Mobile Objects as Mobile Processes*

Massimo Merro
EPFL, Lausanne, Switzerland

Josva Kleist†
BRICS, Aalborg, Denmark

Uwe Nestmann
EPFL, Lausanne, Switzerland

Technical Report IC/2002/72

October 30, 2002

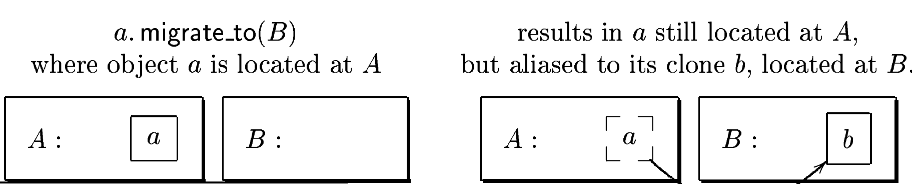
Abstract

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 the *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 tool-box 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 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. For example, let A and B be names of distribution sites, then

$a.\,\mathsf{migrate_to}(B)$ where object a is located at A	results in a still located at A, but aliased to its clone b, located at B.



*An extended abstract has appeared in *Proceedings of IFIP TCS 2000*, volume 1872 of *Lecture Notes in Computer Science*. Springer Verlag, August 2000.
†Partly supported by Danish National Research Foundation grant SNF-28808

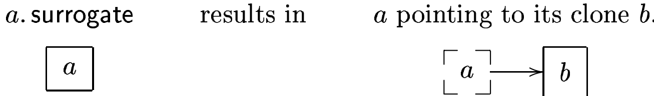
The aim of the current paper is to rigorously study the question whether this form of object migration can be considered as correct in any formal sense.

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 after the migration (ii) the alias cannot be corrupted, and (iii) no client of a can tell that a is now an alias. In Obliq, mobile objects are therefore required to be *serialised* and *protected*: for (i), serialisation guarantees atomicity of the two-phase migration protocol; for (ii), 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 provided that sites do not fail; site failure would allow clients to trivially observe whether an object on one site has moved to another site. 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. Consequently,

$a.\mathsf{surrogate}$ results in a pointing to its clone b.



Correctness as an equation. In [NHKM00], we motivated a precise definition of *correctness* for object surrogation in Øjeblik. The intuition is that the surrogation of an object must be transparent to the clients of that object. It is formalised by means of an equation:

$$x.\mathsf{ping}; x \doteq x.\mathsf{surrogate}; x \tag{1}$$

where x is supposed to be a variable giving access to an object and ";" represents sequential composition. On the left side of the equation, $x.\mathsf{ping}$ returns a reference to the object resulting from the evaluation of x; the following occurrence of x denotes a reference to the same object. On the right side of the equation, $x.\mathsf{surrogate}$ returns a reference to the clone; the following occurrence of x denotes a reference to the proxy. The relation $\doteq$ is a contextual equivalence, based on the possibility of convergence, i.e., it requires corresponding convergence behaviour of the two terms with respect to all contexts.

However, equation (1) prevents any context from actually using the reference to the new clone, as returned from $x.\mathsf{surrogate}$. Therefore, we strengthen it to

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

which compares the respective client-accesses on two different references, namely the original reference on the left hand side (as returned from the ping-operation) with the newly created reference to the clone object on the right hand side (as returned from the surrogate-operation).

This equation, which we call *safety equation*, is strictly stronger than equation (1) in the sense that the set of contexts in which it must validate the convergence properties includes the respective set of contexts needed for equation (1).

Aliasing semantics. Since surrogation is supposed to be implementable as a combination of aliasing and cloning, the proper modelling of aliases is crucial for the correctness of surrogation. Intuitively, when an object becomes alias it should simply *forward* the received requests to the target object. Of course this must be done in accordance with the peculiarities of the requests themselves and also with protection and serialisation requirements. In [NHKM00], we proposed (among others) a formal configuration-style semantics for Øjeblik that was guided by this intuition. In that paper, we also conjectured that the above safety equation holds in the proposed semantics, but no proof was given. The reason for this was the lack of theory and proof tools in that rather ad-hoc setting.

1.2 Contribution

In the current paper, we present a π-calculus semantics for Øjeblik corresponding to the semantics proposed in [NHKM00]. More precisely, our semantics uses an extension of *Localised* π [MS98, Mer00b], in short Lπ, 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 created the channel. The choice of Lπ as the target language is not by accident: one of its fundamental laws is the *forwarder law* (cf. Lemma 2.2.14)

$$\overline{a}b = (\boldsymbol{\nu} c)\,(\overline{a}c \mid\, !c(x).\overline{b}x) \tag{3}$$

where $\mid$ represents parallel composition and ! replication to create as many parallel replicas as needed. If $c \neq b$, then law (3) equates processes that may perform the syntactically different outputs $\overline{a}b$ and $(\boldsymbol{\nu} c)\,\overline{a}c$: the process on the left performs the output of a global name b, whereas the one on the right performs the output of a private name c. The process $!c(x).\overline{b}x$ makes the two sides of the law indistinguishable by forwarding values received at channel c to channel b. Several applications of the forwarder law can be found in [MS98, Mer00a, Mer00b]. The strong similarity between the forwarder law (3) and the forwarding concept of migration in Obliq makes Lπ a promising candidate to provide proof techniques for the safety equation. Indeed, using law (3) in a central position of our technical development (cf. Lemma 7.1.4), we prove the correctness of surrogation for a wide class of Øjeblik-programs. As already pointed out in [NHKM00], equation (2) *cannot* be true for all Øjeblik-programs, due to the inherent possibility of objects to modify their own internal state without any control of when such modifications is performed. The situation is analogous to a program that performs a division by x after having itself in the local scope of x assigned 0 to x. In this respect, our proof concerns the class of all programs in which clients may freely perform surrogation on object different from itself. In Section 6.2 we explain this in more detail.

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 the 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.

1.4 Acknowledgments

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 our work. The paper has changed and improved a lot as a consequence of the suggestions and remarks of the four anonymous referees. We are very grateful to them for their insightful and instructive criticisms.

1.5 Outline

In Section 2 we introduce the π-calculus on which we interpret Øjeblik. Section 3 presents the syntax and informally explains the semantics of Øjeblik. Section 4 is devoted to the translation of Øjeblik into the π-calculus. In Section 5 we show some properties enjoyed by the translation. Section 6 lays the ground work for the proof of safe surrogation, and in Section 7 we prove the main result of the paper. Section 8 contains conclusions. Finally, Appendix A contains proofs omitted from the main part of the paper.

2 The typed Localised π-calculus

Localised π [MS98, Mer00b], 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 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 Lπ^+, a typed version of polyadic Lπ with: (i) labelled values ℓ_v, also called *variants* [San98], with case analysis; (ii) tuple values $\langle\, v_1 .. v_n \,\rangle$ with pattern matching, (iii) value testing.

We introduce a few syntactic categories: the set $\mathbf{N}$ of *names*, the set $\mathbf{X}$ of *variables*, and the set $\mathbf{L}$ of *labels*. *Values* consist of names, variables, variants, and tuples. We use variables to model the requirement that the recipient of a channel may only use it in output actions. This is achieved by disallowing inputs on variables.

Restriction binds names, whereas both inputs, and both destructors are *binders* for the variables $x, x_1, \ldots, x_m$ in the respective scopes $P, P_1, \ldots, P_m$. We assume the usual definitions of free and bound occurrences of names and variables, based on these binders; the inductively defined functions $\mathrm{fn}(P)$ and $\mathrm{bn}(P)$ (resp. $\mathrm{fv}(P)$ and $\mathrm{bv}(P)$) denote those of process P; the names (resp. variables) of P, written $\mathrm{n}(P)$ (resp. $\mathrm{v}(P)$), are given by $\mathrm{fn}(P) \cup \mathrm{bn}(P)$ (resp. $\mathrm{fv}(P) \cup \mathrm{bv}(P)$). Sometimes, $\mathrm{fn}(P, Q)$ is used as a shorthand for $\mathrm{fn}(P) \cup \mathrm{fn}(Q)$. A process P is *closed* if $\mathrm{fv}(P) = \emptyset$. Unless explicitly stated we only consider closed processes. *Substitutions*, denoted by $\{^v\!/\!_x\}$ and ranged over by σ, are mappings from variables to values. 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 $\ell'' \neq \ell$
- $x\rho := x$

Table 1: The Calculus Lπ^+

Names:		$a, b, c, k, \ldots \in \mathbf{N}$	
Variables:		$x, y, u, z \in \mathbf{X}$	
Labels		$\ell \in \mathbf{L}$	
Values			
l, v, w	$::=$	a	name
	$\mid$	x	variable
	$\mid$	ℓ_v	variant
	$\mid$	$\langle v_1 .. v_n \rangle$	tuple
Value types			
T	$::=$	$\mathbf{C}(T)$	channel type
	$\mid$	$\mathbf{K}$	key type
	$\mid$	$[\ell_1{:}T_1 ; \ldots ; \ell_m{:}T_m]$	variant type
	$\mid$	$\langle T_1 .. T_m \rangle$	tuple type
	$\mid$	X	type variable
	$\mid$	$\mu X . T$	recursive type
Processes			
P	$::=$	$\mathbf{0}$	nil process
	$\mid$	$a(x).P$	single *input*
	$\mid$	$\overline{v}w$	output
	$\mid$	$P_1 \mid P_2$	parallel
	$\mid$	$(\boldsymbol{\nu} a{:}T)\, P$	restriction
	$\mid$	$!\, a(x).P$	replicated *input*
	$\mid$	if $[v{=}v_1]$ then P_1 elif $[v{=}v_2]$ 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*

- $((\boldsymbol{\nu} n{:}T)\, P)\rho := (\boldsymbol{\nu} n{:}T\rho)\, P\rho$
- $(\mathsf{case}\ v\ \mathsf{of}\ \ell_1_(x_1){:}P_1\,;\ldots;\ell_n_(x_n){:}P_n)\rho$
  $:= \mathsf{case}\ v\rho\ \mathsf{of}\ \ell_1_(x_1){:}(P_1\rho)\,;\ldots;\ell_n_(x_n){:}(P_n\rho)$.

For the remaining (value and process) constructors, relabellings act as simple homomorphisms. Substitution and relabelling have the highest operator precedence, parallel composition the lowest. In processes $c(x).P$ and $\overline{c}v$, channel c is the *subject* and x and v are the *object* parts.

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 $\tilde{v}$ denotes a sequence $v_1\,..\,v_m$. We often omit the type annotation of restriction, when it is clear from the context or not important for the discussion.

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

Definition 2.1.1 Structural congruence*, written* $\equiv$*, is the smallest congruence over closed processes 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* $k_1{\neq}k{\neq}k_2$
- $\mathsf{case}\ \ell_j_v_j\ \mathsf{of}\ \ell_1_(x_1){:}P_1\,;\ldots;\ell_j_(x_j){:}P_j\,;\ldots;\ell_m_(x_m){:}P_m \equiv P_j\{^{v_j}\!/\!_{x_j}\}$
- $\mathsf{let}\ (\,x_1\,..\,x_m\,) = \langle\,v_1\,..\,v_m\,\rangle\ \mathsf{in}\ P \equiv P\{^{\tilde{v}}\!/\!_{\tilde{x}}\}$.

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. Our type system divides names in *channels* and *keys*: the former may be used in both subject and object position whereas the latter may only be used in object position and testing. We use a type constructor $\mathbf{C}(T)$ for channels carrying values of type T. $\mathbf{K}$ is the type constructors for *keys*. In the sequel, we let s, p, q, r, m, t range over channels and $k, k', \ldots$ over keys. Variant and tuple types are standard (cf. [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. A *type environment* Γ is a finite mapping of variables to value types, and names to either channel or key 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 $\mathrm{dom}(\Gamma) \cap \mathbf{X} = \emptyset$. As expected, the typing in Table 2 satisfies all basic fundamental properties of type environments such as: *weakening*, *contraction*, *substitution*, and *narrowing*.

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. The rules (CASE-W) and (LET-W) with their introduction of wrong transitions (c.f. [KS98]) provide the detection of run-time errors in Theorem 2.2.1; since our calculus offers both variants and tuples, and since it separates value input from value destruction, the premises of both rules have to be more explicit than usual. *Labelled transitions* are of the form $P \xrightarrow{\mu} P'$, where *action* μ is: τ (interaction), cv (free input), $(\boldsymbol{\nu}\tilde{n}{:}\widetilde{T})\,\overline{c}v$ (output at c of value v containing private names $\tilde{n}$ of type $\widetilde{T}$, which we often omit, or wrong (run-time error). The functions $\mathrm{fn}(\cdot)$, $\mathrm{bn}(\cdot)$, $\mathrm{n}(\cdot)$, are extended to actions as usual. Relation $\Rightarrow$ is the reflexive-transitive closure of $\xrightarrow{\tau}$; $\overset{\mu}{\Longrightarrow}$ denotes $\Rightarrow \xrightarrow{\mu} \Rightarrow$. 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*.

Table 2: Typing for Values and Processes

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

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

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

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

$$(\text{T-RES1})\ \frac{\Gamma, c{:}\mathbf{C}(T) \vdash P}{\Gamma \vdash (\boldsymbol{\nu} c{:}\mathbf{C}(T))\ P} \qquad (\text{T-RES2})\ \frac{\Gamma, k{:}\mathbf{K} \vdash P}{\Gamma \vdash (\boldsymbol{\nu} k{:}\mathbf{K})\ P}$$

$$(\text{T-PAR})\ \frac{\Gamma \vdash P_1 \quad \Gamma \vdash P_2}{\Gamma \vdash P_1 \mid P_2} \qquad (\text{T-REP})\ \frac{\Gamma \vdash P}{\Gamma \vdash\ !P}$$

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

$$(\text{T-IF})\ \frac{\Gamma \vdash v, v_1, v_2 : \mathbf{K} \qquad \Gamma \vdash P_1, P_2, P_3}{\Gamma \vdash \mathsf{if}\ [v{=}v_1]\ \mathsf{then}\ P_1\ \mathsf{elif}\ [v{=}v_2]\ \mathsf{then}\ P_2\ \mathsf{else}\ P_3}$$

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

$$(\text{T-CASE})\ \frac{\Gamma \vdash v : [\,\ell_1{:}T_1\,;\ldots;\ell_m{:}T_m\,] \qquad \Gamma, x_i{:}T_i \vdash P_i \quad \forall i \in 1\,..\,m}{\Gamma \vdash \mathsf{case}\ v\ \mathsf{of}\ \ell_1_(x_1){:}P_1\,;\ldots;\ell_m_(x_m){:}P_m}$$

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

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

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

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

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

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

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

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

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

$$(\textsc{Case})\ \frac{P_j\{^{v}/_{x_j}\} \xrightarrow{\mu} Q \qquad j \in 1\,..\,m}{\textsf{case } \ell_j_v \textsf{ of } \ell_1_(x_1){:}P_1\,;\ldots;\ell_m_(x_m){:}P_m \xrightarrow{\mu} Q}$$

$$(\textsc{Case-W})\ \frac{v \neq \ell_j_v_j \quad \forall j \in 1\,..\,m \quad \text{and any value } v_j}{\textsf{case } v \textsf{ of } \ell_1_(x_1){:}P_1\,;\ldots;\ell_m_(x_m){:}P_m \xrightarrow{\textsf{wrong}} \mathbf{0}}$$

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

$$(\textsc{Let-W})\ \frac{v \neq \langle\, v_1\,..\,v_m\,\rangle \quad \text{for any values } v_1\,..\,v_m}{\textsf{let}\ (\,x_1\,..\,x_m\,) = v \textsf{ in } P \xrightarrow{\textsf{wrong}} \mathbf{0}}$$

Theorem 2.2.1 (Type Soundness) *Let Γ be a closed type environment.*

1. *If $\Gamma \vdash P$ then $P \stackrel{\texttt{wrong}}{\not\longrightarrow}$.*
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 2.2.2 (Asynchronous observability) *We write $P\downarrow_c$ if there is a derivative P', and an output action μ with subject c, such that $P \stackrel{\mu}{\longrightarrow} P'$. We write $P\Downarrow_c$ if there is P' such that $P \Rightarrow P'$ and $P'\downarrow_c$.*

Definition 2.2.3 (Barbed bisimilarity) *A symmetric relation $\mathcal{S}$ over closed processes is a* barbed bisimulation *if $P \; \mathcal{S} \; Q$ implies:*

- *If $P \stackrel{\tau}{\longrightarrow} P'$ 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. However, since $\mathrm{L}\pi^+$ is a typed calculus, only well-typed contexts should be considered [PS96, SW01].

Definition 2.2.4 *A* 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 $(\boldsymbol{\nu}\tilde{n})\,(P \mid [\cdot])$, for some P and $\tilde{n}$.*

Definition 2.2.5 *Let Γ and Δ be two type environments. We say that Γ* extends *Δ if $\Delta \subseteq \Gamma$. We say that Γ is a* closed extension *of Δ if Γ is closed and extends Δ.*

Definition 2.2.6 *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:*

$$(\text{T-HOLE}) \; \frac{\Theta \text{ extends } \Gamma}{\Theta \vdash [\cdot]}$$

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

The intuition of Definition 2.2.6 is that if $\Gamma \vdash P$ and $C[\cdot]$ is a (Δ/Γ)-context then $\Delta \vdash C[P]$.

Definition 2.2.7 (Typed barbed relations) *Let Γ be a typing, and P and Q two closed 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 equivalence/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 equivalence/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 2.2.8 (Typed bisimilarity) Typed bisimilarity *is the largest typed relation $\mathcal{S}$ over closed processes such that $(\Delta; P; Q) \in \mathcal{S}$ implies:*

1. *If $P \xrightarrow{\tau} P'$, then there exists Q' such that $Q \Rightarrow Q'$ and $(\Delta; P'; Q') \in \mathcal{S}$.*
2. *If $P \xrightarrow{(\nu \tilde{n}:\widetilde{T})\, \bar{c}v} P'$, with $\tilde{n} \cap \mathrm{fn}(Q) = \emptyset$, then there exists Q' such that $Q \xRightarrow{(\nu \tilde{n}:\widetilde{T})\, \bar{c}v} Q'$ and $((\Delta, \tilde{n}:\widetilde{T}); P'; Q') \in \mathcal{S}$.*
3. *If*
   - *(i) Γ is a closed extension of Δ,*
   - *(ii) $\Gamma \vdash c{:}\mathbf{C}(T)$, $\Gamma \vdash v{:}T$ and for each $a \in \mathrm{fn}(v) \cap \mathrm{fn}(P, Q)$ it holds that $\Gamma \vdash a{:}\mathbf{K}$,*
   - *(iii) $P \xrightarrow{cv} P'$*

   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 $(\Gamma; P'; (\,Q' \mid \bar{c}v\,)) \in \mathcal{S}$.*

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 that for each $a \in \mathrm{fn}(v) \cap \mathrm{fn}(P, Q)$ it holds that $\Gamma \vdash a{:}\mathbf{K}$. In asynchronous calculi without name testing, ground and early bisimilarity coincide [San00, Hon92]. Since in well-typed processes we test only 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 in the input clause that for each $a \in \mathrm{fn}(v) \cap \mathrm{fn}(P, Q)$ it holds that $\Gamma \vdash a{:}\mathbf{K}$, 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 $\simeq_\Gamma$.

Later on, we will deal with processes containing channels that can be used by the environment only in output. This will require to refine the Definitions 2.2.7 and 2.2.8 by adding the set $\mathcal{C}$ of the channels that cannot be used in input by the environment.

Definition 2.2.9 (Barbed Γ;$\mathcal{C}$-relations) *Let $\mathcal{C} \subseteq \mathbf{C}$.* Barbed $\mathcal{C}$-bisimilarity, *written $\dot{\cong}_{\mathcal{C}}$, is the largest symmetric relation over closed 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 closed processes such that $\Gamma \vdash P, Q$. We say that P and Q are barbed Γ;$\mathcal{C}$-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 Γ;$\mathcal{C}$-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 $C[P] \dot{\cong}_{\mathcal{C}} C[Q]$.*

In Definition 2.2.9, when $\mathcal{C}{=}\emptyset$, we get the above definitions of typed barbed bisimilarity (c.f. Definition 2.2.7). If $\mathcal{C}{=}\{s\}$, as abbreviations, we write $\simeq_{\Gamma;s}$ for $\simeq_{\Gamma;\mathcal{C}}$ and $\cong_{\Gamma;s}$ for $\cong_{\Gamma;\mathcal{C}}$. Due to the restriction on the contexts, it holds that $\bar{s}v \cong_{\Gamma;s} \mathbf{0}$ and, by asynchrony, $s(x).\mathbf{0} \cong_{\Gamma;s} \mathbf{0}$. Below, we give the labelled counterpart of barbed Γ;$\mathcal{C}$-equivalence.

Definition 2.2.10 (Typed $\mathcal{C}$-bisimilarity) Typed $\mathcal{C}$-bisimilarity *is the largest typed relation $\mathcal{S}$ over closed processes such that $(\Delta; P; Q) \in \mathcal{S}$ implies:*

1. *If $P \xrightarrow{\tau} P'$, then there exists Q' such that $Q \Rightarrow Q'$ and $(\Delta; P'; Q') \in \mathcal{S}$.*
2. *If $P \xrightarrow{(\boldsymbol{\nu}\tilde{n}:\widetilde{T})\,\overline{c}v} P'$, with $c \notin \mathcal{C}$ and $\tilde{n} \cap \mathrm{fn}(Q) = \emptyset$, then there exists Q' such that $Q \xRightarrow{(\boldsymbol{\nu}\tilde{n}:\widetilde{T})\,\overline{c}v} Q'$ and $((\Delta, \tilde{n}:\widetilde{T}); P'; Q') \in \mathcal{S}$.*
3. *If*
   (i) Γ is a closed extension of Δ,
   (ii) $\Gamma \vdash c{:}\mathbf{C}(T)$, $\Gamma \vdash v{:}T$, and for each $a \in \mathrm{fn}(v) \cap \mathrm{fn}(P, Q)$ it holds that $\Gamma \vdash a{:}\mathbf{K}$,
   (iii) $P \xrightarrow{cv} P'$

   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 $\mathcal{C}$-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 2.2.11 *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 in the input clause that for each $a \in \mathrm{fn}(v) \cap \mathrm{fn}(P, Q)$ it holds that $\Gamma \vdash a{:}\mathbf{K}$, 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 2.2.14 we prove a peculiar algebraic law of $\mathrm{L}\pi^+$ which will be used to prove one of the crucial results of the paper (Theorem 7.1.1). 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 2.2.12 (Link) *Given two names 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 2.2.14, we need the following technical lemma. For convenience, we sometimes use substitutions as mappings from names to names.

Lemma 2.2.13 *Let p and q be two different names, 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*

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

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

Lemma 2.2.14 (Forwarder law) *Let $\Gamma \vdash \overline{p}v$, for some type environment Γ. Let $q \in \mathrm{fn}(v)$ with $\Gamma \vdash q{:}\mathbf{C}(T)$. Let $r \notin (\mathrm{n}(v) \cup \{p, q\})$ and $w = v\{{}^{r}\!/_{q}\}$. Then*

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

Table 4: Øjeblik Syntax and Types

a, b	$::= \mathbb{O}$	object
	$\mid a.\mathrm{l}\langle\, a_1 \ldots a_n \,\rangle$	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	$::= \mu X[\mathrm{l}_j{:}\widetilde{B}_j{\to}B_j]_{j\in J}$	object record type
	$\mid X$	type variable
	$\mid \mathrm{Thr}(A)$	thread type

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

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

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

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

we show that the relation

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

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 $R \xrightarrow{pw} Q$. In this case we get, up to structural congruence, the pair of processes

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

By Lemma 2.2.13, 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 $\mathrm{Thr}(A)$ and recursive object types. Pairs $\tilde{x}_j{:}\widetilde{B}_j$ denote sequences $x_{1_j}{:}B_{1_j}, \ldots, x_{n_j}{:}B_{n_j}$. 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 less burdened, 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 for the argument variables $\tilde{x}$ (if any) 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 shallow copy, i.e., an object a' with the same fields as a, and initialises the fields to the same entries as a; as result, the operation returns a reference to a'.

The operation $a.\mathsf{alias}\langle b\rangle$ replaces object a with a proxy for b, written $a{\gg}b$, and returns a reference to b. We use the term *alias* interchangeably with the term *proxy.* If b is itself an alias, e.g. $b{\gg}c$, then we consequently and naturally create an *alias chain* $a{\gg}b{\gg}c$; we call *nodes* both the aliases of a chain as well as the object at its end. Essentially, invocations arriving at $a{\gg}b$ are forwarded to b. For a more precise discussion of the semantics of aliasing, see the comments in the paragraph on page 15.

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. Like requests for standard methods, requests for surrogation are forwarded by alias nodes. This is also 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. Although not present in Obliq, 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 x to the result, 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, and we often omit the type annotation of the bound variable. 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.

Obliq/Øjeblik-specific object properties. Objects in Obliq can be equipped with keywords `protected` and `serialized`. Their meaning crucially depends on the notion of self-infliction, which we introduce in the following together with some basic terminology. By definition, a client request for an operation on an Øjeblik-object may appear (i) either somewhere within a method body of the object itself, or (ii) just within a let-body, or (iii) at top-level. The *current self of a request* denotes, in case (i), the self of its surrounding method declaration; in the other cases, we may leave it undefined, as we only need to make sure that it is different from all possible object identities. A request for an Øjeblik operation is called *self-inflicted*, or: *internal*, if it addresses its current self; an operation is *external* if it is not self-inflicted. Self-inflicted operations can only be requested from within method bodies; for instance

$$[\mathrm{l}{=}\varsigma(s)s.\mathsf{clone}].\mathrm{l} \tag{4}$$

leads to an internal clone-request. However, not only literal invocations on the self variable s may be self-inflicted, but also indirect invocations on expressions that evaluate to the object itself may be self-inflicted. For instance, in

$$\mathsf{let}\ x = [\mathrm{l}{=}\varsigma(s,z)z.\mathsf{clone}]\ \mathsf{in}\ x.\mathrm{l}\langle x\rangle \tag{5}$$

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

`serialized`: 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 *mutexes* with objects, which must be locked when a thread enters an 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, 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. For instance, the program

$$\mathsf{let}\ x = [\mathrm{l}{=}\varsigma(s)s.\mathrm{k}\,,\mathrm{k}{=}\varsigma(s)s]\ \mathsf{in}\ x.\mathrm{l}$$

will terminate (delivering as a result the identity of x), because the self-recursive (internal) call to method k is permitted. In contrast, the program

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

which attempts a mutual recursion between the objects x and y, will block as soon as the recursive (external) call from y to x for method m is requested, because the mutex x is already locked by the former call of l on x, which has not yet terminated.

`protected`: 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. For instance, the above terms (4) and (5) represent successful calls that terminate (with a result), while

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

blocks (without result), because the clone-request is external.

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, in (our abstraction) Øjeblik *all objects are both protected and serialised*, as opposed to Obliq, where these properties must be specified explicitly for each object by means of keywords.

Aliasing. Surrogation is supposed to be implementable as a combination of aliasing and cloning, so the proper modelling of aliases is crucial for the correctness of surrogation.[1]

As motivated earlier, alias nodes behave like forwarders for invocations. Øjeblik does not only offer invocations on objects, but also cloning, aliasing, and updates, so we have to specify the behaviour of aliases for requests of this kind, too. However, since these operations are protection-critical, it is not obvious whether external and internal requests shall be treated in the same manner with respect to forwarding. In fact, we cannot ask aliases to blindly forward any kind of requests, because the original informal semantics of Obliq [Car95] explicitly requires that if a is an alias to b, i.e. $a \gg b$, a re-aliasing request $a.\mathsf{alias}\langle c \rangle$ results in $a \gg c$. Thus, alias requests must not be forwarded if they can be served, i.e., if they are internal; external alias requests may well be forwarded, though. It makes sense to treat cloning requests in the same way, because like aliasing they can be served locally when internal; moreover, forwarded internal clone requests would be doomed to fail in the target, because no two objects can have the same identity. In contrast, external clone requests are forwarded. All other requests (update, invocation, ping, surrogation) are forwarded as well, irrespective of being external or internal.

Summing up, our semantics models aliases as almost pure forwarders for requests (except for internal aliasing and cloning requests), so any interaction of a client with an alias is likely to be indistinguishable from an interaction with the target of the alias, and thus likely to allow us to prove the safety equation (2). In the following section, we formulate the semantics of alias nodes in terms of Lπ, where each alias is explicitly represented as an alias manager process that behaves essentially like a forwarder for messages at the level of the π-calculus (cf. Lemma 7.1.3).

Types. 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 A translational semantics for Øjeblik

In this section we give a *translational semantics* of Øjeblik into Lπ^+ according to the informal semantics given in Section 3. In addition to the syntax of Lπ^+ 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 $y \notin \mathrm{fn}(P)$;
- *parameterised recursive definitions* $\mathsf{A}(x_1 \mathinner{..} x_n) \stackrel{\text{def}}{=} P$ and the corresponding *instantiations* $\mathsf{A}\langle\, x_1 \mathinner{..} x_n \,\rangle$, which can be faithfully represented in terms of replication [Mil93]. The typing rule associated with a recursive definition is the standard rule, requiring the body to be well-typed under the assumption that the process name is well-typed.

[1] In fact, our previous paper [NHKM00] describes a whole range of different *aliasing models* for Øjeblik, one of them also corresponding to the actual implementation of Obliq which we proved to invalidate the safety equation; here we motivate just the aliasing model that we conjectured in the aforementioned paper to behave correctly with respect to surrogation.

Table 5: Typing Rules for Øjeblik

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

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

$$\text{(T-Obj)}\ \frac{\begin{array}{c} A = \mu X[\mathrm{l}_j{:}\widetilde{B}_j \to B_j]_{j \in J} \\ \forall j \in J \quad \Gamma, s_j{:}A, \tilde{x}_j{:}\widetilde{B}_j\{A/X\} \vdash b_j{:}B_j\{A/X\} \end{array}}{\Gamma \vdash [\mathrm{l}_j = \varsigma(s_j{:}A, \tilde{x}_j{:}\widetilde{B}_j) b_j]_{j \in J} : A}$$

$$\text{(T-Inv)}\ \frac{\Gamma \vdash a : \mu X[\mathrm{l}_j{:}\widetilde{B}_j \to B_j]_{j \in J} \qquad \Gamma \vdash \tilde{b}_k{:}\widetilde{B}_k\{A/X\} \qquad k \in J}{\Gamma \vdash a.\mathrm{l}_k\langle \tilde{b}_k \rangle : B_k\{A/X\}}$$

$$\text{(T-Upd)}\ \frac{\begin{array}{c} \Gamma \vdash a{:}A \qquad A = \mu X[\mathrm{l}_j{:}\widetilde{B}_j \to B_j]_{j \in J} \\ \Gamma, s{:}A, \tilde{x}{:}\widetilde{B}_k\{A/X\} \vdash b{:}B_k\{A/X\} \qquad k \in J \end{array}}{\Gamma \vdash a.\mathrm{l}_k \Leftarrow \varsigma(s{:}A, \tilde{x}{:}\widetilde{B}_k) b : A}$$

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

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

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

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

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

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

$$
\begin{aligned}
[\![\, \mathsf{let}\, x = a \,\mathsf{in}\, b \,]\!]_p^k &\stackrel{\text{def}}{=} (\boldsymbol{\nu} q) \big([\![\, a \,]\!]_q^k \mid q(x, k') \,.\, [\![\, b \,]\!]_p^{k'} \big) \\
[\![\, x \,]\!]_p^k &\stackrel{\text{def}}{=} \overline{p}\langle x, k \rangle
\end{aligned}
$$

$$
\begin{aligned}
[\![\, \mathsf{fork}(a) \,]\!]_p^k &\stackrel{\text{def}}{=} (\boldsymbol{\nu} q t) \big([\![\, a \,]\!]_q^{\boldsymbol{\nu}} \mid \overline{p}\langle t, k \rangle \mid q(x, k').t(r, k'').\overline{r}\langle x, k'' \rangle \big) \\
[\![\, \mathsf{join}(b) \,]\!]_p^k &\stackrel{\text{def}}{=} (\boldsymbol{\nu} q) \big([\![\, b \,]\!]_q^k \mid q(t, k') \,.\, \overline{t}\langle p, k' \rangle \big)
\end{aligned}
$$

For the sake of simplicity, by abuse of notation, in bindings we often use names instead of variables, but making sure that those names are are never used in input subject position conforming to the locality constraint.

The semantics is presented in Tables 6 and 7. To separate the concerns of accessing an object (possibly from within itself) and of checking the identity of an object, we use *access channels* s corresponding to the notion of references used in this paper, and *keys* k that represent *identities*. The translation is then a mapping $[\![\, \cdot \,]\!]_p^k$ parameterised on two names: in a translated term $[\![\, a \,]\!]_p^k$ the channel p is used to return the term's result, while the key k (possibly the identity of an object) represents the term's *current self* (cf. page 14), which we use 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 5.1 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 $[\![\, a.\mathrm{l}_j \langle\, a_1 \,..\, a_n \,\rangle \,]\!]_p^k$, 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. Note that there is no guarantee that the return channels will be used, due to the possibility of blocking or divergence. 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 access 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; since the key is fresh, we have the guarantee that it is different from every current or future object identity. We use $[\![a]\!]_q^{\nu}$ to abbreviate $(\nu k)\,([\![a]\!]_q^k)$. The term $[\![\mathsf{fork}\langle a \rangle]\!]_p^k$ immediately returns on p a private name t representing a fresh thread id, which can be used to retrieve the value of a from the forked thread. Therefore, $[\![\mathsf{join}\langle b \rangle]\!]_p^k$ sends along the value t of b its own result channel p, together with its latest current self k'.

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 $\mathsf{newO}_{\mathbb{O}}\langle\, s, \tilde{t}\,\rangle$ that connects invocations on s from the outside to the method bodies, which are invoked by the trigger names $\tilde{t}$. 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 $\mathsf{newO}_{\mathbb{O}}\langle\, s, \tilde{t}\,\rangle$ 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 support serialisation in our concurrent environment (see later on).

Our semantics associates an object manager OM with 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 a "game" played for each object with 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 (in the sense that an $\overline{m_{\mathrm{i}}}$ appears iff the corresponding $\overline{m_{\mathrm{e}}}$ disappears, and vice versa) and is used to intermediately save some context information. So, 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 (cf. Lemma 7.1.3), 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 seri-

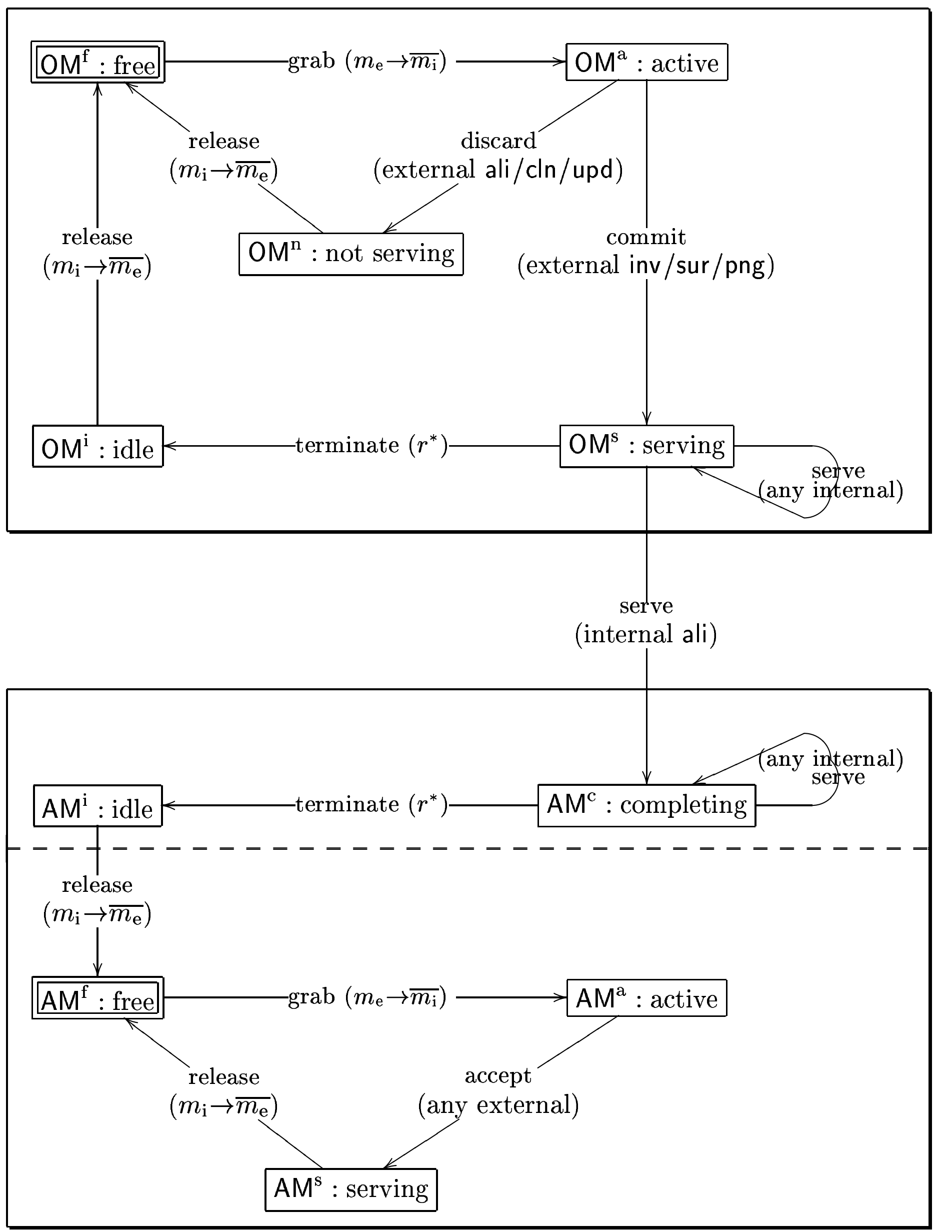


Figure 1: Object and Alias Manager Serving Requests

Table 7: Translational Semantics of Øjeblik — Objects

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

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

$$\begin{array}{l}
\mathsf{OM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k_{\mathrm{i}},\tilde{t}\,\rangle \stackrel{\text{def}}{=} s(l,k).(\boldsymbol{\nu} k^*)\Big(\\
\mathsf{if}\ [k{=}k_{\mathrm{i}}]\ \mathsf{then}\\
\quad \mathsf{case}\ l\ \mathsf{of}\ \mathsf{cln}_(r){:}\mathsf{OM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k^*,\tilde{t}\,\rangle \mid (\boldsymbol{\nu} s^*)\,\big(\,\overline{r}\langle s^*,k^*\rangle \mid \mathsf{newO}_{\mathbb{O}}\langle\, s^*,\tilde{t}\,\rangle\,\big)\ ;\\
\qquad\quad \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\quad \mathsf{upd}_j_(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\quad \mathsf{inv}_j_(\tilde{x},r){:}\mathsf{OM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k^*,\tilde{t}\,\rangle \mid \overline{t_j}\langle s,\tilde{x},r,k^*\rangle\ ;\\
\qquad\quad \mathsf{sur}_(r){:}\mathsf{OM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k^*,\tilde{t}\,\rangle \mid [\![\, s.\mathsf{alias}\langle s.\mathsf{clone}\rangle\,]\!]^{k^*}_r\ ;\\
\qquad\quad \mathsf{png}_(r){:}\mathsf{OM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k^*,\tilde{t}\,\rangle \mid [\![\, s\,]\!]^{k^*}_r\\
\mathsf{elif}\ [k{=}k_{\mathrm{e}}]\ \mathsf{then}\\
\quad \mathsf{OM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k^*,\tilde{t}\,\rangle \ \big|\ \mathsf{case}\ l\ \mathsf{of}\ \mathsf{cln}_(r){:}m_{\mathrm{i}}(k).\overline{m_{\mathrm{e}}}\ ;\\
\qquad\qquad \mathsf{ali}_(s_{\mathrm{a}},r){:}m_{\mathrm{i}}(k).\overline{m_{\mathrm{e}}}\ ;\\
\qquad\qquad \mathsf{upd}_j_(t',r){:}m_{\mathrm{i}}(k).\overline{m_{\mathrm{e}}}\ ;\\
\qquad\qquad \mathsf{inv}_j_(\tilde{x},r){:}\mathsf{CM}^{r^*\succ r}_{\widetilde{m}}[\,\overline{t_j}\langle s,\tilde{x},r^*,k^*\rangle\,]\ ;\\
\qquad\qquad \mathsf{sur}_(r){:}\mathsf{CM}^{r^*\succ r}_{\widetilde{m}}[\,[\![\, s.\mathsf{alias}\langle s.\mathsf{clone}\rangle\,]\!]^{k^*}_{r^*}\,]\ ;\\
\qquad\qquad \mathsf{png}_(r){:}\mathsf{CM}^{r^*\succ r}_{\widetilde{m}}[\,[\![\, s\,]\!]^{k^*}_{r^*}\,]\\
\mathsf{else}\mathsf{OM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k_{\mathrm{i}},\tilde{t}\,\rangle \ \big|\ m_{\mathrm{e}}.(\ \overline{s}\langle l,k_{\mathrm{e}}\rangle\ \mid\ \overline{m_{\mathrm{i}}}k\ )\Big)
\end{array}$$

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

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

alisation, 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 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 (cf. Lemma A.5.3 for the case of object managers).

Serving external requests $[k{=}k_e]$

Serialisation 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 protection-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 $\overline{m_i}k$ and releasing $\overline{m_e}$. Note that this implies that the sender of the current request is blocked, because it is waiting on the just discarded result channel.

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 $\mathsf{CM}_{\tilde{m}}^{r^* \succ r}[\cdot]$: 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 requests 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 own 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 pre-processed requests that arrive at an alias manager (in state AM^a) will be simply forwarded (in state AM^s) without modification of the current-self k (obtained by consuming $\overline{m_i}k$) to the aliasing target s_a. Finally, when releasing $\overline{m_e}$ again, the manager will go back to state AM^f. Note that the

mutex-protocol for alias managers (which we called *touch-and-go* in [NHKM00]) never prevents external requests from being forwarded; we formalise this important property in Lemma A.5.3. However, as argued earlier the mutex-protocol is necessary for any alias manager in order to adequately treat client requests that were pre-processed by their former object managers.

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

No serialisation 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 $\tilde{t}$, 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.

Nonces (νk^*)

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 7.1, 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.

Another way to ensure the desired serialisation property is to use another type system layer in $\mathrm{L}\pi$ limiting to one the number of occurrences of each key in object position (so it would be different from the known linear type systems, e.g., [KPT96]). However, we prefer our solution since it works in a larger class of contexts, and since we do not need to develop another notion of typed equivalence.

Aliases are forwarders!

We recapitulate the key point of the presented semantics: the definition of alias managers makes explicit that all requests, except for internal aliasing and cloning, are forwarded without any change of the values they carry (cf. Lemma 7.1.3 for the case where the alias manager is free, which precludes the existence of internal requests). The mutex-game for external requests, added for compatibility with object managers, only adds harmless computation steps that slow down the computation without preventing the forwarding (cf. Lemma A.5.3).

Example translations and their behaviour

As a first example, let us illustrate the access to an Øjeblik variable by calling an operation on it.

$$
\begin{aligned}
[\![\, s.\mathsf{clone} \,]\!]_p^k &= (\boldsymbol{\nu} q)\left(\, [\![\, s \,]\!]_q^k \mid q(y,k') \,.\, \overline{y}\langle \mathsf{clone_}p, k' \rangle \,\right) \\
&= (\boldsymbol{\nu} q)\left(\, \underline{\overline{q}\langle s,k \rangle} \mid \underline{q(y,k')} \,.\, \overline{y}\langle \mathsf{clone_}p, k' \rangle \,\right) \\
&\rightarrow\equiv \overline{s}\langle \mathsf{clone_}p, k \rangle
\end{aligned}
$$

Like for clone requests, corresponding transitions are carried out whenever we access a variable: after one step, a respective request at the π-calculus level is sent on the variable to some unique manager that is guaranteed by the encoding to wait at s for requests (cf. Lemma 5.2.1). Since the channel q occurs and is used exactly once for communicating a result, we remove it using structural congruence immediately after the communication. As in this example, we depict by underlining the relevant parts of the terms for enabling the next step.

As a second example, let us look at the (five) transitions that take place when a method is called externally on an object record. In the following term, the method body b will not be used, because we are, for now, not interested to see any further steps taken.

$$
\begin{aligned}
& [\![\, [\mathrm{l}{=}\varsigma(s)b].\mathrm{l} \,]\!]_p^k \\
\stackrel{(0.a)}{=}\; & (\boldsymbol{\nu} q)\left(\, \underline{[\![\, [\mathrm{l}{=}\varsigma(s)b] \,]\!]_q^k} \mid q(y,k') \,.\, \overline{y}\langle \mathsf{inv_}p, k' \rangle \,\right) \\
\stackrel{(0.b)}{=}\; & (\boldsymbol{\nu} q)\left(\, (\boldsymbol{\nu} s\tilde{t})\Big(\, \underline{\overline{q}\langle s,k \rangle} \mid \mathsf{newO}_{\mathbb{O}}\langle\, s,t \,\rangle \mid \underbrace{!\, t(s,r,k').[\![\, b \,]\!]_r^{k'}}_{T\langle\, t,b \,\rangle} \,\Big) \mid \underline{q(y,k')} \,.\, \overline{y}\langle \mathsf{inv_}p, k' \rangle \,\right) \\
\stackrel{(1)}{\rightarrow\equiv}\; & (\boldsymbol{\nu} s\tilde{t})\Big(\, \overline{s}\langle \mathsf{inv_}p, k \rangle \,.\, \mid \underline{\mathsf{newO}_{\mathbb{O}}\langle\, s,t \,\rangle} \mid T\langle\, t,b \,\rangle \,\Big) \\
\stackrel{(1.a)}{=}\; & (\boldsymbol{\nu} s\tilde{t})\Big(\, \underline{\overline{s}\langle \mathsf{inv_}p, k \rangle} \mid (\boldsymbol{\nu} \widetilde{m}\tilde{k})\Big(\, \overline{m_{\mathrm{e}}} \mid \underline{\mathsf{OM}_{\mathbb{O}}\langle\, s,\widetilde{m},\tilde{k},t \,\rangle} \,\Big) \mid T\langle\, t,b \,\rangle \,\Big) \\
\stackrel{(2)}{\rightarrow\equiv}\; & (\boldsymbol{\nu} s\tilde{t})(\boldsymbol{\nu} \widetilde{m}\tilde{k}k^*)\Big(\, \underline{m_{\mathrm{e}}} \,.\, (\overline{s}\langle \mathsf{inv_}p, k_{\mathrm{e}} \rangle \mid \overline{m_{\mathrm{i}}}\langle k \rangle) \mid \underline{\overline{m_{\mathrm{e}}}} \mid \mathsf{OM}_{\mathbb{O}}\langle\, s,\widetilde{m},\tilde{k},t \,\rangle \mid T\langle\, t,b \,\rangle \,\Big) \\
\stackrel{(3)}{\rightarrow\equiv}\; & (\boldsymbol{\nu} s\tilde{t})(\boldsymbol{\nu} \widetilde{m}\tilde{k})\Big(\, \underline{\overline{s}\langle \mathsf{inv_}p, k_{\mathrm{e}} \rangle} \mid \overline{m_{\mathrm{i}}}\langle k \rangle \mid \underline{\mathsf{OM}_{\mathbb{O}}\langle\, s,\widetilde{m},\tilde{k},t \,\rangle} \mid T\langle\, t,b \,\rangle \,\Big) \\
\stackrel{(4)}{\rightarrow\equiv}\; & (\boldsymbol{\nu} s\tilde{t})(\boldsymbol{\nu} \widetilde{m}\tilde{k}k^*)\Big(\, \mathsf{CM}_{\widetilde{m}}^{r^*\succ p}[\, \underline{\overline{t}\langle s,r^*,k^* \rangle} \,] \mid \overline{m_{\mathrm{i}}}\langle k \rangle \mid \mathsf{OM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k^*,t \,\rangle \mid \underline{T\langle\, t,b \,\rangle} \,\Big) \\
\stackrel{(5)}{\rightarrow\equiv}\; & (\boldsymbol{\nu} s\tilde{t})(\boldsymbol{\nu} \widetilde{m}\tilde{k}\underline{k^*})\Big(\, \mathsf{CM}_{\widetilde{m}}^{r^*\succ p}[\, [\![\, b \,]\!]_{r^*}^{k^*} \,] \mid \overline{m_{\mathrm{i}}}\langle k \rangle \mid \mathsf{OM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k^*,t \,\rangle \mid T\langle\, t,b \,\rangle \,\Big) \\
\stackrel{(5.a)}{\equiv}\; & (\boldsymbol{\nu} s\tilde{t})(\boldsymbol{\nu} \widetilde{m}\tilde{k})\Big(\, \mathsf{CM}_{\widetilde{m}}^{r^*\succ p}[\, [\![\, b \,]\!]_{r^*}^{k_{\mathrm{i}}} \,] \mid \overline{m_{\mathrm{i}}}\langle k \rangle \mid \mathsf{OM}_{\mathbb{O}}\langle\, s,\widetilde{m},\tilde{k},t \,\rangle \mid T\langle\, t,b \,\rangle \,\Big)
\end{aligned}
$$

The steps $(0.a)$ and $(0.b)$ simply apply the translation, first for method invocation, then for object records, which provides the setup of an object process, together with its method bodies, and the announcement of its access channel via the private "evaluation channel" q. In step (1), the invocation is prepared by receiving on q the access channel s of the object. Step $(1.a)$ simply expands the definition in order to provide the receiver on the access channel s. In step (2), the access channel s is used for the first time. By this step, we enter the object with a key that is different from both the internal and external key of the object itself, so the request is pre-processed (i.e., prepared to be resent with the external key k_{e}) and blocked by a reception on the (external) mutex m_{e}. Also in step (2), a new key k^* is generated, since it is not used at all (the restarted manager does not use it either), we remove it again in step (3) where the external mutex m_{e} is consumed. Now, the pre-processed request is free to enter the object manager with the proper external key in step (4), which creates a call manager for the call of the method that is represented by the trigger name t, and also a fresh key k^*, which is now used in the restarted manager instead of the former k_{i}. This is a good moment to observe the serialisation property that the only occurrence of the fresh key outside the manager itself is passed on to the intended method body. No other term may dispose of the current internal key of the manager. Note also that the former key k is now "parked" on the internal mutex m_{i}. Step (5) finally starts a copy of the translated method body b in the proper context of the internal key of the manager and the result channel that will report (if at all) to the call manager. For simplicity, since we only regard reductions on translations of Øjeblik terms, and not transitions within arbitrary contexts, we can

always safely rename k^* to the former k_i. Note that all of the above transitions were deterministic due to the simplicity of the term: only one thread exists in it.

As a third example, let us put the above two together and see how an internal clone request in place of the above abstract term b is served.

$$
\begin{aligned}
& [\![\, [\mathrm{l}{=}\varsigma(s)s.\mathsf{clone}\,].\mathrm{l}\,]\!]_p^k \\
\overset{(1-5)}{\to\equiv} & (\boldsymbol{\nu} st\widetilde{m}\tilde{k})\Big(\mathsf{CM}_{\widetilde{m}}^{r^*\succ p}[\, \underline{[\![\, s.\mathsf{clone}\,]\!]_{r^*}^{k_\mathrm{i}}}\,] \mid \overline{m_\mathrm{i}}\langle k\rangle \mid \mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{m}, \tilde{k}, t\,\rangle \mid T\langle\, t, s.\mathsf{clone}\,\rangle \Big) \\
\overset{(6)}{\to\equiv} & (\boldsymbol{\nu} st\widetilde{m}\tilde{k})\Big(\mathsf{CM}_{\widetilde{m}}^{r^*\succ p}[\, \underline{\overline{s}\langle \mathsf{cln_}\langle r^*\rangle, k_\mathrm{i}\rangle}\,] \mid \overline{m_\mathrm{i}}\langle k\rangle \mid \underline{\mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{m}, \tilde{k}, t\,\rangle} \mid T\langle\, t, s.\mathsf{clone}\,\rangle \Big) \\
\overset{(7)}{\to\equiv} & (\boldsymbol{\nu} st\widetilde{m}\tilde{k}k^*)\ \Big(\underline{\mathsf{CM}_{\widetilde{m}}^{r^*\succ p}}[\, (\boldsymbol{\nu} s^*)(\overline{r^*}\langle s^*, k^*\rangle \mid \mathsf{newO}_{\mathbb{O}}\langle\, s^*, t\,\rangle)\,] \\
& \qquad \mid \overline{m_\mathrm{i}}\langle k\rangle \mid \mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{m}, k_\mathrm{e}, k^*, t\,\rangle \mid T\langle\, t, s.\mathsf{clone}\,\rangle \Big) \\
\overset{(7.a)}{\equiv} & (\boldsymbol{\nu} st\widetilde{m}\tilde{k})\ \Big((\boldsymbol{\nu} r^*)\ \big(\quad (\boldsymbol{\nu} s^*)(\underline{\overline{r^*}\langle s^*, k_\mathrm{i}\rangle} \mid \mathsf{newO}_{\mathbb{O}}\langle\, s^*, t\,\rangle) \\
& \qquad\qquad \mid \quad \underline{r^*(y, k')}.m_\mathrm{i}(k'').(\overline{p}\langle y, k''\rangle \mid \overline{m_\mathrm{e}})\big) \\
& \qquad \mid \overline{m_\mathrm{i}}\langle k\rangle \mid \mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{m}, \tilde{k}, t\,\rangle \mid T\langle\, t, s.\mathsf{clone}\,\rangle \Big) \\
\overset{(8)}{\to\equiv} & (\boldsymbol{\nu} ss^*t\widetilde{m}\tilde{k})\ \Big(\mathsf{newO}_{\mathbb{O}}\langle\, s^*, t\,\rangle \mid \underline{m_\mathrm{i}(k'')}.(\overline{p}\langle s^*, k''\rangle \mid \overline{m_\mathrm{e}}) \\
& \qquad \mid \underline{\overline{m_\mathrm{i}}\langle k\rangle} \mid \mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{m}, \tilde{k}, t\,\rangle \mid T\langle\, t, s.\mathsf{clone}\,\rangle \Big) \\
\overset{(9)}{\to\equiv} & (\boldsymbol{\nu} ss^*t\underline{\widetilde{m}\tilde{k}})\Big(\mathsf{newO}_{\mathbb{O}}\langle\, s^*, t\,\rangle \mid \overline{p}\langle s^*, k\rangle \mid \underline{\overline{m_\mathrm{e}} \mid \mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{m}, \tilde{k}, t\,\rangle} \mid T\langle\, t, s.\mathsf{clone}\,\rangle \Big) \\
\overset{(9.a)}{\equiv} & (\boldsymbol{\nu} ss^*t)\Big(\mathsf{newO}_{\mathbb{O}}\langle\, s^*, t\,\rangle \mid \underline{\overline{p}\langle s^*, k\rangle} \mid \mathsf{newO}_{\mathbb{O}}\langle\, s, t\,\rangle \mid T\langle\, t, s.\mathsf{clone}\,\rangle \Big)
\end{aligned}
$$

The first 5 steps represent the access to the method itself. Step (6) evaluates the variable s and sends the clone request. Step (7) does the cloning by the creation of a new access channel s^* and the installation of an object at it that serves the same method through t. Step $(7.a)$ expands the call manager to exhibit that it is waiting on the result channel r^*, so it consumes the result in step (8). Afterwards, the key k that was parked on the internal mutex m_i is grabbed again and, by that, the external mutex m_e becomes free, and the cloning result s^* is forwarded to the overall result channel p. The last step $(9.a)$ is only meant to highlight that the former object on which the cloning operation was carried out, survives the operation without effect on itself.

5 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, i.e. that there is always only one manager available to receive requests. We also describe how object (and alias) managers evolve.

5.1 The Lπ^+-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 5.1.1; (ii) Øjeblik's type system itself is provided with some more formal underpinning, as demonstrated in Proposition 5.1.2; (iii) we may employ typeful reasoning about terms. 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 macros to denote (i) the type $\mathbf{R}(T)$ of a result channel, which can be used to retrieve results of type T, together with the current key; (ii) the list $\mathbf{M}(B_1 \mathinner{..} B_n \to B)$ of types for methods, which is the translation of the types of the arguments $[\![\, B_1\,]\!] \mathinner{..} [\![\, B_n\,]\!]$, and the type of the result channel $\mathbf{R}([\![\, B\,]\!])$. The most critical part of the translation is the proper

Table 8: Translation of Øjeblik-types

$$\mathbf{R}(T) \stackrel{\text{def}}{=} \mathbf{C}(T, \mathbf{K})$$

$$\mathbf{M}(B_1 \mathinner{..} B_n \to B) \stackrel{\text{def}}{=} [\![B_1]\!] \mathinner{..} [\![B_n]\!], \mathbf{R}([\![B]\!])$$

$$[\![\mu X [\mathsf{l}_j{:}B_{1_j} \mathinner{..} B_{n_j} \to B_j]_{j \in J}]\!] \stackrel{\text{def}}{=}$$

$$\mu X.\, \mathbf{C}(\begin{bmatrix} \mathsf{cln} : & \mathbf{R}(X) \\ \mathsf{ali} : & \langle X, \mathbf{R}(X) \rangle \\ \mathsf{upd}_j{:} & \langle \mathbf{C}(X, \mathbf{M}(B_{1_j} \mathinner{..} B_{n_j} \to B_j), \mathbf{K}), \mathbf{R}(X) \rangle \\ \mathsf{inv}_j : & \langle \mathbf{M}(B_{1_j} \mathinner{..} B_{n_j} \to B_j) \rangle \\ \mathsf{sur} : & \mathbf{R}(X) \\ \mathsf{png} : & \mathbf{R}(X) \end{bmatrix}_{j \in J}, \mathbf{K})$$

$$[\![X]\!] \stackrel{\text{def}}{=} X$$

$$[\![\mathrm{Thr}(A)]\!] \stackrel{\text{def}}{=} \mathbf{C}(\mathbf{R}([\![A]\!]), \mathbf{K})$$

$$[\![\Gamma, x{:}A]\!] \stackrel{\text{def}}{=} [\![\Gamma]\!], x{:}[\![A]\!]$$

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 Lπ, requiring us to use received names only for output, 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 4.

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

Theorem 5.1.1 (Type Soundness) *Let a be an Øjeblik term, let Γ be a type environment, and let A be a type. Then $\Gamma \vdash a{:}A$ if and only if $[\![\Gamma]\!], p{:}\mathbf{R}([\![A]\!]), k{:}\mathbf{K} \vdash [\![a]\!]_p^k$ 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.

Proposition 5.1.2 (Subject Reduction) *Let $\Gamma \vdash a{:}A$.
If $[\![a]\!]_p^k \Rightarrow Q$, then $[\![\Gamma]\!], p{:}\mathbf{R}([\![A]\!]), k{:}\mathbf{K} \vdash Q$.*

5.2 Properties of object managers

A crucial property in object-oriented languages is the uniqueness of objects. The Lπ constraint on the output capability guarantees this property.

Lemma 5.2.1 (Uniqueness of objects) *Let a be an Øjeblik term. If $[\![a]\!]_p^k \Rightarrow Z$ with*

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

then $s \in \tilde{z}$ 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 in Table 7 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 5.2.2).

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 $\tilde{v}{:=}v_1..v_h$) are pre-processed by the object manager $\mathsf{OM}_{\mathbb{O}}\langle\, s, m_e, m_i, k_e, k_i, \tilde{t}\,\rangle$, so $k_e{\neq}k_j{\neq}k_i$ for all $j \in 1..h$. Then:

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

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 $\mathsf{OM}_{\mathbb{O}}\langle\, s, m_e, m_i, k_e, k_i, \tilde{t}\,\rangle$, 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 $\tilde{n} := m_e, m_i, k_e$ 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.

Lemma 5.2.2 (Object and alias manager evolution) *Let a be an Øjeblik term. If $[\![\, a\,]\!]_p^k \Rightarrow Z$ and $Z = E[\mathsf{OM}_{\mathbb{O}}\langle\, s,\dots\rangle]$ or $Z = E[\mathsf{AM}_{\mathbb{O}}\langle\, s,\dots\rangle]$, with $E[\cdot]$ static, then—without α-converting the name s—*

$$\begin{array}{lrcl} \textit{either} & Z & \equiv & \widehat{E}[\,(\boldsymbol{\nu}\tilde{n})\,(\,M_S \mid \mathsf{PP}_{\mathbb{O}}\langle\, s,\tilde{n},\tilde{v}\,\rangle \mid \mathsf{OM}_{\mathbb{O}}\langle\, s,\tilde{n},k_i,\tilde{t}\,\rangle\,)\,] \\ \textit{or} & Z & \equiv & \widehat{E}[\,(\boldsymbol{\nu}\tilde{n})\,(\,M_S \mid \mathsf{PP}_{\mathbb{O}}\langle\, s,\tilde{n},\tilde{v}\,\rangle \mid \mathsf{AM}_{\mathbb{O}}\langle\, s,\tilde{n},k_i,s_a\,\rangle\,)\,] \end{array}$$

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

S	M_S
$\mathsf{OM}^{\mathrm{f}}, \mathsf{AM}^{\mathrm{f}}$	$\overline{m_e}$
$\mathsf{OM}^{\mathrm{a}}, \mathsf{AM}^{\mathrm{a}}$	$\overline{m_i}k \mid \overline{s}\langle l,k_e\rangle$
OM^{n}	$\overline{m_i}k \mid m_i(k).\overline{m_e}$
$\mathsf{OM}^{\mathrm{s}}, \mathsf{AM}^{\mathrm{c}}$	$\overline{m_i}k \mid r^*(y,k').m_i(k'').(\overline{r}\langle y,k''\rangle \mid \overline{m_e}\,)$
AM^{s}	$\overline{m_i}k \mid m_e.(\overline{s}\langle l,k_e\rangle \mid \overline{m_i}k)$
$\mathsf{OM}^{\mathrm{i}}, \mathsf{AM}^{\mathrm{i}}$	$\overline{m_i}k \mid m_i(k'').(\overline{r}\langle y,k''\rangle \mid \overline{m_e}\,)$.

with S denoting the states of OM *resp.* AM *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 of Z is in one of the ten described states. Details can be found in Appendix A.3. □

In the following two observations, we outline two special cases of Lemma 5.2.2: *free* managers in states OM^{f} and AM^{f}, and *committing* object managers, which are 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. In addition, our semantics guarantees that in such cases, there will then be no internal requests available such that the statement of Lemma 5.2.2 can be strengthened for this case: If $[\![\, a\,]\!]_p^k \Rightarrow Z$ with $Z = E_1[\mathsf{OM}_{\mathbb{O}}\langle\, s,\widetilde{m},\tilde{k},\tilde{t}\,\rangle]$ and

$Z = E_2[\overline{m_e}]$, for some static context $E_1[\cdot]$, and $E_2[\cdot]$, then—without α-converting the name s—there is a static context $\widehat{E}[\cdot]$ such that:

$$Z \equiv \widehat{E}[\,(\boldsymbol{\nu}\tilde{n})\,(\,\overline{m_e} \mid \mathsf{PP}_{\mathbb{O}}\langle\, s,\tilde{n},\tilde{v}\,\rangle \mid (\boldsymbol{\nu}k_i)\,\mathsf{OM}_{\mathbb{O}}\langle\, s,\tilde{n},\tilde{t}\,\rangle\,)\,] \tag{6}$$

This statement basically means that a manager, when reaching state OM^f (from state OM^i), has acquired again full control over its internal key k_i. We do not need a formal proof of this, but we will see in Lemma 7.1.2 an example of such a state change for the case of termination of serving an external surrogation. The intuitive argument is based on serialisation and the fact that at any moment in time there is most one internal request per manager: internal requests may exist as long as some running method of the manager has not yet terminated; only on termination the call manager of the running method release the external mutex. Thus, the availability of the external mutex implies the absence of internal requests, so the scope of the internal mutex can be tightened to the object manager. A similar argument holds for the case that state AM^f is reached from state AM^i. We abbreviate the shape of *free object managers* (the processes sitting in the hole of $\widehat{E}[\cdot]$ of equation (6)) and correspondingly *free alias managers*, again with $\tilde{n} := m_e, m_i, k_e$, as:

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

where the keys mentioned in $\tilde{v}$ of $\mathsf{PP}_{\mathbb{O}}\langle\ldots\rangle$ are different from k_e. Notice that $\mathsf{newO}_{\mathbb{O}}\langle\, s,\tilde{t}\,\rangle \equiv \mathsf{freeO}_{\mathbb{O}}\langle\, s,\tilde{t},\emptyset\,\rangle$, and analogously for $\mathsf{newA}_{\mathbb{O}}\langle\ldots\rangle$.

Observation 4: An object manager is ready to commit*, if it may consume a pre-processed request, i.e., a request that has already grabbed* m_e. An object manager *commits* when it evolves from state OM^a into state OM^s, which according to Lemma 5.2.2 happens when the manager consumes a request $\overline{s}\langle l,k_e\rangle$. The following lemma will be used in the proof of Theorem 7.2.1. It states that the two processes $\mathsf{pngO}_{\mathbb{O}}\langle\, s,r,k,\tilde{t},\tilde{v}\,\rangle$ and $\mathsf{surO}_{\mathbb{O}}\langle\, s,r,k,\tilde{t},\tilde{v}\,\rangle$, as defined by

$$\begin{aligned}Srv[\cdot] &\stackrel{\text{def}}{=} (\boldsymbol{\nu}\tilde{n})\big(\overline{m_i}k \,\big|\, \mathsf{PP}_{\mathbb{O}}\langle\, s,\tilde{n},\tilde{v}\,\rangle \,\big|\, [\cdot]\,\big)\\ \mathsf{pngO}_{\mathbb{O}}\langle\, s,r,k,\tilde{t},\tilde{v}\,\rangle &\stackrel{\text{def}}{=} Srv[(\boldsymbol{\nu}k^*)(\,\mathsf{OM}_{\mathbb{O}}\langle\, s,\tilde{n},k^*,\tilde{t}\,\rangle \mid \mathsf{CM}_{\tilde{m}}^{r^*\succ r}[\,[\![\, s\,]\!]_{r^*}^{k^*}\,]\,)]\\ \mathsf{surO}_{\mathbb{O}}\langle\, s,r,k,\tilde{t},\tilde{v}\,\rangle &\stackrel{\text{def}}{=} Srv[(\boldsymbol{\nu}k^*)(\,\mathsf{OM}_{\mathbb{O}}\langle\, s,\tilde{n},k^*,\tilde{t}\,\rangle \mid \mathsf{CM}_{\tilde{m}}^{r^*\succ r}[\,[\![\, s.\mathsf{alias}\langle s.\mathsf{clone}\rangle\,]\!]_{r^*}^{k^*}\,]\,)]\end{aligned}$$

are obtained when committing to png- and sur-requests, respectively.

Lemma 5.2.3 (Committing manager) *Let a be an Øjeblik term, let* $[\![\, a\,]\!]_p^k \Rightarrow Z$ *and* $Z \equiv E[\,\overline{s}\langle l,k_e\rangle \mid \mathsf{OM}_{\mathbb{O}}\langle\, s,\tilde{n},k_i,\tilde{t}\,\rangle\,]$ *with* $E[\cdot]$ *static,* $\tilde{n} = m_e, m_i, k_e$*. Then:*

1. *If* $l = \mathsf{png_}r$*, then* $Z \xrightarrow{\tau}\equiv \widehat{E}[\,\mathsf{pngO}_{\mathbb{O}}\langle\, s,r,k,\tilde{t},\tilde{v}\,\rangle\,]$
   for some static context $\widehat{E}[\cdot]$*, some key* k*, some set* $\tilde{v}$ *of pre-processed requests.*
2. *If* $l = \mathsf{sur_}r$*, then* $Z \xrightarrow{\tau}\equiv \widehat{E}[\,\mathsf{surO}_{\mathbb{O}}\langle\, s,r,k,\tilde{t},\tilde{v}\,\rangle\,]$
   for some static context $\widehat{E}[\cdot]$*, some key* k*, some set* $\tilde{v}$ *of pre-processed requests.*

PROOF. We only consider the case $l = \mathsf{sur_}r$. The case when $l = \mathsf{png_}r$ is similar. According to Lemma 5.2.2, the only way to have $Z \equiv E[\,\overline{s}\langle \mathsf{sur_}r,k_e\rangle \mid \mathsf{OM}_{\mathbb{O}}\langle\, s,\tilde{n},k_i,\tilde{t}\,\rangle\,]$ is in state OM^a with $Z \equiv \widehat{E}[\,(\boldsymbol{\nu}\tilde{n})\,(\,\overline{s}\langle \mathsf{sur_}r,k_e\rangle \mid \mathsf{PP}_{\mathbb{O}}\langle\, s,\tilde{n},\tilde{v}\,\rangle \mid \mathsf{OM}_{\mathbb{O}}\langle\, s,\tilde{n},k_i,\tilde{t}\,\rangle\,)\,]$. By inspection of the encoding of managers in Table 7, we see that consumption of the request $\mathsf{sur_}r$ results in the creation of a new internal key k^*, which is used in the restarted object manager, and of a call manager $\mathsf{CM}_{\tilde{m}}^{r^*\succ r}[[\![\, s.\mathsf{alias}\langle s.\mathsf{clone}\rangle\,]\!]_{r^*}^{k^*}]$. □

6 Towards the formalisation of safe surrogation

In [NHKM00], we motivated an equation on Øjeblik terms to model the safety of object surrogation. In Subsection 6.1, 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 6.3 will allow us to precisely formalise this restriction.

6.1 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 with or without surrogation*, in all possible contexts. The following equation is a first attempt to model this property:

$$a \doteq a.\mathsf{surrogate} \tag{7}$$

The simplest case of Equation 7 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 adds one unconditional step after reducing a) and that the notion of equivalence takes all Øjeblik contexts into account, Equation 7 can be reduced to the problem of surrogation on variables:

$$x \doteq x.\mathsf{surrogate} \tag{8}$$

However, there is an inherent problem with Equation 8, which is exhibited by the following context that creates a self-alias via method call:

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

In $C[x]$ the evaluation of x returns immediately, while in $C[x.\mathsf{surrogate}]$, the request $x.\mathsf{surrogate}$ is never served because it travels into a loop along the self alias chain $x \gg x$. As a consequence, the term $C[x]$ terminates successfully whereas $C[x.\mathsf{surrogate}]$ does not (see Definition 6.1.1). The problem in Equation 8 is that we do not check whether the "object before surrogation" is actually reachable. This can be easily done as follows

$$x.\mathsf{ping} \doteq x.\mathsf{surrogate} \tag{9}$$

The equation 9 detects cyclic chains by means of the ping-request which travels to the endpoint of the alias chain possibly starting at x. Now, both terms $C[x.\mathsf{ping}]$ and $C[x.\mathsf{surrogate}]$ loop.

In the remainder of the paper, Equation 9 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 [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 6.1.1 (Convergence) *Given an Øjeblik term a, we write $a\Downarrow$ if $[\![\,a\,]\!]_p^k \Downarrow_p$.*

In the sequel, given an Øjeblik-term a we say that a context $C[\cdot]$ is *closing* for a if $C[a]\Downarrow$ is a closed term.

Definition 6.1.2 (Behavioural equivalence) *Two Øjeblik terms a and b are* behaviourally equivalent*, written $a \doteq b$, if*

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

for all closing Øjeblik contexts $C[\cdot]$.

Table 9: Øjeblik contexts

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

6.2 On the problem 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 *internal* surrogation might lead to a misuse, by intention or by accident, of the newly created references. For example, let us look at the context

$$C[\cdot] \stackrel{\mathrm{def}}{=} [\mathrm{l}{=}\varsigma(s)[\cdot].\mathsf{clone}\,].\mathrm{l}$$

which performs a cloning operation on the hole inside a method. When we plug the term $s.\mathsf{surrogate}$ into the hole, then the cloning operation will be carried out on the result of the surrogate operation. At least, it will be tried, but since the surrogate operation returns a reference to the just created copy, the clone operation will block because it is external. However, if we instead plug $s.\mathsf{ping}$ into the hole, then the cloning operation will be performed without problems: here, it is internal due to ping in this case returning just the current self of its surrounding method. Adapting and extending the example reductions at the end of Section 4, the reader may convince herself that both

$$C[s.\mathsf{surrogate}]\not\Downarrow \qquad \text{and} \qquad C[s.\mathsf{ping}]\Downarrow$$

hold. In both cases, there are only deterministic reductions, which in the case of surrogate lead to a final state in which the result channel for the evaluation of $C[s.\mathsf{surrogate}]$ is captured underneath the input of the call manager's private result channel, which in turn will never be served, because there will never be a matching message available, as the clone request is rejected by the object manager of the surrogation target. The above attempt to perform $s.\mathsf{surrogate.clone}$ inside a method with self parameter s must be regarded as an intentional misuse of (the result of) the surrogate operation: it is the method itself who performs the surrogate operation, so it (i.e., its programmer) knows for sure that it returns a reference o which is different from its self s, so it should be aware of the fact that cloning o will inevitably block. (For other examples of misused internal surrogation, the reader may consult [NHKM00].) However, as we have seen in Equation (5) on page 14, internal surrogation can not only be syntactically addressed to the current self, but it may also appear without intention as, for example, in the adaptation of the aforementioned accidentally internal call:

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

As for the previous example, we get that

$$C[z.\mathsf{surrogate}]\not\Downarrow \qquad \text{and} \qquad C[z.\mathsf{ping}]\Downarrow$$

hold, but this time the internal surrogation was not intended by the code of the method itself. In summary, we conclude that internal surrogation poses inevitable problems to the transparency of surrogation as expressed by our safety equation. [2]

[2] The situation is analogous to the problem of division by zero: of course, knowing that $x{=}0$, a programmer should never use division by x, and such programs are definitely seen as *programming errors*, and not as a fault of

In [NHKM00], we conjectured that *external surrogation is safe*. Recall that it is syntactically undecidable [Car95] whether a call $x.\mathsf{operate}$ occurring in $C[x.\mathsf{operate}]$ is going to be performed internally or externally; the only way to find this out is by a run-time test, e.g., by using our π-calculus semantics. This is precisely what we do in the next subsection: we use our π-calculus semantics to 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.

6.3 Formalising the absence of internal surrogation

In our semantics, the computation $[\![\, a \,]\!]_p^k \Rightarrow Z$ of an Øjeblik term a yields a self-inflicted sur-request if $Z \equiv E[\, \overline{s}\langle \mathsf{sur}_r, k\rangle \mid \mathsf{OM}_{\mathbb{O}}\langle\, s, \widetilde{m}, k_{\mathrm{e}}, k_{\mathrm{i}}, \tilde{t}\,\rangle\,]$, 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". Other sur-requests would arise from the context itself and would be the same on both sides of the equation. 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 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, $\mathsf{OM}^\star$, $\mathsf{AM}^\star$, $\mathsf{pngO}^\star$ 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 6.3.1 *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 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. □

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

the language semantics, because the misuse is intentional. In contrast, if a program receives the binding for x from some other module, then it should be somehow guaranteed that x will never be 0, because the programmer cannot enforce this property autonomously.

Table 10: Translational semantics — Additional tagged clauses

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

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

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

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

Definition 6.3.2 (External Contexts) *Let x be a variable and $C[\cdot]$ an untagged Øjeblik context. Then, $C[\cdot]$ is called* external *for x.*surrogate*, if whenever*

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

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

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

7 On the safety of surrogation

In this section, we prove that

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

under the assumption that $C[\cdot]$ will never lead to internal occurrences of x.surrogate. In Subsection 7.1, we study the behaviour of the committed object managers $\mathsf{pngO}_{\mathbb{O}}\langle\, s, \ldots\,\rangle$ and $\mathsf{surO}_{\mathbb{O}}\langle\, s, \ldots\,\rangle$, as defined at the end of Subsection 5.2, and prove them algebraically to be barbed Γ-equivalent. In Subsection 7.2, 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 7.3, we give a static type system that guarantees that surrogations will always be external.

7.1 On committing external surrogations

By Lemma 5.2.3, when an object manager commits to either a png or a sur request, we get the processes $\mathsf{pngO}_{\mathbb{O}}\langle\, s, r, k, \tilde{t}, \tilde{v}\,\rangle$ or $\mathsf{surO}_{\mathbb{O}}\langle\, s, r, k, \tilde{t}, \tilde{v}\,\rangle$, respectively. In the following we show that they are related by typed barbed equivalence $\simeq_{\Gamma;s}$ (Definition 2.2.9). Note that the process $\mathsf{surO}_{\mathbb{O}}\langle\, s, r, k, \tilde{t}, \tilde{v}\,\rangle$ may still interact with the environment via the reference s, but it may also interact via the new reference that it is ready to communicate to the environment on the result channel r.

Theorem 7.1.1 *Let $\Gamma \vdash \mathsf{surO}_{\mathbb{O}}\langle\, s, r, k, \tilde{t}, \tilde{v}\,\rangle, \mathsf{pngO}_{\mathbb{O}}\langle\, s, r, k, \tilde{t}, \tilde{v}\,\rangle$. Then:*

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

The proof of Theorem 7.1.1 requires five lemmas whose proofs 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. For the sake of clarity we omit type annotations in restrictions.

Lemma 7.1.2 proves that surrogation results in a *free alias* pointing to a clone of the old object. The proof relies on the nonces (cf. page 22) used in the implementation of both object and alias managers, which control the interference with the environment.

Lemma 7.1.2 *If Γ is a suitable type environment for the processes below, then:*

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

Note that Lemma 7.1.2 precisely represents the requirement (i), as mentioned in the Introduction on page 2, that migration (or surrogation) must not be prevented from proper completion.

Lemma 7.1.3 proves that *free* alias managers, like the one appearing in Lemma 7.1.2, behave as forwarders. It is a crucial point in the proof that the alias manager is *free*, because the lemma would not hold if the alias manager was not in a state implying the absence of self-inflicted requests (we essentially just need the guarantee that the alias manager is not in a serving/completing state AM^c), as it is precisely guaranteed by allowing only external surrogations.

Lemma 7.1.3 (Free alias managers are forwarders) *Let* $\tilde{v} := v_1..v_n$, $v_j{:=}\langle\, l_j, k_j\,\rangle$ *for* $j \in 1..n$. *If* Γ *is a suitable type environment for the processes below, then:*

$$\mathsf{freeA}_{\mathbb{O}}\langle\, s, s^*, \tilde{v}\,\rangle \approx_{\Gamma;s} s \triangleright s^* \Big| \prod_{j\in 1..n} \overline{s^*}v_j.$$

Note that Lemma 7.1.3 also precisely represents the requirement (ii), as mentioned in the Introduction on page 2, that migration (or surrogation) must not be corrupted after completion. Note that without the well-typedness hypothesis the two processes above would have a different behaviour. Now, this lemma allows us to apply the theory of Lπ.

Lemma 7.1.4 uses the forwarder law of Lπ^+ given in Lemma 2.2.14. Note that the proof of Lemma 7.1.4 is not a trivial application of Lemma 2.2.14.

Lemma 7.1.4 *Let* P *be a process and* s *a name. If* $\Gamma \vdash s{:}\mathbf{C}(T)$ *and* 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 7.1.5 proves that pre-processing external requests does not preclude other requests.

Lemma 7.1.5 *Let* $\tilde{v} := v_1..v_n$ *with* $v_j{:=}\langle\, l_j, k_j\,\rangle$ *and* $k_j{\neq}k_{\mathrm{i}}$ *for* $j \in 1..n$. *If* Γ *is a suitable type environment for the processes below, then:*

$$\prod_{j\in 1..n} \overline{s}v_j \;\Big|\; \mathsf{newO}_{\mathbb{O}}\langle\, s, \tilde{t}\,\rangle \approx_{\Gamma;s} \mathsf{freeO}_{\mathbb{O}}\langle\, s, \tilde{t}, \tilde{v}\,\rangle.$$

Lemma 7.1.6 is a technical lemma involving two confluent reductions.

Lemma 7.1.6 *Let* $\tilde{v} := v_1..v_n$ *with* $v_j{:=}\langle\, l_j, k_j\,\rangle$ *and* $k_j{\neq}k_{\mathrm{i}}$ *for* $j \in 1..n$. *If* Γ *is a suitable type environment for the processes below, then:*

$$\mathsf{freeO}_{\mathbb{O}}\langle\, s, \tilde{t}, \tilde{v}\,\rangle \;\Big|\; \overline{r}\langle s, k\rangle \approx_{\Gamma} \mathsf{pngO}_{\mathbb{O}}\langle\, s, r, k, \tilde{t}, \tilde{v}\,\rangle.$$

Proof of Theorem 7.1.1
PROOF. We recall that relations $\approx_\Gamma$ and $\approx_{\Gamma;s}$ imply $\simeq_{\Gamma;s}$. By subsequently applying Lemmas 7.1.2, 7.1.3, 7.1.4, 7.1.5, and 7.1.6 we have:

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

□

7.2 External Surrogation is Safe

Based on the knowledge of Theorem 7.1.1 that the committed object managers $\mathsf{pngO}_{\mathbb{O}}\langle\, s, \dots\rangle$ and $\mathsf{surO}_{\mathbb{O}}\langle\, s, \dots\rangle$ 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 7.2.1 (Safety) *Let* x *be an object variable and* $C[\cdot]$ *a closing (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 6.3.1 our proof obligation is equivalent to:

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

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 (cf. Proposition 5.1.2) 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

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

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 5.2, 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. Given a general process P, we know that whenever $P \xrightarrow{\tau} P'$, then either

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

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

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

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 $\overline{c}v = \overline{c}\langle \mathsf{sur}^\star_q, j\rangle$ (or $\overline{c}v = \overline{c}\langle \mathsf{png}^\star_q, j\rangle$) and $c(x).R = \mathsf{AM}\langle\, c, \widetilde{m}, k_{\mathrm{e}}, k_{\mathrm{i}}, \tilde{t}\,\rangle$, or
- case 1 applies where $\overline{c}v = \overline{c}\langle \mathsf{sur}^\star_q, j\rangle$ (or $\overline{c}v = \overline{c}\langle \mathsf{png}^\star_q, j\rangle$) and $c(x).R = \mathsf{OM}\langle\, c, \widetilde{m}, k_{\mathrm{e}}, k_{\mathrm{i}}, \tilde{t}\,\rangle$ with $k_{\mathrm{i}} \neq j \neq k_{\mathrm{e}}$.

We denote these by $P \rightarrow_{\mathrm{i}} P'$.

The missing case of $\overline{c}v = \overline{c}\langle \mathsf{sur}^\star_q, k\rangle$ and $c(x).R = \mathsf{OM}\langle\, c, \widetilde{m}, k_{\mathrm{e}}, k_{\mathrm{i}}, \tilde{t}\,\rangle$ with $k = k_{\mathrm{i}}$ is excluded by the assumption that $C[\cdot]$ is external for x (c.f. Definition 6.3.2). 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 are able to classify the reduction steps of the given reduction sequence $D[\overline{y}\langle \mathsf{sur}^\star_q, j\rangle] \Rightarrow P\downarrow_p$. Let d be the number of significant steps in this reduction sequence. If $d = 0$, then the sequence contains only insignificant steps and by Lemma 7.2.2 (see below), we conclude. If $d > 0$,

then:

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

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

By (the tagged counterpart of) Lemma 5.2.3 it holds that:

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

for some $\tilde{z}_h$ 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 congruence *and* barbed equivalence.

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

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

where:

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

The insignificant reduction steps $\to_{\rm i}$ exist because of Lemma 7.2.2. The significant reduction steps $Q_{h,n_h} \;\to_{\rm s} Q_h$ are analogous to their counterparts $P_{h,n_h} \;\to_{\rm s} P_h$. Precisely, by (the tagged counterpart of) Lemma 5.2.3, they give rise (up to structural congruence) to a $\mathsf{pngO}^\star$ instead of a $\mathsf{surO}^\star$, that is:

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

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

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

The relations $Q_h \simeq_\Gamma \widehat{Q}_h$ hold by application of (the tagged counterparts of) Theorem 7.1.1 and Lemma 5.2.1, 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

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

Lemma 7.2.2 *Let a be an Øjeblik term possibly containing a tagged request. If $[\![\, a \,]\!]^k_p \Rightarrow R \to_{\rm i} R'$, then $R[^{\mathsf{png}^\star}/_{\mathsf{sur}^\star}] \;\to_{\rm i} R'[^{\mathsf{png}^\star}/_{\mathsf{sur}^\star}]$ and $R[^{\mathsf{sur}^\star}/_{\mathsf{png}^\star}] \;\to_{\rm i} R'[^{\mathsf{sur}^\star}/_{\mathsf{png}^\star}]$.*

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 7.2.1. □

Table 11: Typing Rules Ensuring External Surrogate Operations

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

$$(\text{T-UPD})\quad \frac{\Gamma \vdash_D a{:}A \qquad A = \mu X[\mathrm{l}_j{:}\widetilde{B}_j{\to}B_j]_{j\in J} \qquad \Gamma, s_k{:}A, \tilde{x}_k{:}\widetilde{B}_k\{A/X\} \vdash_A b_k{:}B_k\{A/X\} \qquad k{\in}J}{\Gamma \vdash_D a.\mathrm{l}_k{\Leftarrow}\varsigma(s_k{:}A, \tilde{x}_k{:}\widetilde{B}_k)b_k : A}$$

$$(\text{T-SUR})\quad \frac{\Gamma \vdash_D a{:}A \qquad A = \mu X[\mathrm{l}_j{:}\widetilde{B}_j{\to}B_j]_{j\in J} \qquad D \neq A}{\Gamma \vdash_D a.\mathsf{surrogate} : A}$$

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

7.3 Typing for External Surrogation

In the previous subsections, we showed that (only) external surrogations are safe; we also pointed out the inherent undecidability of the external/internal character of operations. In this subsection, we now provide a sound (and necessarily incomplete) *static* technique to rule out the occurrence of internal surrogations. The most obvious case of internal surrogation is $s.\mathsf{surrogate}$, where s is the self-variable of the immediately enclosing method. A less obvious case is $a.\mathsf{surrogate}$, 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 $a.\mathsf{surrogate}$ cannot result in $\mathsf{surrogate}$ 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 $\vdash_D$.

Theorem 7.3.1 *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 access 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 $[\![\Gamma]\!] \vdash [\![x.\mathsf{surrogate}]\!]^k_p$ for $[\![\Gamma]\!](x){=}[\![A]\!]$, $[\![\Gamma]\!](k){=}\mathbf{K}_B$ with $A \neq B$. □

Let us adapt the notion of behavioural equivalence of Definition 6.1.2 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 7.3.2 (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 7.3.3 *If* x *is an object variable, then* $x.\mathsf{ping} \doteq^{\vdash}_{\mathrm{ext}} x.\mathsf{surrogate}$*.*

8 Conclusion

In this paper we have given a formal proof of the safety of a distribution-free abstraction of object migration, called object surrogation, for the 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 occurrences of surrogation are indeed external.

Summary of the main proof idea

The proof is in two parts: an *algebraic part* and an *iterative part.*

The algebraic part (Theorem 7.1.1) relates the core component of the translation of a single object after having committed to a ping and a surrogate request, respectively. We use adaptations of powerful proof techniques, from standard π-calculus and Lπ, to prove that two such instances are barbed equivalent. Indeed, the essence of this part—and arguably of the whole contribution of this paper—is that if object aliases behave as forwarders then object surrogation introduces just a level of indirection which cannot be observed by object's clients. This is why our proof relies on the Lπ forwarder law

$$\overline{a}b = (\boldsymbol{\nu} c)\,(\overline{a}c \mid \,!\,c(x).\overline{b}x)$$

where on the left hand side we may think of emitting a direct reference for some particular object whereas on the right hand side we emit an indirect reference for this object. As already showed in [MS98, Mer00a, Mer00b], the forwarder law represents a powerful proof technique which has been used in several applications.

The iterative part (Theorem 7.2.1) relates the may-convergence behaviour of the terms x.ping and x.surrogate within arbitrary closing 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 7.2.1, we constructively simulate arbitrarily long converging sequences "up to" Theorem 7.1.1.

The main difficulties of Theorem 7.1.1 are (i) to prove that external surrogation may not be prevented from proper completion (cf. Lemma 7.1.2) and (ii) to prove that the proxy obtained from surrogation behaves as a forwarder regardless of serialisation and protection requirements (cf. Lemma 7.1.3). The main difficulty of Theorem 7.2.1 is rather notational in finding a proper way to trace the requests "arising from the hole" (cf. Definition 6.3.2). The whole proof is rather complex because of the amount of details arising from the particular combination of language features in Øjeblik. However, since surrogation requires cloning and aliasing as object operations, and it also requires protection and serialisation to make the combined operation atomic, it seems difficult to conceive a simpler, but still useful, abstraction of Obliq. To our knowledge, we give the first formal proof that object surrogation 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).

From Øjeblik to Obliq

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. As Obliq is a lexically-scoped distributed language, our result says that if we assume our semantics for object aliases, and lift the Øjeblik type system to Obliq, then any well-typed program in Obliq will never observe a difference 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.

Transferring the result to other equivalences on Øjeblik terms

Our proof shows that object surrogation is correct in terms of may-equivalence. Alternatively, one may consider stronger notions of equivalence for Øjeblik terms, such as must or barbed equivalence. The reasons why we chose may-equivalence are essentially two. Firstly, may-equivalence

is easier to work with, and makes the presentation of our results easier to understand. Secondly, even for this rather weak equivalence, the original semantics of Obliq [Car95] failed, so may-equivalence suffices to neatly underline the achievements of our semantics proposal. However, we believe that Theorem 7.1.1, which forms the basis of our safety theorem, can be reformulated in terms of stronger notions of Øjeblik equivalence. We conjecture that our main result still holds when considering must and barbed equivalences, but the amount of detail required to give the corresponding rigorous proofs would add too much complexity to the current paper.

Why a π-calculus semantics?

The modelling of objects in the π-calculus using object managers has proven to be extremely robust, being able to model not only sequential functional and imperative objects [San98, KS98], but now also concurrent objects that are equipped with non-trivial protection and serialisation requirements. Moreover, only slight variations in the definition of managers are needed to model other forms of protection and serialisation. This indicates, that this way of modelling objects is indeed a good one. More specifically, the π-calculus semantics suggested us that aliases must behave as much as possible like forwarders, even for protection-critical requests (such as cloning, aliasing, and update). This design choice is actually critical. Variations of our aliasing semantics with a stronger form of protection and serialisation (as in Cardelli [Car95] and Talcott [Tal96]), as we have modelled as an operational semantics in [NHKM00] and as a translational semantics in [Mer00b], lead to an unsafe behaviour of object migration.

Translational versus operational semantics

A 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. Apart from the translational semantics that we have provided in the current paper, we also developed operational semantics for Øjeblik in [NHKM00]. In fact, we found it very natural and useful to develop two semantics at different abstraction levels hand-in-hand. Interestingly, most of the examples of unsafe surrogation were discovered by means of the π-calculus semantics [Mer00b], and only then "verified" in the direct semantics [NHKM00].

Obviously, the question for some formal correspondence result among the semantics by translation and the direct semantics of [NHKM00] arises. Since we have developed both levels of semantics in lock-step, we have indeed a valuable basis for formalising their interrelation. However, the merits of such a result, which promises to be extremely tedious, would in our opinion not sufficiently justify the work spent.

Instead, we would rather work on a transfer of our proof structure and ideas from the level of the π-calculus to the level of the operational semantics. Now that we have given a formal proof using the π-calculus, we indeed believe that it is possible to perform this transfer. However, we still preferred to develop the proof in the π-calculus for two reasons: Firstly, the π-calculus offers a set of tools, like the forwarder law, to be exploited in proofs. Similar help does not exist in the ad-hoc setting of the operational semantics for Øjeblik of [NHKM00]. Also on the intuitive side, since the direct semantics models behaviours of programs as a whole, the forwarding character of aliases is not explicit in it, but only as part of a configuration in which requests may travel atomically along chains of aliases to some final object. We have some initial ideas to extract this hidden knowledge, which could ultimately lead to a "theory of path compression", by which certain alias nodes might be proved to be removable from the configuration. But the semantics itself does not offer any support for this. We also conjecture that we would need a "theory of partial confluence" which would allow us to complete ping and surrogation operations without affecting the convergence behaviour of the overall system. Again, the π-calculus counterpart with its quite standard notion of barbed equivalence and the syntactic structure of the translated Øjeblik terms gives us this confluence information almost for free (cf. Lemmas 7.1.2 and 7.1.6). The second reason why we preferred to develop the proof in the π-calculus was to explore how feasible it is to use the π-calculus to solve a problem of the given complexity. We believe that our

result represents a concrete and significant application of the π-calculus in the area of distributed programming language design which shows the appropriateness of the π-calculus as a tool-box for reasoning about mobile systems.

Future work

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 serialisation via self-infliction, but rather reentrant mutexes, as in Java.

A Proofs

A.1 Proof of Lemma 2.2.13

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 congruence.
Let's consider first the requirement on barbs. If $Q\{p/q\}{\downarrow_a}$, then there are two cases:

- If $a \neq p$ then $Q{\downarrow_a}$, where $a \neq q$, and therefore $(\boldsymbol{\nu} q{:}\mathbf{C}(T))\,(Q \mid q \triangleright p){\downarrow_a}$, as desired.
- If $a = p$ then there are two sub-cases:
  - either $Q{\downarrow_a}$, and therefore $(\boldsymbol{\nu} q{:}\mathbf{C}(T))\,(Q \mid q \triangleright p){\downarrow_a}$, as desired.
  - or $Q{\downarrow_q}$, and therefore $(\boldsymbol{\nu} q{:}\mathbf{C}(T))\,(Q \mid q \triangleright p){\Downarrow_a}$, as desired.

In similar manner we can prove that if $(\boldsymbol{\nu} q{:}\mathbf{C}(T))\,(Q \mid q \triangleright p){\downarrow_a}$ then $Q{\downarrow_a}$.
We prove now that the relation $\mathcal{S}$ is reduction closed.

- Let $Q\{p/q\} \stackrel{\tau}{\longrightarrow} Q'\{p/q\}$. There are two cases:
  1. either $Q \stackrel{\tau}{\longrightarrow} Q'$; this case is straightforward;
  2. or $Q\{p/q\} \stackrel{\tau}{\longrightarrow} Q'\{p/q\}$ due to a τ-action via p which cannot fire in Q. This may only happen if Q contains an occurrence of q in output subject position (we recall that q may only appear in Q in output position) and an occurrence of p in input position. Up to structural congruence, this implies that

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

     and $Q\{p/q\} \; \mathcal{S} \; (\boldsymbol{\nu} q{:}\mathbf{C}(T))\,(Q' \mid q \triangleright p)$, as desired.
- Let $(\boldsymbol{\nu} q{:}\mathbf{C}(T))\,(Q \mid q \triangleright p) \stackrel{\tau}{\longrightarrow} R$ for some R. There are two cases:
  1. either $R = (\boldsymbol{\nu} q{:}\mathbf{C}(T))\,(Q' \mid q \triangleright p)$ and $Q \stackrel{\tau}{\longrightarrow} Q'$; this case is straightforward;
  2. or the τ-action is due to a communication along q between Q and the link $q \triangleright p$. This means that $Q \equiv (\tilde{z})(Q' \mid \overline{q}v)$ and

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

     The left hand side can mimic the above move as follows:

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

     and $(\boldsymbol{\nu} \tilde{z})\,(Q' \mid \overline{p}v)\{p/q\} \; \mathcal{S} \; (\boldsymbol{\nu} q{:}\mathbf{C}(T))\,((\boldsymbol{\nu} \tilde{z})\,(Q' \mid \overline{p}v) \mid q \triangleright p)$, as desired.

$\square$

A.2 Proof of Theorem 5.1.1

To prove Theorem 5.1.1 we need the following lemma, allowing us to type object/alias managers using the translation of an object type.

Lemma A.2.1 *If* $A = \mu X[\mathrm{l}_j{:}\widetilde{B}_j{\to}B_j]_{j\in 1..n}$ *and* $\Gamma \vdash \mathbb{O}{:}A$.

$$\begin{aligned}\Gamma = \;& s{:}[\![A]\!], t_1{:}\mathbf{C}([\![A]\!], \mathbf{M}(\widetilde{B}_1\{A/X\}{\to}B_1\{A/X\}), \mathbf{K})\dots \\ & t_n{:}\mathbf{C}([\![A]\!], \mathbf{M}(\widetilde{B}_n\{A/X\}{\to}B_n\{A/X\}), \mathbf{K}), m_{\mathrm{e}}{:}\mathbf{C}(), m_{\mathrm{i}}{:}\mathbf{C}(\mathbf{K}), k_{\mathrm{e}}{:}\mathbf{K}, k_{\mathrm{i}}{:}\mathbf{K}\end{aligned}$$

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

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

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

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

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

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

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

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

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

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

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

and by lookup in Γ'' we get that

$$\begin{array}{rl}\Gamma'' \vdash & t_j{:}\mathbf{C}([\![A]\!], [\![\widetilde{B}_j\{A/X\}]\!], \mathbf{R}([\![B_j\{A/X\}]\!]), \mathbf{K}),\\ & s{:}[\![A]\!],\\ & \tilde{x}{:}[\![\widetilde{B}_j\{A/X\}]\!],\\ & r{:}\mathbf{R}([\![B_j\{A/X\}]\!]),\\ & k^*{:}\mathbf{K}\end{array}$$

□

Proof. [Proof of Theorem 5.1.1.] 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 $\Gamma \vdash [\mathrm{l}_j{=}\varsigma(s_j{:}A, \tilde{x}_j{:}\widetilde{B}_j)b_j]_{j\in J}{:}A$ with $A = \mu X[\mathrm{l}_j{:}\widetilde{B}_j{\to}B_j]_{j\in J}$. By induction

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

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

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

where $T_j = \mathbf{C}([\![A]\!], [\![\widetilde{B}_j\{A/X\}]\!], \mathbf{R}([\![B_j\{A/X\}]\!]), \mathbf{K})$ and $\tilde{t} = t_j\ j \in J$. 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 $\Gamma' \vdash \mathsf{newO}_{\mathbb{O}}\langle\, s, \tilde{t}\,\rangle$, follows from applications of (T-Res), (T-Par), narrowing and Lemma A.2.1. To establish $\Gamma' \vdash\ {!}\,t_j(s_j, \tilde{x}_j, r, k').[\![b_j]\!]_r^{k'}$ we finally 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

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

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

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

proves the last subgoal.

(T-CLO) Assume $\Gamma \vdash a.\mathsf{clone}{:}A$ with $A = \mu X[\mathrm{l}_j{:}\widetilde{B}_j{\to}B_j]_{j\in J}$. The translation of $a.\mathsf{clone}$ is

$$(\boldsymbol{\nu} q{:}\mathbf{R}(\llbracket A \rrbracket))\,(\llbracket a \rrbracket_q^k \mid q(y,k')\,.\,\overline{y}\langle \mathsf{cln_}p, k'\rangle).$$

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

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

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

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

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

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 $A = \mu X[\mathrm{l}_k{:}\widetilde{B}_k{\to}B_k]_{k\in K}$, 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 manager. The typing of the object manager also yields that we must have the types $T_j = \mathbf{C}(\llbracket A \rrbracket, \llbracket \widetilde{B}_j\{A/X\} \rrbracket, \mathbf{R}(\llbracket B_j\{A/X\} \rrbracket), \mathbf{K})$. We are now able to write a typing for $!\, t_j(s_j, \tilde{x}_j, r, k').\llbracket b_j \rrbracket_r^{k'}$, which has as premise

$$\Gamma', s{:}\llbracket A \rrbracket, t_j{:}T_j, s_j{:}\llbracket A \rrbracket, k'{:}\mathbf{K}, r_j{:}\mathbf{R}(\llbracket B_j\{A/X\} \rrbracket), \tilde{x}_j{:}\llbracket \widetilde{B}_j\{A/X\} \rrbracket \vdash \llbracket b_j \rrbracket_r^{k'}.$$

Using narrowing and the induction hypothesis we derive $\Gamma, s_j{:}A, \tilde{x}_j{:}\widetilde{B}_j\{A/X\} \vdash b_j{:}B_j\{A/X\}$. And using (T-OBJ) we conclude $\Gamma \vdash [\mathrm{l}_j{=}\varsigma(s_j{:}A, \tilde{x}_j{:}\widetilde{B}_j)b_j]_{j\in J}{:}A$.

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

$$(\boldsymbol{\nu} q{:}T)\,(\llbracket a \rrbracket_q^k \mid q(y,k')\,.\,\overline{y}\langle \mathsf{cln_}p, k'\rangle)$$

for some type annotation T. By the use of the name q we can conclude that $T = \mathbf{R}(\llbracket A \rrbracket)$ and that A cannot be a thread type (because of the $\mathsf{cln_}p$ request). Knowing that q has type $\mathbf{R}(\llbracket A \rrbracket)$ 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 5.2.2

PROOF. When reading this proof it might be helpful to consult the definition of managers in Table 7 and Figure 1 which depicts the connection between the different states of a manager.

In the proof we restrict the analysis to the cast of the object manager, the proof for the alias manager is handled the same way.

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 $\mathsf{newO}_{\mathbb{O}}\langle\, s, \tilde{t}\,\rangle$ is not fulfilled. Then

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

Using structural congruence, we immediately get

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

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

State $\mathsf{OM}^{\rm f}$ can only evolve into some state $\mathsf{OM}^{\rm a}$; it does so by grabbing the external mutex $m_{\rm e}$ for one of its pre-processed requests in $\tilde{v}$. The only other possible reduction involving state $\mathsf{OM}^{\rm f}$ is pre-processing another request, but such an action does not change the state—it only adds to the set of pre-processed requests $\tilde{v}$. 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 $\tilde{v}$, we may arrive at some Z of the form:

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

where Z corresponds to state $\mathsf{OM}^{\rm a}$.

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

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

In state $\mathsf{OM}^{\rm s}$, a png request drives the system into state $\mathsf{OM}^{\rm i}$. 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 $m_{\rm e}$ 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 not 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 $\mathsf{OM}^{\rm i}$. sur requests are treated similarly.

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

A.4 Proof of Lemma 7.1.2

Proof. We show that there is a sequence of seven τ-actions

$$\underbrace{\mathsf{surO}_{\mathbb{O}}\langle\, s, r, k, \tilde{t}, \tilde{v}\,\rangle}_{P_0}\ (\stackrel{\tau}{\rightarrow}_{\equiv})^7\ \underbrace{(\boldsymbol{\nu}s^*)\big(\mathsf{freeA}_{\mathbb{O}}\langle\, s, s^*, \tilde{v}\,\rangle \,\big|\, \mathsf{newO}_{\mathbb{O}}\langle\, s^*, \tilde{t}\,\rangle \,\big|\, \overline{r}\langle s^*, k\rangle\big)}_{P_7}.$$

and that $\approx_{\Gamma;s}$ is insensitive to each of them.

For convenience, we recall some earlier definitions, where $\tilde{n} \stackrel{\rm def}{=} m_{\rm e}, m_{\rm i}, k_{\rm e}$, which make it more feasible to list the intermediate states of the transition sequence:

$$\begin{array}{rcl} \mathit{Srv}[\cdot] & \stackrel{\rm def}{=} & (\boldsymbol{\nu}\tilde{n})\big(\overline{m_{\rm i}}k \,\big|\, \mathsf{PP}_{\mathbb{O}}\langle\, s, \tilde{n}, \tilde{v}\,\rangle \,\big|\, [\cdot]\big) \\ \mathit{OM}[\cdot] & \stackrel{\rm def}{=} & (\boldsymbol{\nu}k^*)(\,\mathsf{OM}_{\mathbb{O}}\langle\, s, \tilde{n}, k^*, \tilde{t}\,\rangle \mid [\cdot]\,) \\ \mathit{AM}[\cdot] & \stackrel{\rm def}{=} & (\boldsymbol{\nu}k^*)(\,\mathsf{AM}_{\mathbb{O}}\langle\, s, \tilde{n}, k^*, s^*\,\rangle \mid [\cdot]\,) \\ \mathsf{CM}^{r^*\succ r}_{\widetilde{m}}[\cdot] & \stackrel{\rm def}{=} & (\boldsymbol{\nu}r^*)\big(\,[\cdot] \;\mid\; r^*(y, k').m_{\rm i}(k'').(\,\overline{r}\langle y, k''\rangle \;\mid\; \overline{m_{\rm e}}\,)\big) \\ \mathsf{surO}_{\mathbb{O}}\langle\, s, r, k, \tilde{t}, \tilde{v}\,\rangle & \stackrel{\rm def}{=} & \mathit{Srv}\underbrace{[\mathit{OM}[\mathsf{CM}^{r^*\succ r}_{\widetilde{m}}[\,[\![\, s.\mathsf{alias}\langle s.\mathsf{clone}\rangle\,]\!]^{k^*}_{r^*}\,]]]}_{S_0} \end{array}$$

Let $P_i \stackrel{\text{def}}{=} Srv[S_i]$ for $i \in 0..6$ with:

$$
\begin{array}{l}
S_0 \stackrel{\text{def}}{=} OM[\mathsf{CM}_{\widetilde{m}}^{r^* \succ r}[\,(\boldsymbol{\nu} \underline{q})\,\Big(\underline{[\![\,s\,]\!]_q^{k^*}} \mid \underline{q(y,k)}.(\boldsymbol{\nu} q')\,\big([\![\,s.\mathsf{clone}\,]\!]_{q'}^{k^*} \mid q'(y',k').\overline{y}\langle \mathsf{ali_}\langle y', r^*\rangle, k'\rangle\big)\Big)\,]] \\
S_1 \stackrel{\text{def}}{=} OM[\mathsf{CM}_{\widetilde{m}}^{r^* \succ r}[\,(\boldsymbol{\nu} q')\,\Big((\boldsymbol{\nu} \underline{p})\,\big(\underline{[\![\,s\,]\!]_p^{k^*}} \mid \underline{p(y,k)}.\overline{y}\langle \mathsf{cln_}\langle q', k\rangle\rangle\big) \mid q'(y',k').\overline{s}\langle \mathsf{ali_}\langle y', r^*\rangle, k'\rangle\Big)\,]] \\
S_2 \stackrel{\text{def}}{=} \underline{OM}[\mathsf{CM}_{\widetilde{m}}^{r^* \succ r}[\,(\boldsymbol{\nu} q')\,\Big(\underline{\overline{s}\langle \mathsf{cln_}\langle q', k^*\rangle\rangle} \mid q'(y',k').\overline{s}\langle \mathsf{ali_}\langle y', r^*\rangle, k'\rangle\Big)\,]] \\
S_3 \stackrel{\text{def}}{=} OM[\mathsf{CM}_{\widetilde{m}}^{r^* \succ r}[\,(\boldsymbol{\nu} \underline{q'})\,\Big((\boldsymbol{\nu} s^*)\,\big(\underline{\overline{q'}\langle s^*, k^*\rangle} \mid \mathsf{newO}_{\mathbb{O}}\langle\, s^*, \tilde{t}\,\rangle\big) \mid \underline{q'(y',k')}.\overline{s}\langle \mathsf{ali_}\langle y', r^*\rangle, k'\rangle\Big)\,]] \\
S_4 \stackrel{\text{def}}{=} \underline{OM}[\mathsf{CM}_{\widetilde{m}}^{r^* \succ r}[\,(\boldsymbol{\nu} s^*)\,\big(\mathsf{newO}_{\mathbb{O}}\langle\, s^*, \tilde{t}\,\rangle \mid \underline{\overline{s}\langle \mathsf{ali_}\langle s^*, r^*\rangle, k^*\rangle}\big)\,]] \\
S_5 \stackrel{\text{def}}{=} (\boldsymbol{\nu} s^*) AM[\underline{\mathsf{CM}_{\widetilde{m}}^{r^* \succ r}}[\,\big(\mathsf{newO}_{\mathbb{O}}\langle\, s^*, \tilde{t}\,\rangle \mid \underline{\overline{r^*}\langle s^*, k^*\rangle}\big)\,]] \\
S_6 \stackrel{\text{def}}{=} (\boldsymbol{\nu} s^*) AM[\big(\mathsf{newO}_{\mathbb{O}}\langle\, s^*, \tilde{t}\,\rangle \mid \underline{m_{\mathrm{i}}(k)}.(\,\overline{r}\langle s^*, k\rangle \mid \overline{m_{\mathrm{e}}}\,)\big)]
\end{array}
$$

and

$$
\begin{array}{rl}
P_6 & = (\boldsymbol{\nu}\tilde{n})\big(\overline{m_{\mathrm{i}}}k \mid \mathsf{PP}_{\mathbb{O}}\langle\, s, \tilde{n}, \tilde{v}\,\rangle \mid (\boldsymbol{\nu} s^*) AM[\big(\mathsf{newO}_{\mathbb{O}}\langle\, s^*, \tilde{t}\,\rangle \mid \underline{m_{\mathrm{i}}(k)}.(\,\overline{r}\langle s^*, k\rangle \mid \overline{m_{\mathrm{e}}}\,)\big)]\,\big) \\
& \xrightarrow{\tau} (\boldsymbol{\nu}\tilde{n})\big(\overline{m_{\mathrm{e}}} \mid \mathsf{PP}_{\mathbb{O}}\langle\, s, \tilde{n}, \tilde{v}\,\rangle \mid (\boldsymbol{\nu} s^*) AM[\big(\mathsf{newO}_{\mathbb{O}}\langle\, s^*, \tilde{t}\,\rangle \mid \overline{r}\langle s^*, k\rangle\big)]\,\big) \\
& \equiv (\boldsymbol{\nu} s^*)\Big(\underline{(\boldsymbol{\nu}\tilde{n})\big(\overline{m_{\mathrm{e}}} \mid \mathsf{PP}_{\mathbb{O}}\langle\, s, \tilde{n}, \tilde{v}\,\rangle \mid (\boldsymbol{\nu} k^*)\mathsf{AM}_{\mathbb{O}}\langle\, s, \tilde{n}, k^*, s^*\,\rangle\big)} \mid \mathsf{newO}_{\mathbb{O}}\langle\, s^*, \tilde{t}\,\rangle \mid \overline{r}\langle s^*, k\rangle\Big) \\
& \equiv (\boldsymbol{\nu} s^*)\,\Big(\mathsf{freeA}_{\mathbb{O}}\langle\, s, s^*, \tilde{v}\,\rangle \mid \mathsf{newO}_{\mathbb{O}}\langle\, s^*, \tilde{t}\,\rangle \mid \overline{r}\langle s^*, k\rangle\Big) = P_7
\end{array}
$$

The reader may observe, by verifying that the underlined components give rise to reductions, that there are reduction steps $P_i \xrightarrow{\tau}\equiv P_{i+1}$ for $i \in 0..6$, which are of three kinds:

1. confluent reductions (for $i \in \{0,1,3,5\}$) along restricted channels of the form

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

   where $q \notin \mathrm{fn}(P)$, and
2. reductions (for $i \in \{2,4\}$) involving internal requests induced by the surrogation of the form

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

3. confluent reductions (for $i = 6$) along restricted channels of the form

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

   where any occurrence of q in subject position within Q is guarded by some prefix with subject p, which may only be triggered from within P.

It is well-known that $\approx_\Gamma$ (as well as $\approx_{\Gamma;s}$) is insensitive to reductions of the first kind. In Lemma A.4.1 below, we show that $\approx_{\Gamma;s}$ is insensitive to the reductions of the second kind.Finally, In Lemma A.4.2 below, we show that $\approx_\Gamma$ (and therefore also $\approx_{\Gamma;s}$) is insensitive to the reductions of the third kind.This conclude the proof. $\square$

The auxiliary lemma A.4.1 is possible, because in the implementation of object and alias managers, we use nonces (cf. page 22) in order to guarantee that the internal key of the object manager, as represented by the context $OM[\cdot]$, is always restricted. In this manner, the environment cannot produce any "malicious" internal request which might potentially interfere with the cloning and the aliasing requests.

Lemma A.4.1 *Let $P_2 \rightarrow_\equiv P_3$ and $P_4 \rightarrow_\equiv P_5$ be defined as above. Then:*

1. $P_2 \approx_{\Gamma;s} P_3$
2. $P_4 \approx_{\Gamma;s} P_5$

PROOF.

1. For simplicity, we omit the obligations on types in the coinductive definition of $\approx_{\Gamma;s}$. For the proof, it is handy to carry explicitly as a parameter the set of values, on which the component PP, and thus the P_i are defined. So, we rewrite P_2 as $P_2\langle\tilde{v}\rangle$ and P_3 as $P_3\langle\tilde{v}\rangle$. We prove that the relation:

$$\mathcal{S} = \{(P_2\langle\,\widetilde{w}\,\rangle, P_3\langle\,\widetilde{w}\,\rangle) : \widetilde{w} = w_1 \mathinner{..} w_m \text{ with } w_j := \langle\, l_j, k_j\,\rangle,\ j \in 1..m\} \,\cup\, \mathcal{I}$$

where $\mathcal{I}$ is the identity relation, is a $\approx_{\Gamma;s}$-bisimulation up to $\equiv$.
The only channel which appears free in subject position in $P_2\langle\,\widetilde{w}\,\rangle$ and $P_3\langle\,\widetilde{w}\,\rangle$ is s. Since both the external key k_e and the internal key k^* are restricted in $P_2\langle\,\widetilde{w}\,\rangle$ and $P_3\langle\,\widetilde{w}\,\rangle$, and 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 $P_2\langle\,\widetilde{w}\,\rangle$ 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. Notice that due the type requirement on s we do not consider outputs via s. 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 $\mathsf{PP}_{\mathbb{O}}\langle\, s,\tilde{n},\widetilde{w}\,\rangle$ obtaining some $\mathsf{PP}_{\mathbb{O}}\langle\, s,\tilde{n},\widetilde{w}'\,\rangle$ with $\widetilde{w}' = \widetilde{w} \cup \langle l,k\rangle$. The process $P_3\langle\,\widetilde{w}\,\rangle$ can perform the same action and the derivatives are again related by $\mathcal{S}$. In case (ii), the process $P_3\langle\,\widetilde{w}\,\rangle$ can mimic the τ-action by not performing any reduction at all. Up to structural congruence, we get into the identity relation.
The process $P_3\langle\,\widetilde{w}\,\rangle$ can only perform two kinds of actions. Either (i) an 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 $P_2\langle\,\widetilde{w}\,\rangle$ can perform two silent actions, along s and q, getting, up to structural congruence, into the identity relation.
2. Analogous to the previous case.

□

Lemma A.4.2 *Let P_6 and P_7 defined as above. Then, $P_6 \approx_\Gamma P_7$.*

PROOF. For simplicity, we omit the obligations on types in the coinductive definition of $\approx_{\Gamma;s}$. For the proof, it is handy to carry explicitly as a parameter the set of values, on which the component PP, and thus the P_i are defined. So, we rewrite P_6 as $P_6\langle\tilde{v}\rangle$ and P_7 as $P_7\langle\tilde{v}\rangle$. Let $\mathcal{I}$ be the identity relation, we show that the relation

$$\mathcal{S} = \{(\,P_6\langle\widetilde{w}\rangle, P_7\langle\widetilde{w}\rangle\,) : \widetilde{w} = w_1 \mathinner{..} w_m \text{ with } w_j := \langle\, l_j, k_j\,\rangle,\ j \in 1..m\} \cup \mathcal{I}$$

is a $\approx_\Gamma$-bisimilarity up to $\equiv$.

The process $P_6\langle\widetilde{w}\rangle$ may perform only two actions:

1. $P_6\langle\widetilde{w}\rangle \xrightarrow{s\langle l,k\rangle}\equiv P_6\langle\widetilde{w'}\rangle$, with $\widetilde{w'} = \widetilde{w} \cup \langle l,k\rangle$. Since k_i and k_e are bound in $P_6\langle\widetilde{w}\rangle$ it holds that $k_\mathrm{i} \neq k \neq k_\mathrm{e}$. In this case, $P_7\langle\widetilde{w}\rangle \xrightarrow{s\langle l,k\rangle}\equiv P_7\langle\widetilde{w'}\rangle$ and we are done.
2. $P_6\langle\widetilde{w}\rangle \xrightarrow{\tau} R$. This τ-action is only possible when consuming the internal mutex m_i. In this case, $P_7\langle\widetilde{w}\rangle \Rightarrow P_7\langle\widetilde{w}\rangle \equiv R$ where $P_7\langle\widetilde{w}\rangle$ performs no actions at all. Since $\mathcal{S}$ contains the identity relation $\mathcal{I}$ we are done.

The process $P_7\langle\widetilde{w}\rangle$ may perform only three actions:

1. $P_7\langle\widetilde{w}\rangle \xrightarrow{s\langle l,k\rangle} P_7\langle\widetilde{w'}\rangle$, with $\widetilde{w'} = \widetilde{w} \cup \langle l,k\rangle$. We reason as in the first of the two cases above.
2. $P_7\langle\widetilde{w}\rangle \xrightarrow{\tau} R$. This τ-action is only possible when consuming the external mutex m_e. In this case, $P_6\langle\widetilde{w}\rangle \xrightarrow{\tau}\xrightarrow{\tau}\equiv R$. Where the first τ action is due to the consumption of the internal mutex m_i, while the second τ-action is due to the consumption of the external mutex m_e. Again, since $\mathcal{S}$ contains the identity we are done.
3. $P_7\langle\widetilde{w}\rangle \xrightarrow{(\nu s^*)\quad \overline{r}\langle s^*,k\rangle} R$. Then $P_6\langle\widetilde{w}\rangle \xrightarrow{\tau} \xrightarrow{(\nu s^*)\,\overline{r}\langle s^*,k\rangle}\equiv R$, where the τ-action is due to the consumption of the internal mutex m_i.

This concludes the proof.

□

A.5 Proof of Lemma 7.1.3

Lemma 7.1.3 proves that the aliased object manager appearing in Lemma 7.1.2 behaves as a forwarder. As a first step we recall a property of replicated input which is proved in [San99a] by using typed variant of Milner's replication theorems [Mil93].

Lemma A.5.1 *Let P be a process, $C[\cdot]$ a π-calculus context, and s a name such that $s \notin \mathrm{n}(P, C[\cdot])$. Let Γ be a suitable type environment for the processes below such that $\Gamma \vdash s{:}\mathbf{C}(T)$. Then*

$$! \, s(x).P \mid C[\overline{s}v] \simeq_{\Gamma;s} ! \, s(x).P \mid C[P\{^v/_x\}]$$

The following lemma makes explicit the behaviour of the alias manager created during the surrogation of an object. In such alias managers the internal key k_i is always restricted and never extruded. As an abbreviation we use a construct $\mathsf{if}\ [k{=}k']\ \mathsf{then}\ P\ \mathsf{else}\ Q$ which can be easily rewritten in terms of our value testing.

Lemma A.5.2 *Let Γ be a suitable environment for the processes below, then*

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

PROOF. By exhibiting the appropriate bisimulation. We prove that the relation

$$\mathcal{S} = \{(LHS, RHS)\}$$

where LHS and RHS denote left hand side, right hand side respectively of the identity, is a $\approx_\Gamma$-bisimilarity up to $\equiv$ and up to context [San96]. □

The next lemma shows that the mutex-protocol, in the case of alias managers, only adds harmless τ-actions. This is the crucial step to prove that alias nodes created during surrogation essentially behave like forwarders.

Lemma A.5.3 *Let $\tilde{v} := v_1 \,..\, v_n$ with $v_j := \langle\, l_j, k_j \,\rangle$ for $j \in 1..n$. Let Γ be a suitable type environment for the processes P and Q defined below:*

$$P\langle\tilde{v}\rangle \stackrel{\mathrm{def}}{=} (\boldsymbol{\nu}\widetilde{m})\,\big(\,\overline{m_\mathrm{e}} \mid \, ! \, s(l,k).m_\mathrm{e}.(m_\mathrm{i}(k).(\overline{s^*}\langle l,k\rangle \mid \overline{m_\mathrm{e}}) \mid \overline{m_\mathrm{i}}k) \mid \prod_{j\in 1..n} m_\mathrm{e}.(m_\mathrm{i}(k).(\overline{s^*}\langle l_j,k\rangle \mid \overline{m_\mathrm{e}}) \mid \overline{m_\mathrm{i}}k_j)\,\big)$$

and

$$Q\langle\tilde{v}\rangle \stackrel{\mathrm{def}}{=} ! \, s(l,k).\overline{s^*}\langle l,k\rangle \mid \prod_{j\in 1..n} \overline{s^*}\langle l_j,k_j\rangle$$

Then, $P\langle\tilde{v}\rangle \approx_\Gamma Q\langle\tilde{v}\rangle$.

PROOF. For simplicity, we omit the obligations on types on the coinductive definition of $\approx_\Gamma$. We prove that the relation $\mathcal{S} = \{(P\langle w\rangle, Q\langle w\rangle) : \widetilde{w} = w_1 \,..\, w_m$ with $w_j := \langle\, l_j, k_j \,\rangle,\ j \in 1..m\}$ is a $\approx_\Gamma$-bisimilarity up-to expansion and up-to context [SM92, San96].

Let's consider all the possible actions of $P\langle\widetilde{w}\rangle$ and $Q\langle\widetilde{w}\rangle$.

1. The case $P\langle\widetilde{w}\rangle \xrightarrow{s\langle l,k\rangle} P'\langle\widetilde{w'}\rangle$ with $\widetilde{w'} = \widetilde{w} \cup \langle l,k\rangle$ is straightforward.
   Similarly for $Q\langle\widetilde{w}\rangle \xrightarrow{s\langle l,k\rangle} Q'\langle\widetilde{w'}\rangle$.
2. If $Q\langle\widetilde{w}\rangle \xrightarrow{\overline{s^*}\langle l_j,k_j\rangle} Q'\langle\widetilde{w'}\rangle$, with $\widetilde{w'} = \widetilde{w} \setminus \langle l_j,k_j\rangle$,
   then $P\langle\widetilde{w}\rangle \xrightarrow{\tau}\xrightarrow{\tau} \xrightarrow{\overline{s^*}\langle l_j,k_j\rangle} \equiv P'\langle\widetilde{w'}\rangle$ and $P'\langle\widetilde{w'}\rangle \;\mathcal{S}\; Q'\langle\widetilde{w'}\rangle$.

3. Finally, the most interesting case is when

$$P\langle\widetilde{w}\rangle \xrightarrow{\tau} (\boldsymbol{\nu}\widetilde{m})\big(\ !\,s(l,k).m_{\mathrm{e}}.(m_{\mathrm{i}}(k).(\overline{s^*}\langle l,k\rangle \mid \overline{m_{\mathrm{e}}}) \mid \overline{m_{\mathrm{i}}}k) \mid \prod_{z\in 1..n, z\neq j} m_{\mathrm{e}}.(m_{\mathrm{i}}(k).(\overline{s^*}\langle l_z,k\rangle \mid \overline{m_{\mathrm{e}}}) \mid \overline{m_{\mathrm{i}}}k_z) \mid m_{\mathrm{i}}(k).(\overline{s^*}\langle l_j,k\rangle \mid \overline{m_{\mathrm{e}}}) \mid \overline{m_{\mathrm{i}}}k_j\ \big)$$

In the process above the subterm $m_{\mathrm{i}}(k).(\overline{s^*}\langle l_j,k\rangle \mid \overline{m_{\mathrm{e}}}) \mid \overline{m_{\mathrm{i}}}k_j$ contains an input and an output along the internal mutex m_{i}. No other terms can interfere on this communication along m_{i}. This is because all the other occurrences of m_{i} in subject position are guarded by an input along m_{e} which will be eventually released only after the aforementioned reduction along m_{i} takes place. As a consequence, the process above is related by the expansion relation $\gtrsim$ to the process below

$$P' \stackrel{\mathrm{def}}{=} (\boldsymbol{\nu}m_{\mathrm{e}})\big(\ \overline{m_{\mathrm{e}}} \mid !\,s(l,k).m_{\mathrm{e}}.(m_{\mathrm{i}}(k).(\overline{s^*}\langle l,k\rangle \mid \overline{m_{\mathrm{e}}}) \mid \overline{m_{\mathrm{i}}}k) \mid \prod_{z\in 1..n, z\neq j} m_{\mathrm{e}}.(m_{\mathrm{i}}(k).(\overline{s^*}\langle l_z,k\rangle \mid \overline{m_{\mathrm{e}}}) \mid \overline{m_{\mathrm{i}}}k_z)\big) \mid \overline{s^*}\langle l_j,k_j\rangle$$

So, there exist a static context $C[\cdot] \stackrel{\mathrm{def}}{=} [\cdot] \mid \overline{s^*}\langle l_j,k_j\rangle$ and processes $P''\langle\widetilde{w''}\rangle$ and $Q''\langle\widetilde{w''}\rangle$ with $\widetilde{w''} = \widetilde{w} \setminus \langle l_j,k_j\rangle$ such that $P \xrightarrow{\tau} P' \gtrsim C[P''\langle\widetilde{w''}\rangle]$ and $Q \Rightarrow Q \gtrsim C[Q''\langle\widetilde{w''}\rangle]$ with $P'\langle\widetilde{w''}\rangle\ \mathcal{S}\ Q'\langle\widetilde{w''}\rangle$.

This suffices to prove that $\mathcal{S}$ is a $\approx_\Gamma$-bisimilarity up to expansion and up to context. □

PROOF. [Proof of Lemma 7.1.3.] 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 $(\boldsymbol{\nu}k_{\mathrm{i}})\,\mathsf{AM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k_{\mathrm{i}},s^*\,\rangle$ (see Table 1) we note that, since k_{i} is restricted and never extruded, the aliased object manager will never receive self-inflicted requests. By Lemma A.5.2 we have

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

We recall that

$$\mathsf{freeA}_{\mathbb{O}}\langle\, s,s^*,\tilde{v}\,\rangle \stackrel{\mathrm{def}}{=} (\boldsymbol{\nu}m_{\mathrm{e}}m_{\mathrm{i}}k_{\mathrm{e}})\big(\overline{m_{\mathrm{e}}} \mid (\boldsymbol{\nu}k_{\mathrm{i}})\,\mathsf{AM}_{\mathbb{O}}\langle\, s,\widetilde{m},k_{\mathrm{e}},k_{\mathrm{i}},s^*\,\rangle \mid \prod_{j\in 1..n} m_{\mathrm{e}}.\big(\overline{s}\langle l_j,k_{\mathrm{e}}\rangle \mid \overline{m_{\mathrm{i}}}k_j\big)\big).$$

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

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

Since the environment cannot use s in input, 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 "know" k_{e}.

By Lemma A.5.1 the right hand side of the equation above is related by $\simeq_{\Gamma;s}$ to the process below

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

By Lemma A.5.3 the process above is $\approx_\Gamma$-bisimilar to the process below

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

which by definition is the required process

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

□

A.6 Proof of Lemma 7.1.4

This is a rather technical lemma. Lemma 7.1.4 essentially says that a restricted forwarder can be replaced by a substitution. This is the only proof where we use the theory of Lπ, and more specifically the forwarder law of Lemma 2.2.14.

PROOF. We apply Lemma 2.2.14 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 2.2.14 in such a way. Note that we focus only on channel s^*. The other channels are not affected by our transformation. Since Lemma 2.2.14 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 always 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 7.1.5

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

Lemma A.7.1 *Let $\tilde{v} := v_1 \,..\, v_n$ 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:*

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

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

Case $n = 0$. Trivial.

Inductive case. Let

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

By inductive hypothesis it holds that:

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

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

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

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

we prove that the relation:

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

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

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

1. If $(\boldsymbol{\nu}\tilde{z})\,(A \mid R) \xrightarrow{\mu} (\boldsymbol{\nu}\tilde{y})\,(A \mid R')$ then it is easy.
2. If $(\boldsymbol{\nu}\tilde{z})\,(A \mid R) \xrightarrow{s\langle l,k\rangle} (\boldsymbol{\nu}\tilde{z})\,(A' \mid R)$, 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 $(\boldsymbol{\nu}\tilde{z})\,(B' \mid R)$. By inspection of the encoding we have that $(\boldsymbol{\nu}\tilde{z})\,(A' \mid R) \equiv (\boldsymbol{\nu}\tilde{y})\,(A'' \mid R')$ and $(\boldsymbol{\nu}\tilde{z})\,(B' \mid R) \equiv (\boldsymbol{\nu}\tilde{y})\,(B'' \mid R')$, for some $\tilde{y}$ 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 $(\boldsymbol{\nu}\tilde{y})\,(A'' \mid R')\ \mathcal{S}\ (\boldsymbol{\nu}\tilde{y})\,(B'' \mid R')$.
3. If $(\boldsymbol{\nu}\tilde{z})\,(A \mid R) \xrightarrow{\tau} (\boldsymbol{\nu}\tilde{y})\,(A' \mid R')$, 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 $(\boldsymbol{\nu}\tilde{z})\,(A \mid R) \xrightarrow{\tau} (\boldsymbol{\nu}\tilde{z})\,(A' \mid R)$, where the τ action 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 $(\boldsymbol{\nu}\tilde{z})\,(B \mid R) \Longrightarrow (\boldsymbol{\nu}\tilde{z})\,(B \mid R)$. It holds that $(\boldsymbol{\nu}\tilde{z})\,(A' \mid R) \equiv\mathcal{S}\ (\boldsymbol{\nu}\tilde{z})\,(B \mid R)$.

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

1. If $(\boldsymbol{\nu}\tilde{z})\,(B \mid R) \xrightarrow{\mu} (\boldsymbol{\nu}\tilde{y})\,(B \mid R')$ then it is easy.
2. If $(\boldsymbol{\nu}\tilde{z})\,(B \mid R) \xrightarrow{s\langle l,k\rangle} (\boldsymbol{\nu}\tilde{z})\,(B' \mid R)$, 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 $(\boldsymbol{\nu}\tilde{z})\,(A' \mid R)$. By inspection of the encoding we have that $(\boldsymbol{\nu}\tilde{z})\,(B' \mid R) \equiv (\boldsymbol{\nu}\tilde{y})\,(B'' \mid R')$ and $(\boldsymbol{\nu}\tilde{z})\,(A' \mid R) \equiv (\boldsymbol{\nu}\tilde{y})\,(A'' \mid R')$, for some $\tilde{y}$ 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 $(\boldsymbol{\nu}\tilde{y})\,(B'' \mid R')\ \mathcal{S}\ (\boldsymbol{\nu}\tilde{y})\,(A'' \mid R')$.
3. If $(\boldsymbol{\nu}\tilde{z})\,(B \mid R) \xrightarrow{\tau} (\boldsymbol{\nu}\tilde{y})\,(B' \mid R')$, 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 $(\boldsymbol{\nu}\tilde{z})\,(B \mid R) \xrightarrow{m_{\mathrm{e}}} (\boldsymbol{\nu}\tilde{z})\,(B' \mid R)$ and $B' = \mathsf{OM}_{\mathbb{O}}\langle\, s, \tilde{m}, \tilde{k}, \tilde{t}\,\rangle \mid \overline{s}\langle l_1, k_{\mathrm{e}}\rangle \mid \overline{m_{\mathrm{i}}}k_1$, then the left side can mimic this action by serving the request $\overline{s}v_1$ and then grabbing the mutex. In practice,
$(\boldsymbol{\nu}\tilde{z})\,(A \mid R) \xrightarrow{\tau} \xrightarrow{m_{\mathrm{e}}} (\boldsymbol{\nu}\tilde{z})\,(A' \mid R)$ with $A' \equiv B'$. So, $(\boldsymbol{\nu}\tilde{z})\,(B' \mid R)\ \mathcal{S}\ (\boldsymbol{\nu}\tilde{z})\,(A' \mid R)$.

□

PROOF. [Proof of Lemma 7.1.5.] It follows directly from Lemma A.7.1 and the fact that $\approx_{\Gamma;s}$ is preserved by parallel composition and restriction. □

A.8 Proof of Lemma 7.1.6

PROOF. We show that there is a sequence of two τ-actions

$$\underbrace{\mathsf{pngO}_{\mathbb{O}}\langle\, s, r, k, \tilde{t}, \tilde{v}\,\rangle}_{P_0}\ (\xrightarrow{\tau}_{\equiv})^2\ \underbrace{\mathsf{freeO}_{\mathbb{O}}\langle\, s, s^*, \tilde{v}\,\rangle \mid \overline{r}\langle s^*, k\rangle}_{P_2}.$$

and that $\approx_\Gamma$ is insensitive to each of them. We recall some definitions, with $\tilde{n} \stackrel{\text{def}}{=} m_e, m_i, k_e$:

$$\begin{array}{rcl}
Srv[\cdot] & \stackrel{\text{def}}{=} & (\boldsymbol{\nu}\tilde{n})\big(\overline{m_i}k \mid \mathsf{PP}_{\mathbb{O}}\langle\, s, \tilde{n}, \tilde{v}\,\rangle \mid [\cdot]\,\big) \\
OM[\cdot] & \stackrel{\text{def}}{=} & (\boldsymbol{\nu}k^*)(\,\mathsf{OM}_{\mathbb{O}}\langle\, s, \tilde{n}, k^*, \tilde{t}\,\rangle \mid [\cdot]\,) \\
\mathsf{CM}_{\tilde{m}}^{r^* \succ r}[\cdot] & \stackrel{\text{def}}{=} & (\boldsymbol{\nu}r^*)\big(\,[\cdot] \mid r^*(y,k').m_i(k'').(\,\overline{r}\langle y, k''\rangle \mid \overline{m_e}\,)\big) \\
\mathsf{pngO}_{\mathbb{O}}\langle\, s, r, k, \tilde{t}, \tilde{v}\,\rangle & \stackrel{\text{def}}{=} & Srv[OM[\mathsf{CM}_{\tilde{m}}^{r^* \succ r}[\,[\![\, s\,]\!]_{r^*}^{k^*}]]]
\end{array}$$

The sequence $P_0 \rightarrow_\equiv P_1 \rightarrow_\equiv P_2$ is then established with:

$$\begin{array}{rcl}
P_0 & = & Srv[\,OM[\,\underline{\mathsf{CM}_{\tilde{m}}^{r^* \succ r}[\,\overline{r^*}\langle s, k^*\rangle}\,]\,]\,] \\
P_1 & \stackrel{\text{def}}{=} & Srv[\,OM[\,m_i(k'').(\,\overline{r}\langle s, k''\rangle \mid \overline{m_e}\,)\,]\,] \\
& \equiv & (\boldsymbol{\nu}\tilde{n})\big(\underline{\overline{m_i}k} \mid \mathsf{PP}_{\mathbb{O}}\langle\, s, \tilde{n}, \tilde{v}\,\rangle \mid OM[\,\underline{m_i(k'')}.(\,\overline{r}\langle s, k''\rangle \mid \overline{m_e}\,)\,]\big) \\
& \xrightarrow{\tau}_\equiv & (\boldsymbol{\nu}\tilde{n})\big(\overline{m_e} \mid \mathsf{PP}_{\mathbb{O}}\langle\, s, \tilde{n}, \tilde{v}\,\rangle \mid OM[\,\overline{r}\langle s, k\rangle\,]\,\big) \\
& \equiv & \underline{(\boldsymbol{\nu}\tilde{n})\big(\overline{m_e} \mid \mathsf{PP}_{\mathbb{O}}\langle\, s, \tilde{n}, \tilde{v}\,\rangle \mid (\boldsymbol{\nu}k^*)\,\mathsf{OM}_{\mathbb{O}}\langle\, s, \tilde{n}, k^*, \tilde{t}\,\rangle\big)} \mid \overline{r}\langle s, k\rangle \\
& \equiv & \mathsf{freeO}_{\mathbb{O}}\langle\, s, \tilde{t}, \tilde{v}\,\rangle \mid \overline{r}\langle s, k\rangle = P_2
\end{array}$$

The reduction $P_0 \rightarrow_\equiv P_1$ along r^* is confluent, and therefore insensitive to $\approx_\Gamma$, as for the first kind of steps in Appendix A.4. The second step is proved through a minor variation of Lemma A.4.2. Essentially, the two terms above differ for a reduction along the internal mutex m_i. □

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.

[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.

[KPT96] N. Kobayashi, B. C. Pierce and D. N. Turner. Linearity and the Pi-Calculus. In *Proceedings of POPL '96*, pages 358–371. ACM, Jan. 1996.

[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.

[Mer00a] M. Merro. Locality and Polyadicity in Asynchronous Name-Passing Calculi. In J. Tiuryn, ed, *Proceedings of FoSSaCS 2000*, volume 1784 of *LNCS*, pages 238–251. Springer, 2000.

[Mer00b] 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.

[San96] D. Sangiorgi. Locality and non-interleaving semantics in calculi for mobile processes. *Theoretical Computer Science*, 155, 1996. See also Report ECS-LFCS-94-282, University of Edinburgh. An extract appeared in *Proceedings of TACS'94*, LNCS 789.

[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.

[SM92] D. Sangiorgi and R. Milner. The Problem of "Weak Bisimulation up to". In R. Cleaveland, ed, *Proceedings of CONCUR '92*, volume 630 of *LNCS*, pages 32–46. Springer, 1992.

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

[Tal96] C. L. Talcott. Obliq semantics notes. Unpublished note. Available from `clt@cs.stanford.edu`, Jan. 1996.

[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.