{i}, t_1 \ldots t_n \rangle$ is

$$s(l,k).(\boldsymbol{\nu} k^*{:}\mathbf{K}) \Big(\mathsf{if}\ [k{=}k_\mathrm{i}]\ \mathsf{then}\ P_1\ \mathsf{elif}\ [k{=}k_\mathrm{e}]\ \mathsf{then}\ P_2\ \mathsf{else}\ P_3\Big)$$

and we are to check that this process is well-typed under Γ with the extra assumption that $\mathsf{OM}_{\mathbb{O}}\langle s, m_\mathrm{e}, m_\mathrm{i}, k_\mathrm{e}, k_\mathrm{i}, t_1 \ldots t_n \rangle$ is well-typed under Γ.

By rule (T-INP) we must establish that s has a channel type. In Γ we have the assumption $s{:}[\![A]\!]$, and using (T-REC2) we can unfold $[\![A]\!]$, obtaining $\mathbf{C}(A^*([\![A]\!]), \mathbf{K})$. This yields as new subgoal (using (T-RES) to handle the restriction), that we must prove:

$$\Gamma' \vdash \mathsf{if}\ [k{=}k_\mathrm{i}]\ \mathsf{then}\ P_1\ \mathsf{elif}\ [k{=}k_\mathrm{e}]\ \mathsf{then}\ P_2\ \mathsf{else}\ P_3$$

with $\Gamma' = \Gamma, l{:}A^*([\![A]\!]), k{:}\mathbf{K}, k^*{:}\mathbf{K}$. Checking that all of k, k_i and k_e has type $\mathbf{K}$ as required by (T-IF) is easily done by a lookup in Γ'. And we must now prove that processes P_1, P_2 and P_3 are well-typed under Γ'. We restrict ourselves to consider only P_1. P_1 is a large case construct

$$\begin{array}{rcl} \mathsf{case}\ l\ \mathsf{of}\ \mathsf{cln}\_(r) & : & \mathsf{OM}_{\mathbb{O}}\langle s, m_\mathrm{e}, m_\mathrm{i}, k_\mathrm{e}, k^*, \widetilde{t} \rangle \mid (\boldsymbol{\nu} s^*)\,(\,\overline{r}\langle s^*, k^* \rangle \mid \mathsf{newO}_{\mathbb{O}}\langle s^*, \widetilde{t} \rangle\,)\ ; \\ \mathsf{ali}\_(s_\mathrm{a}, r) & : & \mathsf{AM}_{\mathbb{O}}\langle s, m_\mathrm{e}, m_\mathrm{i}, k_\mathrm{e}, k^*, s_\mathrm{a} \rangle \mid \overline{r}\langle s_\mathrm{a}, k^* \rangle\ ; \\ \mathsf{upd}_j\_(t', r) & : & \mathsf{OM}_{\mathbb{O}}\langle s, m_\mathrm{e}, m_\mathrm{i}, k_\mathrm{e}, k^*, t_1 \mathinner{..} t_{j-1}, t', t_{j+1} \mathinner{..} t_n \rangle \mid \overline{r}\langle s, k^* \rangle\ ; \\ \mathsf{inv}_j\_(\widetilde{x}, r) & : & \mathsf{OM}_{\mathbb{O}}\langle s, m_\mathrm{e}, m_\mathrm{i}, k_\mathrm{e}, k^*, \widetilde{t} \rangle \mid \overline{t_j}\langle s, \widetilde{x}, r, k^* \rangle\ ; \\ \mathsf{sur}\_(r) & : & \mathsf{OM}_{\mathbb{O}}\langle s, m_\mathrm{e}, m_\mathrm{i}, k_\mathrm{e}, k^*, \widetilde{t} \rangle \mid [\![s.\mathsf{alias}\langle s.\mathsf{clone}\rangle]\!]_r^{k^*}\ ; \\ \mathsf{png}\_(r) & : & \mathsf{OM}_{\mathbb{O}}\langle s, m_\mathrm{e}, m_\mathrm{i}, k_\mathrm{e}, k^*, \widetilde{t} \rangle \mid [\![s]\!]_r^{k^*} \end{array}$$

with $j \in 1..n$. By inspection we see that the case construct has the labels required by $A^*(X)$. And we must now type the continuations. We only show how the continuation for label $\mathsf{inv}_j\_(\widetilde{x}, r)$ is handled. Let $\Gamma'' = \Gamma', \widetilde{x}{:}[\![\widetilde{B}_j]\!], r{:}\mathbf{R}(\widehat{B}_j)$. We shall now establish

$$\Gamma'' \vdash \mathsf{OM}_{\mathbb{O}}\langle s, m_\mathrm{e}, m_\mathrm{i}, k_\mathrm{e}, k^*, \widetilde{t} \rangle \mid \overline{t_j}\langle s, \widetilde{x}, r, k^* \rangle$$

By narrowing Γ'' and our initial assumption, we get that

$$\Gamma'' \vdash \mathsf{OM}_{\mathbb{O}}\langle s, m_\mathrm{e}, m_\mathrm{i}, k_\mathrm{e}, k^*, \widetilde{t} \rangle$$

and by lookup in Γ'' we get that

$$\Gamma'' \vdash t_j{:}\mathbf{C}([\![A]\!], [\![\widetilde{B}_j]\!], \mathbf{R}(\widehat{B}_j), \mathbf{K}), s{:}[\![A]\!], \widetilde{x}{:}[\![\widetilde{B}_j]\!], r{:}\mathbf{R}(\widehat{B}_j), k^*{:}\mathbf{K}$$

□

PROOF OF THEOREM 6.1. The implication from left to right is proved using induction in the depth of the derivation of $\Gamma \vdash a{:}A$ with a case analysis of the last rule used. We show a few of the cases below.

(T-VAR) Assume $\Gamma \vdash x{:}A$, by rule (T-VAR) we have $\Gamma(x) = A$. The translation of x is $\overline{p}\langle x, k\rangle$ and $[\![\Gamma]\!](x) = [\![A]\!]$. Let $\Gamma' = [\![\Gamma]\!], p{:}\mathbf{R}([\![A]\!])\,, k{:}\mathbf{K}$ We can now complete the derivation:

$$\frac{\Gamma' \vdash p{:}\mathbf{C}([\![A]\!], \mathbf{K}),\ x{:}[\![A]\!],\ k{:}\mathbf{K}}{\Gamma' \vdash \overline{p}\langle x, k\rangle}$$

(T-OBJ) Assume $\Gamma \vdash [l_j{=}\varsigma(s_j{:}A, \tilde{x}_j{:}\widetilde{B}_j)b_j]_{j\in J}{:}A$ with $A = [l_j{:}\widetilde{B}_j{\to}\widehat{B}_j]_{j\in J}$. By induction

$$[\![\, \Gamma, s_j{:}A, \tilde{x}_j{:}\widetilde{B}_j \,]\!], r{:}\mathbf{R}([\![\, \widehat{B}_j \,]\!]), k'{:}\mathbf{K} \vdash [\![\, b_j \,]\!]_r^{k'}$$

for all $j \in J$.
The translation of $[l_j{=}\varsigma(s_j{:}A, \tilde{x}_j{:}\widetilde{B}_j)b_j]_{j\in J}$ is

$$(\boldsymbol{\nu} s{:}[\![\, A \,]\!], t_j{:}T_j)_{\,j\in J} \Big(\overline{p}\langle s, k\rangle \mid \mathsf{newO}_{\mathbb{O}}\langle\, s, \widetilde{t}\,\rangle \mid \prod_{j\in J} {!}\, t_j(s_j, \widetilde{x}_j, r, k').[\![\, b_j \,]\!]_r^{k'} \Big)$$

where $T_j = \mathbf{C}([\![\, A \,]\!], [\![\, \widetilde{B}_j \,]\!], \mathbf{R}(\widehat{B}_j), \mathbf{K})$ and $\widetilde{t} = t_j \; j \in J$. Let

$$\Gamma' = [\![\, \Gamma \,]\!], p{:}\mathbf{R}([\![\, A \,]\!]), k{:}\mathbf{K}, s{:}[\![\, A \,]\!], T_j \;{}_{j\in J}.$$

We now got three subgoals. Proving that $\Gamma' \vdash \overline{p}\langle s, k\rangle$ follows easily from a lookup in Γ'. That $\Gamma' \vdash \mathsf{newO}_{\mathbb{O}}\langle\, s, \widetilde{t}\,\rangle$, follows from applications of (T-RES), (T-PAR), narrowing and Lemma A.1. Finally, to establish $\Gamma' \vdash \; {!}\, t_j(s_j, \widetilde{x}_j, r, k').[\![\, b_j \,]\!]_r^{k'}$ we apply (T-REP), (T-INP) and the induction hypothesis.

(T-FORK) Assume $\Gamma \vdash \mathsf{fork}\langle a\rangle{:}\, \mathrm{Thr}(A)$. The translation of $\mathsf{fork}\langle a\rangle$ is

$$(\boldsymbol{\nu} q{:}\mathbf{R}([\![\, A \,]\!]), t{:}[\![\, \mathrm{Thr}(A) \,]\!], k^*{:}\mathbf{K}) \left(\, [\![\, a \,]\!]_q^{k^*} \;\mid\; \overline{p}\langle t, k\rangle \;\mid\; q(x, k').t(r, k'').\overline{r}\langle x, k''\rangle \, \right)$$

Let $\Gamma' = [\![\, \Gamma \,]\!], p{:}\mathbf{R}([\![\, \mathrm{Thr}(A) \,]\!]), k{:}\mathbf{K}, q{:}\mathbf{R}([\![\, A \,]\!]), t{:}[\![\, \mathrm{Thr}(A) \,]\!]$. We now got three subgoals. $\Gamma' \vdash [\![\, a \,]\!]_q^{k'}$ follows using narrowing and the induction hypothesis. $\Gamma \vdash \overline{p}\langle t, k^*\rangle$ follows using (T-OUT). Finally, the following derivation

$$\dfrac{\Gamma' \vdash q{:}\mathbf{C}([\![\, A \,]\!], \mathbf{K}) \qquad \dfrac{\begin{array}{c}\Gamma', x{:}[\![\, A \,]\!], k'{:}\mathbf{K} \vdash t{:}\mathbf{C}(\mathbf{R}([\![\, A \,]\!]), \mathbf{K}) \\ \Gamma', x{:}[\![\, A \,]\!], k'{:}\mathbf{K}, r{:}\mathbf{R}([\![\, A \,]\!]), k''{:}\mathbf{K} \vdash r{:}\mathbf{C}([\![\, A \,]\!], \mathbf{K}), x{:}[\![\, A \,]\!], k''{:}\mathbf{K}\end{array}}{\Gamma', x{:}[\![\, A \,]\!], k'{:}\mathbf{K} \vdash t(r, k'').\overline{r}\langle x, k''\rangle}}{\Gamma' \vdash q(x, k').t(r, k'').\overline{r}\langle x, k''\rangle}$$

proves the last subgoal.

(T-CLO) Assume $\Gamma \vdash a.\mathsf{clone}{:}A$ with $A = [l_j{:}A_j]_{j\in J}$. The translation of $a.\mathsf{clone}$ is

$$(\boldsymbol{\nu} q{:}\mathbf{R}([\![\, A \,]\!]))\, ([\![\, a \,]\!]_q^k \,|\, q(y, k')\,.\,\overline{y}\langle \mathsf{cln}\_p, k'\rangle).$$

Let $\Gamma' = \Gamma, p{:}\mathbf{R}([\![\, \mathrm{Thr}(A) \,]\!]), k{:}\mathbf{K}, q{:}\mathbf{R}([\![\, \mathrm{Thr}(A) \,]\!])$. We have two subgoals. $\Gamma' \vdash [\![\, a \,]\!]_q^k$ follows from narrowing and the induction hypothesis. For the second subgoal, application of (T-INP) yields that we must establish $\Gamma', y{:}[\![\, A \,]\!], k'{:}\mathbf{K} \vdash \overline{y}\langle \mathsf{cln}\_p, k'\rangle$, which is handled using (T-REC2) to unfold the translation of the object type $[l_j{:}A_j]_{j\in J}$, (T-VAR) to check that the unfolded type has the required variant tag, and finally (T-BAS) to check that p has type $[\![\, A \,]\!]$.

The implication from right to left is proved by induction in the structure of a. Again we again only show a few of the cases.

x: Assume $[\![\, \Gamma \,]\!], p{:}\mathbf{R}([\![\, A \,]\!]), k{:}\mathbf{K} \vdash \overline{p}\langle x, k\rangle$. This typing must have been derived using (T-OUT) with premise $\Gamma' \vdash p{:}\mathbf{C}([\![\, A \,]\!], \mathbf{K})$, $x{:}[\![\, A \,]\!]$, $k{:}\mathbf{K}$. This can only be true if $x \in \mathrm{dom}(\Gamma)$ with $\Gamma(x) = A$. We can now apply (T-VAR) to derive $\Gamma \vdash x{:}A$.

$[l_j{=}\varsigma(s_j{:}A, \tilde{x}_j{:}\widetilde{B}_j)b_j]_{j\in J}$: Assume $[\![\, \Gamma \,]\!], p{:}\mathbf{R}([\![\, A \,]\!]), k{:}\mathbf{K} \vdash [\![\, [l_j{=}\varsigma(s_j{:}A, \tilde{x}_j{:}\widetilde{B}_j)b_j]_{j\in J} \,]\!]$. The type A can either be an object type $[l_k{=}\varsigma(s_k{:}A, \tilde{x}_k{:}\widetilde{B}_k)b_k]_{k\in K}$ or a thread type $\mathrm{Thr}(B)$. The translation of $[l_j{=}\varsigma(s_j{:}A, \tilde{x}_j{:}\widetilde{B}_j)b_j]_{j\in J}$ is

$$(\boldsymbol{\nu} s{:}[\![\, A \,]\!], t_j{:}T_j)_{\,j\in J} \Big(\overline{p}\langle s, k\rangle \mid \mathsf{newO}_{\mathbb{O}}\langle\, s, \widetilde{t}\,\rangle \mid \prod_{j\in J} {!}\, t_j(s_j, \widetilde{x}_j, r, k').[\![\, b_j \,]\!]_r^{k'} \Big)$$

We can easily rule out the possibility that $A = \mathrm{Thr}(B)$ because if A was a thread type, we would not be able to type the object manager. Therefore $A = [l_k{=}\varsigma(s_k{:}A, \tilde{x}_k{:}\widetilde{B}_k)b_k]_{k\in K}$, and in order to type the object manager we must also have $K = J$ in order to have the same number of methods in the type and the object manger. The typing of the object manger also yields that we must have the types $T_j = \mathbf{C}([\![A]\!], [\![\widetilde{B}_j]\!], \mathbf{R}(\widehat{B}_j), \mathbf{K})$. We are now able to write a typing for $!\, t_j(s_j, \widetilde{x}_j, r, k').[\![b_j]\!]_r^{k'}$, which as premise has

$$\Gamma', s{:}[\![A]\!], t_j{:}T_j, s_j{:}[\![A]\!], k'{:}\mathbf{K}, r_j{:}\mathbf{R}([\![\widehat{B}_j]\!]), \widetilde{x}_j{:}[\![\widetilde{B}_j]\!] \vdash [\![b_j]\!]_r^{k'}.$$

Using narrowing and the induction hypothesis we derive that $\Gamma, s_j{:}A, \widetilde{x}_j \vdash b_j{:}\widehat{B}_j$. And we can now apply (T-Obj) to conclude $\Gamma \vdash [\mathrm{l}_j{=}\varsigma(s_j{:}A, \tilde{x}_j{:}\widetilde{B}_j)b_j]_{j\in J}{:}A$.

a.clone: Assume $[\![\Gamma]\!], p{:}\mathbf{R}([\![A]\!]), k{:}\mathbf{K} \vdash a.\mathsf{clone}$. The type A can either be an object type $[\mathrm{l}_j{:}\widetilde{B}_j{\to}\widehat{B}_j]_{j\in J}$ or a thread type $\mathrm{Thr}(B)$. The translation of a.clone is

$$(\boldsymbol{\nu} q{:}T)\,([\![a]\!]_q^k \mid q(y, k')\,.\,\overline{y}\langle \mathsf{cln\_}p, k'\rangle)$$

for some type annotation T. By the use of the name q we can conclude that $T = \mathbf{R}([\![A]\!])$ and that A cannot be a thread type (because of the cln_p request). Knowing that q has type $\mathbf{R}([\![A]\!])$ allows us to use the induction hypothesis (together with narrowing) to conclude that $\Gamma \vdash a{:}A$, and then we can apply (T-Cln) to get $\Gamma \vdash a.\mathsf{clone}{:}A$.

□

A.3 Proof of Lemma 6.5

Proof. As the base case, we consider Z, where the object manager at s has just been created; all previous steps in the sequence are obviously irrelevant, because the condition of containing $\mathsf{newO}_{\mathbb{O}}\langle s, \widetilde{t}\rangle$ is not fulfilled. Then

$$Z = C'[\, \mathsf{newO}_{\mathbb{O}}\langle s, \widetilde{t}\rangle \,] = C'[\, (\boldsymbol{\nu}\widetilde{n}k_{\mathrm{i}})\,(\, \overline{m_{\mathrm{e}}} \mid \mathsf{OM}_{\mathbb{O}}\langle s, \widetilde{n}, k_{\mathrm{i}}, \widetilde{t}\rangle\,)\,]$$

Using structural equivalence, we immediately get

$$Z \equiv E[\, (\boldsymbol{\nu}\widetilde{n})\,(\, \overline{m_{\mathrm{e}}} \mid \mathsf{OM}_{\mathbb{O}}\langle s, \widetilde{n}, k_{\mathrm{i}}, \widetilde{t}\rangle \mid \mathsf{PP}_{\mathbb{O}}\langle s, \widetilde{n}, \emptyset\rangle\,)\,]$$

for some static context $E[\cdot]$, such that Z corresponds to state OM^{f}. It is important to notice that names in $\widetilde{n}$ will only appear inside the object manager and the pre-processed requests.

State OM^{f} can only evolve into some state OM^{a}; it does so by grabbing the external mutex m_{e} for one of its pre-processed requests in $\widetilde{v}$. The only other possible reduction involving state OM^{f} is pre-processing another request, but such an action does not change the state—it only adds to the set of pre-processed requests $\widetilde{v}$. A similar reasoning applies to the other states, so we simply skip pre-processing.

Thus, by consuming the pre-processed request $\overline{s}\langle l, k\rangle$ and leaving untouched the other pre-processed requests $\widetilde{v}$, we may arrive at some Z of the form:

$$\begin{aligned} & E[\, (\boldsymbol{\nu}\widetilde{n})\,(\, \overline{m_{\mathrm{e}}} \mid \mathsf{OM}_{\mathbb{O}}\langle s, \widetilde{n}, k_{\mathrm{i}}, \widetilde{t}\rangle \mid \mathsf{PP}_{\mathbb{O}}\langle s, \widetilde{n}, \widetilde{v}\rangle\,)\,] \\ \to_{\equiv}\ & E[\, (\boldsymbol{\nu}\widetilde{n})\,(\, \underline{\overline{m_{\mathrm{i}}}k \mid \overline{s}\langle l, k_{\mathrm{e}}\rangle} \mid \mathsf{OM}_{\mathbb{O}}\langle s, \widetilde{n}, k_{\mathrm{i}}, \widetilde{t}\rangle \mid \mathsf{PP}_{\mathbb{O}}\langle s, \widetilde{n}, \widetilde{v}{-}\langle l, k\rangle\rangle\,)\,] \\ \stackrel{\mathrm{def}}{=}\ & Z \end{aligned}$$

where Z corresponds to state OM^{a}.

State OM^{a} can only evolve into either state OM^{n} or OM^{s}, by consuming the request $\overline{s}\langle l, k_{\mathrm{e}}\rangle$:

- State OM^{a} evolves into state OM^{n} if l is one of $\mathsf{ali\_}\langle x, p\rangle$, $\mathsf{cln\_}p$, or $\mathsf{upd}_j\mathsf{\_}\langle t, p\rangle$, which are disallowed as external request, the object manager is restarted and, up to structural equivalence, we get state OM^{n}.
- In the remaining cases, that is, when l is one of $\mathsf{inv}_j\mathsf{\_}\langle x, p\rangle$, $\mathsf{sur\_}p$, or $\mathsf{sur\_}\langle p\rangle$, state OM^{a} evolves into state OM^{s}. Indeed, a call-manager is started concurrently with the restarted object manager. By using structural equivalence, we can move components that are not in the scope of $\widetilde{n}$ outside this scope, so as to recognize state OM^{s}.

In state OM^{s}, a png request drives the system into state OM^{i}. In the case of method invocation a reduction along t_j may occur which allows the evaluation of the method body. At this point a number of self-inflicted requests may be served (external requests are blocked because the external mutex m_{e} is no available). This part of the computation will not change the state. Notice that, by hypothesis, since we suppose that Z contain an object manager and non an alias manager, we exclude self-inflicted aliasing operations. When the last self-inflicted request is served, a reply $\overline{r^*}\langle\, o, k\,\rangle$ will appear unguarded. The confluent reduction along r^* will drive the computation to state OM^{i}. sur requests are treated similarly.

State OM^{i} can only evolve, by reducing along m_{i}, to state OM^{f}. □

A.4 Proof of Lemma 8.2

We show that there is a sequence of τ-actions such that:

$$\begin{array}{l}\mathsf{surO}_{\mathbb{O}}\langle\, s, r, k, \tilde{t}, \tilde{v}\,\rangle \Rightarrow_{\equiv} \\ (\boldsymbol{\nu} s^*)\,\big((\boldsymbol{\nu} k_{\mathrm{i}})\,\mathsf{freeA}_{\mathbb{O}}\langle\, s, k_{\mathrm{i}}, s^*, \tilde{v}\,\rangle \,\big|\, \mathsf{newO}_{\mathbb{O}}\langle\, s^*, \tilde{t}\,\rangle \,\big|\, \overline{r}\langle s^*, k\rangle\big).\end{array}$$

We prove that $\approx_{\Gamma;s}$ is insensitive to these particular τ-actions. To this end, we supply the two lemmas A.2 and A.3. We recall that $\mathsf{CM}[\cdot]$ denote the call manager protocol as defined in Table 7.

Lemma A.2 *Let $\widetilde{n} := m_{\mathrm{e}}, m_{\mathrm{i}}, k_{\mathrm{e}}$, and $\widetilde{v} := v_1 \,..\, v_n$ with $v_j := \langle\, l_j, k_j\,\rangle$ for $j \in 1.n$, and*

$$\begin{array}{rcl}
C_1 & := & \mathsf{CM}[\,(\boldsymbol{\nu} q)\,(\overline{s}\langle \mathsf{cln_}q, k^*\rangle \mid q(x, k').\overline{s}\langle \mathsf{ali_}\langle x, r^*\rangle, k'\rangle)\,] \\
C_2 & := & \mathsf{CM}[\,(\boldsymbol{\nu} q)\,(\overline{q}\langle s^*, k^*\rangle \mid q(x, k').\overline{s}\langle \mathsf{ali_}\langle x, r^*\rangle, k'\rangle)\,] \\
P\langle\widetilde{v}\rangle & := & (\boldsymbol{\nu}\widetilde{n}k^*)\,\big(\,\overline{m_{\mathrm{i}}}k \big|\, \mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{n}, k^*, \widetilde{t}\,\rangle \,\big|\, \mathsf{PP}_{\mathbb{O}}\langle\, s, \widetilde{n}, \widetilde{v}\,\rangle \,\big|\, C_1\,\big) \\
 & & \text{with } k^* \notin \mathrm{fn}(\widetilde{v}) \\
Q\langle\widetilde{v}\rangle & := & (\boldsymbol{\nu}\widetilde{n}k^* s^*)\,\big(\,\overline{m_{\mathrm{i}}}k \big|\, \mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{n}, k^*, \widetilde{t}\,\rangle \,\big|\, \mathsf{newO}_{\mathbb{O}}\langle\, s^*, \widetilde{t}\,\rangle \,\big|\, \mathsf{PP}_{\mathbb{O}}\langle\, s, \widetilde{n}, \widetilde{v}\,\rangle \,\big|\, C_2\,\big) \\
 & & \text{with } k^* \notin \mathrm{fn}(\widetilde{v}) \\
\Gamma & \vdash & P\langle\widetilde{v}\rangle, Q\langle\widetilde{v}\rangle \;\text{ for some } \Gamma.
\end{array}$$

Then, $P\langle\widetilde{v}\rangle \approx_{\Gamma;s} Q\langle\widetilde{v}\rangle$.

Proof. For simplicity, we omit the obligations on types in the coinductive definition of $\approx_{\Gamma;s}$. So, we prove that the relation:

$$\mathcal{S} = \{(P\langle\widetilde{w}\rangle, Q\langle\widetilde{w}\rangle) : \widetilde{w} = w_1 \,..\, w_m \text{ with } w_j := \langle\, l_j, k_j\,\rangle,\ j \in 1..n\} \cup \mathcal{I}$$

where $\mathcal{I}$ is the identity relation, is a $\approx_{\Gamma;s}$-bisimulation up to $\equiv$.

The only channel which appear free in subject position in $P\langle\widetilde{w}\rangle$ and $Q\langle\widetilde{w}\rangle$ is s. Since both the external key k_{e} and the internal key k^* are restricted in $P\langle\widetilde{w}\rangle$ and $Q\langle\widetilde{w}\rangle$, an by well-typedness, the environment can send requests only of the form $\overline{s}\langle l, k\rangle$ with $k_{\mathrm{e}} \neq k \neq k^*$.

The process $P\langle\widetilde{w}\rangle$ can perform only two kinds of actions. Either (i) an input action $s\langle l, k\rangle$ (with $k_{\mathrm{e}} \neq k \neq k^*$), or (ii) a silent move along s involving the self-inflicted cloning request contained in C_1. In case (i), the pre-processing of the request creates the process $m_{\mathrm{e}}.(\overline{s}\langle l, k_{\mathrm{e}}\rangle \mid \overline{m_{\mathrm{i}}}k)$ which can be added in $\mathsf{PP}_{\mathbb{O}}\langle\, s, \widetilde{n}, \widetilde{w}\,\rangle$ obtaining some $\mathsf{PP}_{\mathbb{O}}\langle\, s, \widetilde{n}, \widetilde{w'}\,\rangle$ with $\widetilde{w'} = \widetilde{w} \cup \langle l, k\rangle$. The process $Q\langle\widetilde{w}\rangle$ can perform the same action and the derivatives are again related by $\mathcal{S}$. In case (ii), the process $Q\langle\widetilde{w}\rangle$ can mimic the τ-action by not performing any reduction at all. Up to structural equivalence, we get into the identity relation.

The process $Q\langle\widetilde{w}\rangle$ can only perform two kinds of actions. Either (i) a input action $s\langle l, k\rangle$ (with $k_{\mathrm{e}} \neq k \neq k^*$), and we reason as above, or (ii) a silent move along the restricted channel q in C_2. In this case $P\langle\widetilde{w}\rangle$ can perform two silent actions, along s and q, getting, up to structural equivalence, into the identity relation. □

Lemma A.3 *Let $\widetilde{n} := m_{\mathrm{e}}, m_{\mathrm{i}}, k_{\mathrm{e}}$, and $\widetilde{v} := v_1 \,..\, v_n$ with $v_j := \langle\, l_j, k_j\,\rangle$ for $j \in 1..n$, and*

$$
\begin{aligned}
C_3 &:= \mathsf{CM}[\,\overline{s}\langle \mathsf{ali\_}\langle s^*, r^*\rangle, k^*\rangle\,] \\
C_4 &:= \mathsf{CM}[\,\overline{r^*}\langle s^*, k^*\rangle\,] \\
P\langle\widetilde{v}\rangle &:= (\boldsymbol{\nu}\widetilde{n}k^*s^*)\big(\,\overline{m_{\mathrm{i}}}k \,\big|\, \mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{n}, k^*, \widetilde{t}\,\rangle \,\big|\, \mathsf{newO}_{\mathbb{O}}\langle\, s^*, \widetilde{t}\,\rangle \,\big|\, \mathsf{PP}_{\mathbb{O}}\langle\, s, \widetilde{n}, \widetilde{v}\,\rangle \,\big|\, C_3\,\big) \\
&\ \text{with } k^* \notin \mathrm{fn}(\widetilde{v}) \\
Q\langle\widetilde{v}\rangle &:= (\boldsymbol{\nu}\widetilde{n}k^*s^*)\big(\,\overline{m_{\mathrm{i}}}k \,\big|\, \mathsf{AM}_{\mathbb{O}}\langle\, s, \widetilde{n}, k^*, s^*\,\rangle \,\big|\, \mathsf{newO}_{\mathbb{O}}\langle\, s^*, \widetilde{t}\,\rangle \,\big|\, \mathsf{PP}_{\mathbb{O}}\langle\, s, \widetilde{n}, \widetilde{v}\,\rangle \,\big|\, C_4\,\big) \\
&\ \text{with } k^* \notin \mathrm{fn}(\widetilde{v}). \\
\Gamma &\vdash P\langle\widetilde{v}\rangle, Q\langle\widetilde{v}\rangle \ \text{ for some } \Gamma.
\end{aligned}
$$

Then, $P\langle\widetilde{v}\rangle \approx_{\Gamma;s} Q\langle\widetilde{v}\rangle$.

Proof. Similar to that of Lemma A.2. □

Proof of Lemma 8.2. As said above there is a sequence of τ-actions, such that:

$$\mathsf{surO}_{\mathbb{O}}\langle\, s, r, k, \tilde{t}, \tilde{v}\,\rangle \Rightarrow_{\equiv} (\boldsymbol{\nu}s^*)\,\big((\boldsymbol{\nu}k_{\mathrm{i}})\,\mathsf{freeA}_{\mathbb{O}}\langle\, s, k_{\mathrm{i}}, s^*, \tilde{v}\,\rangle \,\big|\, \mathsf{newO}_{\mathbb{O}}\langle\, s^*, \tilde{t}\,\rangle \,\big|\, \overline{r}\langle s^*, k\rangle\big).$$

The above sequence consists of 7 silent steps. These τ-steps are of two kinds: (i) confluent reductions along restricted channels of the form

$$C[(\boldsymbol{\nu}q)\,(\overline{q}\langle\widetilde{v}\rangle \mid q(\widetilde{x}).P)] \xrightarrow{\tau}_{\equiv} C[P\{^{\widetilde{v}}/_{\widetilde{x}}\}]$$

where $q \notin \mathrm{fn}(P)$, let us call these reductions of kind α; (ii) reductions involving self-inflicted requests (induced by the surrogation) of the form

$$C[(\boldsymbol{\nu}k^*)\,(\mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k^*, \tilde{t}\,\rangle \mid \overline{s}\langle \mathsf{op\_}r^*, k^*\rangle)] \xrightarrow{\tau} \ldots$$

let us call these reductions of kind β. It is well-known that $\approx_{\Gamma}$ (as well as $\approx_{\Gamma;s}$) is insensitive to reductions of kind α. In Lemma A.2 and A.3 we show that $\approx_{\Gamma;s}$ is insensitive to the reductions of kind β appearing in the sequence mentioned above. This is possible because, in the implementation of object and alias managers, we use nonces (c.f. page 21) in order to guarantee that the self-inflicted key of the object manager is always restricted. In this manner, the environment cannot produce any "malicious" self-inflicted request which might potentially interfere with the cloning and the aliasing requests.

The first and the second reductions are of kind α and they are due to the process $\mathsf{CM}[\,[\![\, s.\mathsf{alias}\langle s.\mathsf{clone}\rangle\,]\!]^{k^*}_{r^*}\,]$ (contained in $\mathsf{surO}_{\mathbb{O}}\langle\, s, r, k, \tilde{t}, \tilde{v}\,\rangle$) which, after two τ-steps, reduces to the process

$$\mathsf{CM}[\,(\boldsymbol{\nu}q)\,(\overline{s}\langle \mathsf{cln\_}q, k^*\rangle \mid q(x,i).\overline{s}\langle \mathsf{ali\_}\langle x, r^*\rangle, i\rangle)\,].$$

We abbreviate this process by C_1. The situation is that:

$$\mathsf{surO}_{\mathbb{O}}\langle\, s, r, k, \tilde{t}, \tilde{v}\,\rangle \xrightarrow{\tau}\xrightarrow{\tau} (\boldsymbol{\nu}\widetilde{n}k^*)\big(\,\overline{m_{\mathrm{i}}}k \,\big|\, \mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{n}, k^*, \widetilde{t}\,\rangle \,\big|\, \mathsf{PP}_{\mathbb{O}}\langle\, s, \widetilde{n}, \widetilde{v}\,\rangle \,\big|\, C_1\,\big)$$

where $k^* \notin \mathrm{fn}(\widetilde{v})$.

The third reduction is of kind β and involves the self-inflicted cloning request in C_1. Let C_2 be the process $\mathsf{CM}[\,(\boldsymbol{\nu}q)\,(\overline{q}\langle s^*, k^*\rangle \mid q(x,i).\overline{s}\langle \mathsf{ali\_}\langle x, r^*\rangle, i\rangle)\,]$, then the process

$$(\boldsymbol{\nu}\widetilde{n}k^*)\big(\,\overline{m_{\mathrm{i}}}k \,\big|\, \mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{n}, k^*, \widetilde{t}\,\rangle \,\big|\, \mathsf{PP}_{\mathbb{O}}\langle\, s, \widetilde{n}, \widetilde{v}\,\rangle \,\big|\, C_1\,\big)$$

reduces, up to structural equivalence, to

$$(\boldsymbol{\nu}\widetilde{n}k^*s^*)\big(\,\overline{m_{\mathrm{i}}}k \,\big|\, \mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{n}, k^*, \widetilde{t}\,\rangle \,\big|\, \mathsf{newO}_{\mathbb{O}}\langle\, s^*, \widetilde{t}\,\rangle \,\big|\, \mathsf{PP}_{\mathbb{O}}\langle\, s, \widetilde{n}, \widetilde{v}\,\rangle \,\big|\, C_2\,\big)$$

where $k^* \notin \mathrm{fn}(\widetilde{v})$. By Lemma A.2 the relation $\approx_{\Gamma;s}$ is insensitive to this reduction.

The fourth reduction is of kind α and it is due to C_2. So, if we denote with C_3 the process $\mathsf{CM}[\,\overline{s}\langle \mathsf{ali\_}s^*, r^*, k^*\rangle\,]$ the situation is that that $\mathsf{surO}_{\mathbb{O}}\langle\, s, r, k, \tilde{t}, \tilde{v}\,\rangle$ evolves in four silent steps, up to structural equivalence, to

$$(\boldsymbol{\nu}\widetilde{n}k^*s^*)\big(\,\overline{m_{\mathrm{i}}}k \,\big|\, \mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{n}, k^*, \widetilde{t}\,\rangle \,\big|\, \mathsf{newO}_{\mathbb{O}}\langle\, s^*, \widetilde{t}\,\rangle \,\big|\, \mathsf{PP}_{\mathbb{O}}\langle\, s, \widetilde{n}, \widetilde{v}\,\rangle \,\big|\, C_3\,\big)$$

where $k^* \notin \mathrm{fn}(\widetilde{v})$.

In the fifth τ-step we reduce the self-inflicted aliasing request contained in C_3. So, let us denote with C_4 the process $\mathsf{CM}[\,\overline{r^*}\langle s^*, k^*\rangle\,]$. It holds that the process

$$(\boldsymbol{\nu}\widetilde{n}k^*s^*)\big(\,\overline{m_{\mathrm{i}}}k \,\big|\, \mathsf{OM}_{\mathbb{O}}\langle\, s,\widetilde{n},k^*,\widetilde{t}\,\rangle \,\big|\, \mathsf{newO}_{\mathbb{O}}\langle\, s^*,\widetilde{t}\,\rangle \,\big|\, \mathsf{PP}_{\mathbb{O}}\langle\, s,\widetilde{n},\widetilde{v}\,\rangle \,\big|\, C_3\,\big)$$

reduces, up to structural equivalence, to

$$(\boldsymbol{\nu}\widetilde{n}k^*s^*)\big(\,\overline{m_{\mathrm{i}}}k \,\big|\, \mathsf{AM}_{\mathbb{O}}\langle\, s,\widetilde{n},k^*,s^*\,\rangle \,\big|\, \mathsf{newO}_{\mathbb{O}}\langle\, s^*,\widetilde{t}\,\rangle \,\big|\, \mathsf{PP}_{\mathbb{O}}\langle\, s,\widetilde{n},\widetilde{v}\,\rangle \,\big|\, C_4\,\big)$$

where $k^* \notin \mathrm{fn}(\widetilde{v})$. By Lemma A.3 the relation $\approx_{\Gamma;s}$ is insensitive to this reduction.

The sixth and the seventh reductions are of kind α and involve channels r^* and m_{i}, respectively. Up to structural equivalence we get the desired process

$$(\boldsymbol{\nu}s^*)\,\big((\boldsymbol{\nu}k_{\mathrm{i}})\,\mathsf{freeA}_{\mathbb{O}}\langle\, s,k_{\mathrm{i}},s^*,\tilde{v}\,\rangle \,\big|\, \mathsf{newO}_{\mathbb{O}}\langle\, s^*,\tilde{t}\,\rangle \,\big|\, \overline{r}\langle s^*,k\rangle\big).$$

□

A.5 Proof of Lemma 8.3

Lemma 8.3 proves that the aliased object manager appearing in Lemma 8.2 behaves as a forwarder.

As a first step we recall a well-known property of replicated input.

Lemma A.4 *Let $C[\cdot]$ be a π-calculus context where channel c does not appear either in input or in output object position. Then*

$$(\boldsymbol{\nu}c)\;(\;!\,c(x).P \;|\; C[\overline{c}v]\;) \approx_\Gamma (\boldsymbol{\nu}c)\;(\;!\,c(x).P \;|\; C[P\{^v/_x\}]\;)$$

Proof. By applying Milner's replications theorems [Mil93]. □

Proof of Lemma 8.3. The obligations on types guarantee that values received along channel s are of the right type. This allows us to use polyadic input along s. By observing process $(\boldsymbol{\nu}k_{\mathrm{i}})\,\mathsf{AM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k_{\mathrm{i}},s^*\,\rangle$ we note that, since k_{i} is restricted and never extruded, the aliased object manager will never receive self-inflicted requests. By exhibiting the appropriate bisimulation, we can prove that such a process has the following functional behaviour.

$$\begin{array}{l}(\boldsymbol{\nu}k_{\mathrm{i}})\,(\mathsf{AM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k_{\mathrm{i}},s^*\,\rangle) \approx_\Gamma \,!\,s(l,k).\mathsf{if}\ [k{=}k_{\mathrm{e}}]\ \ \mathsf{then}\ m_{\mathrm{i}}(k).(\overline{s^*}\langle l,k\rangle \mid \overline{m_{\mathrm{e}}})\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\ \ m_{\mathrm{e}}.(\overline{s}\langle l,k_{\mathrm{e}}\rangle \mid \overline{m_{\mathrm{i}}}k)\end{array}$$

Since $\approx_\Gamma$ is preserved by parallel composition and restriction, we have that:

$$\begin{array}{l}(\boldsymbol{\nu}k_{\mathrm{i}})\,(\mathsf{freeA}_{\mathbb{O}}\langle\, s,k_{\mathrm{i}},s^*,\widetilde{v}\,\rangle)\\ \approx_\Gamma\\ (\boldsymbol{\nu}\widetilde{m}k_{\mathrm{e}})\,\big(\,\overline{m_{\mathrm{e}}} \;\mid\; !\,s(l,k).\mathsf{if}\ [k{=}k_{\mathrm{e}}]\ \ \mathsf{then}\ m_{\mathrm{i}}(k).(\overline{s^*}\langle l,k\rangle \mid \overline{m_{\mathrm{e}}})\\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\ \ m_{\mathrm{e}}.(\overline{s}\langle l,k_{\mathrm{e}}\rangle \mid \overline{m_{\mathrm{i}}}k)\\ \qquad\qquad\quad \mid\; \prod_{j\in 1..n} m_{\mathrm{e}}.\big(\overline{s}\langle l_j,k_{\mathrm{e}}\rangle \mid \overline{m_{\mathrm{i}}}k_j\big)\,\big)\end{array}$$

If we assume that the environment cannot use s in input, then all requests on s are captured by the unique replicated input on s. Moreover, the external identity k_{e} is restricted and never extruded to the environment, and therefore only pre-processed requests "knows" k_{e}. Using these informations, up to harmless confluent reductions along m_{i}, we can safely internalise the management of pre-processed requests by introducing a restricted channel s_{e} with the same type as s and s^*. In this manner we can drop the matching on the identity, and the replicated input on s will only take care of serving external requests. Formally, we have the following.

$$\begin{array}{l}(\boldsymbol{\nu}\widetilde{m}k_{\mathrm{e}})\,\big(\,\overline{m_{\mathrm{e}}}\ \mid\ !\,s(l,k).\mathsf{if}\ [k{=}k_{\mathrm{e}}]\ \ \mathsf{then}\ m_{\mathrm{i}}(k).(\overline{s^*}\langle l,k\rangle \mid \overline{m_{\mathrm{e}}}) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{else}\ \ m_{\mathrm{e}}.(\overline{s}\langle l,k_{\mathrm{e}}\rangle \mid \overline{m_{\mathrm{i}}}k) \\ \qquad\qquad \mid\ \prod_{j\in 1..n} m_{\mathrm{e}}.\big(\overline{s}\langle l_j,k_{\mathrm{e}}\rangle \mid \overline{m_{\mathrm{i}}}k_j\big)\,\big)\end{array}$$

$\approx_{\Gamma;s}$ (by exhibiting the appropriate bisimulation)

$$\begin{array}{l}(\boldsymbol{\nu}\widetilde{m}s_{\mathrm{e}})\,\big(\,\overline{m_{\mathrm{e}}}\ \mid\ !\,s(l,k).m_{\mathrm{e}}.(\overline{s_{\mathrm{e}}}\langle l,k\rangle \mid \overline{m_{\mathrm{i}}}k) \\ \qquad\qquad\quad \mid\ !\,s_{\mathrm{e}}(l,k).m_{\mathrm{i}}(k).(\overline{s^*}\langle l,k\rangle \mid \overline{m_{\mathrm{e}}}) \\ \qquad\qquad\quad \mid\ \prod_{j\in 1..n} m_{\mathrm{e}}.(\overline{s_{\mathrm{e}}}\langle l_j,k_j\rangle \mid \overline{m_{\mathrm{i}}}k_j)\,\big)\end{array}$$

$\approx_{\Gamma}$ (reductions on m_{i} are confluent)

$$\begin{array}{l}(\boldsymbol{\nu}m_{\mathrm{e}}s_{\mathrm{e}})\,\big(\,\overline{m_{\mathrm{e}}}\ \mid\ !\,s(l,k).m_{\mathrm{e}}.\overline{s_{\mathrm{e}}}\langle l,k\rangle \\ \qquad\qquad\quad \mid\ !\,s_{\mathrm{e}}(l,k).(\overline{s^*}\langle l,k\rangle \mid \overline{m_{\mathrm{e}}}) \\ \qquad\qquad\quad \mid\ \prod_{j\in 1..n} m_{\mathrm{e}}.\overline{s_{\mathrm{e}}}\langle l_j,k_j\rangle\,\big)\end{array}$$

$\approx_{\Gamma}$ (by Lemma A.4)

$$\begin{array}{l}(\boldsymbol{\nu}m_{\mathrm{e}}s_{\mathrm{e}})\,\big(\,\overline{m_{\mathrm{e}}}\ \mid\ !\,s(l,k).m_{\mathrm{e}}.\big(\,\overline{s^*}\langle l,k\rangle \mid \overline{m_{\mathrm{e}}}\,\big) \\ \qquad\qquad\quad \mid\ !\,s_{\mathrm{e}}(l,k).(\overline{s^*}\langle l,k\rangle \mid \overline{m_{\mathrm{e}}}) \\ \qquad\qquad\quad \mid\ \prod_{j\in 1..n} m_{\mathrm{e}}.(\overline{s^*}\langle l_j,k_j\rangle \mid \overline{m_{\mathrm{e}}})\,\big)\end{array}$$

$\approx_{\Gamma}$ (by garbage collection on s_{e})

$$\begin{array}{l}(\boldsymbol{\nu}m_{\mathrm{e}})\,\big(\,\overline{m_{\mathrm{e}}}\ \mid\ !\,s(l,k).m_{\mathrm{e}}.\big(\,\overline{s^*}\langle l,k\rangle \mid \overline{m_{\mathrm{e}}}\,\big) \\ \qquad\qquad \mid\ \prod_{j\in 1.n} m_{\mathrm{e}}.\big(\,\overline{s^*}\langle l_j,k_j\rangle \mid \overline{m_{\mathrm{e}}}\,\big)\,\big)\end{array}$$

$\approx_{\Gamma}$ (reductions on m_{e} are confluent)

$$!\,s(l,k).\overline{s^*}\langle l,k\rangle \mid \prod_{j\in 1..n} \overline{s^*}\langle l_j,k_j\rangle$$

$\stackrel{\text{def}}{=}$ (by definition)

$$s \triangleright s^* \mid \prod_{j\in 1..n} \overline{s^*}v_j$$

□

A.6 Proof of Lemma 8.4

This is a rather technical lemma. It is the only place where the theory of Lπ is exploited.

Proof. We apply Lemma 2.15 to process P to remove all the occurrences of s^* in output object position. Let's call $\widehat{P}$ the process obtained by applying Lemma 2.15 in such a way. Note that we focus only on channel s^*. The other channels are not affected by our transformation. Since Lemma 2.15 works with respect to (typed) barbed congruence, it holds that $P \cong_\Gamma \widehat{P}$. This implies

$$(\boldsymbol{\nu}s^*)\,\big(\,s \triangleright s^* \mid P\,\big) \cong_\Gamma (\boldsymbol{\nu}s^*)\,\big(\,s \triangleright s^* \mid \widehat{P}\,\big) \text{ and } P\{{}^{s}\!/_{s^*}\} \cong_\Gamma \widehat{P}\{{}^{s}\!/_{s^*}\}.$$

So, we are left with proving that $(\boldsymbol{\nu}s^*)\,\big(\,s \triangleright s^* \mid \widehat{P}\,\big) \simeq_{\Gamma;s} \widehat{P}\{{}^{s}\!/_{s^*}\}$. The proof follows by showing that the relation:

$$\{(\,(\boldsymbol{\nu}s^*)\,(s \triangleright s^* \mid \widehat{P})\,,\,\widehat{P}\{{}^{s}\!/_{s^*}\}\,) : s \notin \mathrm{fn}(\widehat{P}) \text{ and } s^* \text{ not free in obj. pos. in } \widehat{P}\}$$

is a $\approx_{\Gamma;s}$ bisimilarity. The obligations on types guarantee that values received along channel s are of the right type. A part this, we can safely omit the types in the coinductive definition of $\approx_{\Gamma;s}$.

We recall that $\approx_{\Gamma;s}$ is ground on channels. This means that we alway suppose to receive fresh channels, in particular, we never receive channels s and s^*.

As regards the left side, the only interesting transition is the input action along s. This action can be emulated by the right side by exploiting the asynchronous clause for input.

As regards the right side, we recall that $\approx_{\Gamma;s}$ is not sensitive to output actions along s. Since s^* does not appear free in output object position in $\widehat{P}$, the only interesting action of $\widehat{P}\{^s/_{s^*}\}$ is the input action along s which can be mimicked by the left side up to a τ-action. $\square$

A.7 Proof of Lemma 8.5

We first prove a more general result asserting that pre-processing of external requests is harmless.

Lemma A.5 *Let $\widetilde{v} := v_1 \ldots v_n$ where $v_j := \langle l_j, k_j \rangle$ with $k_\mathrm{e} \neq k_j \neq k_\mathrm{i}$ for $j \in 1..n$. It holds that:*

$$\mathsf{OM}_{\mathbb{O}}\langle s, \widetilde{m}, k_\mathrm{e}, k_\mathrm{i}, \tilde{t}\rangle \mid \prod_{j\in 1..n} \overline{s}v_j \ \approx_{\Gamma;s}\ \mathsf{OM}_{\mathbb{O}}\langle s, \widetilde{m}, k_\mathrm{e}, k_\mathrm{i}, \tilde{t}\rangle \mid \mathsf{PP}_{\mathbb{O}}\langle s, \widetilde{n}, \widetilde{v}\rangle.$$

PROOF. We prove the result by induction on the number of elements of $\widetilde{v}$.

Case $n = 0$. Trivial.

Inductive case. Let

$$\prod_{j\in 1..n} \overline{s}v_j \ \stackrel{\mathrm{def}}{=}\ \overline{s}v_1 \mid \prod_{j\in 2..n} \overline{s}v_j \ \text{and}$$
$$\mathsf{PP}_{\mathbb{O}}\langle s, \widetilde{n}, \widetilde{v}\rangle \ \stackrel{\mathrm{def}}{=}\ m_\mathrm{e}.\big(\overline{s}\langle l_1, k_\mathrm{e}\rangle \mid \overline{m_\mathrm{i}}k_1\big) \ \mid\ \mathsf{PP}_{\mathbb{O}}\langle s, \widetilde{n}, v_2 \ldots v_n\rangle.$$

By inductive hypothesis it holds that:

$$\mathsf{OM}_{\mathbb{O}}\langle s, \widetilde{m}, \widetilde{k}, \tilde{t}\rangle \mid \prod_{j\in 2..n} \overline{s}v_j \ \approx_{\Gamma;s}\ \mathsf{OM}_{\mathbb{O}}\langle s, \widetilde{m}, \widetilde{k}, \tilde{t}\rangle \mid \mathsf{PP}_{\mathbb{O}}\langle s, \widetilde{n}, v_2 \ldots v_n\rangle.$$

Since $\approx_{\Gamma;s}$ is preserved by parallel composition, for proving our result it suffices to show that:

$$\mathsf{OM}_{\mathbb{O}}\langle s, \widetilde{m}, \widetilde{k}, \tilde{t}\rangle \mid \overline{s}v_1 \ \approx_{\Gamma;s}\ \mathsf{OM}_{\mathbb{O}}\langle s, \widetilde{m}, \widetilde{k}, \tilde{t}\rangle \mid m_\mathrm{e}.\big(\overline{s}\langle l_1, k_\mathrm{e}\rangle \mid \overline{m_\mathrm{i}}k_1\big).$$

Let $A \stackrel{\mathrm{def}}{=} \mathsf{OM}_{\mathbb{O}}\langle s, \widetilde{m}, \widetilde{k}, \tilde{t}\rangle \mid \overline{s}v_1$ and
$B \stackrel{\mathrm{def}}{=} \mathsf{OM}_{\mathbb{O}}\langle s, \widetilde{m}, \widetilde{k}, \tilde{t}\rangle \mid m_\mathrm{e}.\big(\overline{s}\langle l_1, k_\mathrm{e}\rangle \mid \overline{m_\mathrm{i}}k_1\big)$

we prove that the relation:

$$\mathcal{S} = \{\big((\boldsymbol{\nu}\widetilde{z})\,(A \mid R), (\boldsymbol{\nu}\widetilde{z})\,(B \mid R)\big) \ :\ s \notin \widetilde{z} \text{ and } s \text{ not in input in } R\} \cup \mathcal{I}$$

where $\mathcal{I}$ is the identity relation, is a $\approx_{\Gamma;s}$-bisimulation up to structural equivalence. The obligation on types in the coinductive definition of $\approx_{\Gamma;s}$ can be safely omitted. We first show how the right side can emulate the actions performed by the left side and then the vice versa.

From left to right. Let us see the possible actions of $(\boldsymbol{\nu}\widetilde{z})\,(A \mid R)$.

1. If $(\boldsymbol{\nu}\widetilde{z})\,(A \mid R) \xrightarrow{\mu} (\boldsymbol{\nu}\widetilde{y})\,(A \mid R')$ then it is easy.
2. If $(\boldsymbol{\nu}\widetilde{z})\,(A \mid R) \xrightarrow{s\langle l,k\rangle} (\boldsymbol{\nu}\widetilde{z})\,(A' \mid R)$, then there are three possibilities: (i) either $k = k_\mathrm{i}$, or (ii) $k = k_\mathrm{e}$, or (iii) $k_\mathrm{i} \neq k \neq k_\mathrm{e}$. In each case the right side can perform an input $s\langle l, k\rangle$ obtaining a process $(\boldsymbol{\nu}\widetilde{z})\,(B' \mid R)$. By inspection of the encoding we have that $(\boldsymbol{\nu}\widetilde{z})\,(A' \mid R) \equiv (\boldsymbol{\nu}\widetilde{y})\,(A'' \mid R')$ and $(\boldsymbol{\nu}\widetilde{z})\,(B' \mid R) \equiv (\boldsymbol{\nu}\widetilde{y})\,(B'' \mid R')$, for some $\widetilde{y}$ and some process R', where A'' (resp. B'') is the same as A (resp. A''), up to renaming k_i with a fresh key k^*. Therefore $(\boldsymbol{\nu}\widetilde{y})\,(A'' \mid R')\ \mathcal{S}\ (\boldsymbol{\nu}\widetilde{y})\,(B'' \mid R')$.

3. If $(\boldsymbol{\nu}\widetilde{z})\,(A \mid R) \xrightarrow{\tau} (\boldsymbol{\nu}\widetilde{y})\,(A' \mid R')$, where the τ-action is due to a communication along s between A and R (recall that s can only appear in output in R), then we reason similarly to the previous case.
4. If $(\boldsymbol{\nu}\widetilde{z})\,(A \mid R) \xrightarrow{\tau} (\boldsymbol{\nu}\widetilde{z})\,(A' \mid R)$, where the τ actions is due to a communication along s between the object manager and the external request $\overline{s}v_1$, then, by inspection of the encoding, it holds that $A' \equiv B$. On the right side we can mimic the τ action by performing $(\boldsymbol{\nu}\widetilde{z})\,(B \mid R) \Longrightarrow (\boldsymbol{\nu}\widetilde{z})\,(B \mid R)$. It holds that $(\boldsymbol{\nu}\widetilde{z})\,(A' \mid R) \equiv \mathcal{S}\ (\boldsymbol{\nu}\widetilde{z})\,(B \mid R)$.

From right to left. Let us see the possible actions of $(\boldsymbol{\nu}\widetilde{z})\,(B \mid R)$.

1. If $(\boldsymbol{\nu}\widetilde{z})\,(B \mid R) \xrightarrow{\mu} (\boldsymbol{\nu}\widetilde{y})\,(B \mid R')$ then it is easy.
2. If $(\boldsymbol{\nu}\widetilde{z})\,(B \mid R) \xrightarrow{s\langle l,k\rangle} (\boldsymbol{\nu}\widetilde{z})\,(B' \mid R)$, then there are three possibilities: (i) either $k = k_{\mathrm{i}}$, or (ii) $k = k_{\mathrm{e}}$, or (iii) $k_{\mathrm{i}} \neq k \neq k_{\mathrm{e}}$. In each case the left side can perform an input $s\langle l, k\rangle$ obtaining a process $(\boldsymbol{\nu}\widetilde{z})\,(A' \mid R)$. By inspection of the encoding we have that $(\boldsymbol{\nu}\widetilde{z})\,(B' \mid R) \equiv (\boldsymbol{\nu}\widetilde{y})\,(B'' \mid R')$ and $(\boldsymbol{\nu}\widetilde{z})\,(A' \mid R) \equiv (\boldsymbol{\nu}\widetilde{y})\,(A'' \mid R')$, for some $\widetilde{y}$ and some process R', where B'' (resp. A'') is the same as B (resp. A), up to renaming k_{i} with a fresh key k^*. Therefore $(\boldsymbol{\nu}\widetilde{y})\,(B'' \mid R')\ \mathcal{S}\ (\boldsymbol{\nu}\widetilde{y})\,(A'' \mid R')$.
3. If $(\boldsymbol{\nu}\widetilde{z})\,(B \mid R) \xrightarrow{\tau} (\boldsymbol{\nu}\widetilde{y})\,(B' \mid R')$, where the τ-action is due to a communication along s between B and R (recall that s can only appear in output in R), then we reason similarly to the previous case.
4. If $(\boldsymbol{\nu}\widetilde{z})\,(B \mid R) \xrightarrow{m_{\mathrm{e}}} (\boldsymbol{\nu}\widetilde{z})\,(B' \mid R)$ and $B' = \mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{m}, \widetilde{k}, \tilde{t}\,\rangle \mid \overline{s}\langle l_1, k_{\mathrm{e}}\rangle \mid \overline{m_{\mathrm{i}}}k_1$, then the left side can mimic this action by serving the request $\overline{s}v_1$ and then grabbing the mutex. In practise,
$(\boldsymbol{\nu}\widetilde{z})\,(A \mid R) \xrightarrow{\tau} \xrightarrow{m_{\mathrm{e}}} (\boldsymbol{\nu}\widetilde{z})\,(A' \mid R)$ with $A' \equiv B'$. So, $(\boldsymbol{\nu}\widetilde{z})\,(B' \mid R)\ \mathcal{S}\ (\boldsymbol{\nu}\widetilde{z})\,(A' \mid R)$.

$\square$

Proof of Lemma 8.5. It follows directly from Lemma A.5 and the fact that $\approx_{\Gamma;s}$ is preserved by parallel composition and restriction. $\square$

A.8 Proof of Lemma 8.6

Proof. It holds that:

$$\mathsf{pingO}_{\mathbb{O}}\langle\, s, r, k, \widetilde{t}, \widetilde{v}\,\rangle \xrightarrow{\tau}{}^{2}_{\equiv} (\boldsymbol{\nu}\widetilde{m}\widetilde{k})\left(\overline{m_{\mathrm{e}}} \mid \mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{m}, \widetilde{k}, \widetilde{t}\,\rangle \mid \mathsf{PP}_{\mathbb{O}}\langle\, s, \widetilde{n}, \widetilde{v}\,\rangle\right) \mid \overline{r}\langle s, k\rangle.$$

Since $\approx_{\Gamma}$ is insensitive to these two silent moves, we can conclude. $\square$

References

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer, 1996.

[ACS98] R. M. Amadio, I. Castellani and D. Sangiorgi. On Bisimulations for the Asynchronous π-Calculus. *Theoretical Computer Science*, 195(2):291–324, 1998. An extended abstract appeared in *Proceedings of CONCUR '96*, LNCS 1119: 147–162.

[Bou92] G. Boudol. Asynchrony and the π-calculus (Note). Rapport de Recherche 1702, INRIA Sophia-Antipolis, May 1992.

[Car94] L. Cardelli. `obliq-std.exe` — Binaries for Windows NT. `http://www.luca.demon.co.uk/Obliq/Obliq.html`, 1994.

[Car95] L. Cardelli. A Language with Distributed Scope. *Computing Systems*, 8(1):27–59, 1995. Short version in *Proceedings of POPL '95*. A preliminary version appeared as Report 122, Digital Systems Research, June 1994.

[DF96] P. Di Blasio and K. Fisher. A Concurrent Object Calculus. In U. Montanari and V. Sassone, eds, *Proceedings of CONCUR '96*, volume 1119 of *LNCS*, pages 655–670. Springer, 1996. An extended version appeared as Stanford University Technical Note STAN-CS-TN-96-36, 1996.

[FG96] C. Fournet and G. Gonthier. The Reflexive Chemical Abstract Machine and the Join-Calculus. In *Proceedings of POPL '96*, pages 372–385. ACM, Jan. 1996.

[GH98] A. D. Gordon and P. D. Hankin. A Concurrent Object Calculus: Reduction and Typing. In U. Nestmann and B. C. Pierce, eds, *Proceedings of HLCL '98*, volume 16.3 of *ENTCS*. Elsevier Science Publishers, 1998.

[GHL97] A. D. Gordon, P. D. Hankin and S. B. Lassen. Compilation and Equivalence of Imperative Objects. In S. Ramesh and G. Sivakumar, eds, *Proceedings of FSTTCS '97*, volume 1346 of *LNCS*, pages 74–87. Springer, Dec. 1997. Full version available as Technical Report 429, University of Cambridge Computer Laboratory, June 1997.

[HK96] H. Hüttel and J. Kleist. Objects as Mobile Processes. Research Series RS-96-38, BRICS, Oct. 1996. Presented at MFPS '96.

[HKMN99] H. Hüttel, J. Kleist, M. Merro and U. Nestmann. Migration = Cloning ; Aliasing (Preliminary Version). In *Informal Proceedings of the Sixth International Workshop on Foundations of Object-Oriented Languages (FOOL 6, San Antonio, Texas, USA)*. Sponsored by ACM/SIGPLAN, 1999.

[Hon92] K. Honda. Two bisimilarities for the ν-calculus. Technical Report 92-002, Keio University, 1992.

[HT91] K. Honda and M. Tokoro. An Object Calculus for Asynchronous Communication. In P. America, ed, *Proceedings of ECOOP '91*, volume 512 of *LNCS*, pages 133–147. Springer, July 1991.

[HY95] K. Honda and N. Yoshida. On Reduction-Based Process Semantics. *Theoretical Computer Science*, 152(2):437–486, 1995. An extract appeared in *Proceedings of FSTTCS '93*, LNCS 761.

[JLHB88] E. Jul, H. Levy, N. Hutchinson and A. Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions of Computer Systems*, 6(1), Feb. 1988.

[KS98] J. Kleist and D. Sangiorgi. Imperative Objects and Mobile Processes. In D. Gries and W.-P. de Roever, eds, *Proceedings of PROCOMET '98*, pages 285–303. International Federation for Information Processing (IFIP), Chapman & Hall, 1998.

[Mer00] M. Merro. *Locality in the π-calculus and applications to distributed objects*. PhD thesis, Ecole des Mines, France, October 2000.

[Mil93] R. Milner. The Polyadic π-Calculus: A Tutorial. In F. L. Bauer, W. Brauer and H. Schwichtenberg, eds, *Logic and Algebra of Specification*, volume 94 of *Series F: Computer and System Sciences*. NATO Advanced Study Institute, Springer, 1993. Available as Technical Report ECS-LFCS-91-180, University of Edinburgh, October 1991.

[Mor68] J.-H. Morris. *Lambda Calculus Models of Programming Languages*. PhD thesis, MIT, 1968.

[MS92] R. Milner and D. Sangiorgi. Barbed Bisimulation. In W. Kuich, ed, *Proceedings of ICALP '92*, volume 623 of *LNCS*, pages 685–695. Springer, 1992.

[MS98] M. Merro and D. Sangiorgi. On Asynchrony in Name-Passing Calculi. In K. G. Larsen, S. Skyum and G. Winskel, eds, *Proceedings of ICALP '98*, volume 1443 of *LNCS*, pages 856–867. Springer, July 1998.

[NHKM00] U. Nestmann, H. Hüttel, J. Kleist and M. Merro. Aliasing Models for Mobile Objects. Accepted for *Journal of Information and Computation*. Available from `http://www.cs.auc.dk/research/FS/ojeblik/`. An extended abstract has appeared as Distinguished Paper in the *Proceedings of EUROPAR '99*, LNCS 1685, 2000.

[PS96] B. C. Pierce and D. Sangiorgi. Typing and Subtyping for Mobile Processes. *Mathematical Structures in Computer Science*, 6(5):409–454, 1996. An extract appeared in *Proceedings of LICS '93*: 376–385.

[PW98] A. Philippou and D. Walker. On Transformations of Concurrent Object Programs. *Theoretical Computer Science*, 195(2):259–289, 1998. An extended abstract appeared in *Proceedings of CONCUR '96*, LNCS 1119: 131–146.

[San98] D. Sangiorgi. An Interpretation of Typed Objects into Typed π-Calculus. *Information and Computation*, 143(1):34–73, 1998. Earlier version published as Rapport de Recherche RR-3000, INRIA Sophia-Antipolis, August 1996.

[San99a] D. Sangiorgi. The Name Discipline of Uniform Receptiveness. *Theoretical Computer Science*, 221(1–2):457–493, 1999. An abstract appeared in the *Proceedings of ICALP '97*, LNCS 1256, pages 303–313.

[San99b] D. Sangiorgi. The Typed π-Calculus at work: A Proof of Jones's Parallelisation Theorem on Concurrent Objects. *Theory and Practice of Object-Oriented Systems*, 5(1), 1999. An early version was included in the *Informal proceedings of FOOL 4*, January 1997.

[San00] D. Sangiorgi. Lazy Functions and Mobile Processes. In G. Plotkin, C. Stirling and M. Tofte, eds, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, Foundations of Computing. MIT Press, May 2000. Available as INRIA Sophia-Antipolis Rapport de Recherche RR-2515.

[SW01] D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes.* Cambridge University Press, 2001. To appear.

[VHB+97] P. Van Roy, S. Haridi, P. Brand, G. Smolka, M. Mehl and R. Scheidhauer. Mobile Objects in Distributed Oz. *ACM Transactions on Programming Languages and Systems*, 19(5):804–851, Sept. 1997.

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

Contents