# Migration = Cloning ; Aliasing
# (preliminary version)

Hans Hüttel  Josva Kleist  Massimo Merro[*]  Uwe Nestmann[†]

BRICS[‡], Aalborg University[§], Denmark

**Abstract**

In Obliq, a lexically scoped, distributed, object-based programming language, object migration was suggested as creating a (remote) copy of an object's state at the target site, followed by turning the (local) object itself into an alias, also called *surrogate*, for the just created remote copy. There is no proof that object migration in Obliq is safe in any sense.

We consider the creation of object surrogates as an abstraction of the above-mentioned style of migration. We introduce Øjeblik, a distribution-free subset of Obliq, and provide two formal semantics, one in an intuitive configuration style, the other in terms of $\pi$-calculus. The intuitive semantics shows why surrogation is neither safe in Obliq, nor can it be so in full generality in Repliq (a repaired Obliq). The $\pi$-calculus semantics allows us to prove that surrogation in Øjeblik is safe for certain well-identified cases, thus suggesting that migration in Repliq may be safe, accordingly.

## 1 Motivation: From migration to surrogation

Øjeblik[1] is an object-based language that is not only inspired by Obliq [Car95], but rather represents its concurrent core. Obliq is a lexically scoped, distributed, object-based programming language. Lexical scoping in distributed settings makes program analysis easier since the binding of variables is only determined by their location in the program text and not by the site at which execution takes place.

It can be advantageous, for example for efficiency improvements, to migrate an object from one site to another, which is also called for in Obliq. Here, however, mutable values in general are never sent over the network; instead, only network references are transmitted. Accordingly, migration of objects is carried out in Obliq by creating a copy of the object at the target site and then modifying the original (local) object such that it forwards all future requests to the new (remote) object. Since Obliq is lexically scoped, we may safely ignore the aspects of distribution, so in Øjeblik we concentrate on just the concurrent aspects: *surrogation* of an object $a$ can be described as creating a copy $b$ of $a$ and then turn $a$ itself into a proxy for $b$, i.e., which forwards all future request for methods of $a$ to $b$.

[*]On leave from INRIA, Sophia-Antipolis, France

[†]{hans,kleist,nice,uwe}@cs.auc.dk

[‡]Basic Research in Computer Science, Centre of the Danish National Research Foundation.

[§]Department of Computer Science, Fredrik Bajers Vej 7E, DK-9220 Aalborg Ø

[1]Danish for *moment*, '*blink of the eye*'.

## 2 Øjeblik, a language for serialized concurrent objects

In this section, we present Øjeblik as an untyped language (types can be added in a straightforward manner), but we sometimes refer to types when we think it helps us in explaining our design decisions or it eases the understanding. For the sake of simplicity, compared to Obliq we

1. omit ground values, data operations, and procedures,
2. restrict field selection to just method invocations,
3. restrict multiple cloning to single cloning,
4. omit flexibility of object attributes,
5. replace field aliasing with object aliasing,
6. omit explicit distribution (sites, engines, name-servers),
7. omit exceptions and advanced synchronization,

so we get a still non-trivial, but more feasible language. The set $\mathcal{L}$ of Øjeblik-expressions is given by the grammar in Figure 2, where the $\mathrm{l}$ represent method *labels*.

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

Figure 1: Syntax of Øjeblik expressions

An object $[\mathrm{l}_j {=} m_j]_{j\in J}$ consists of a finite collection of named methods $\mathrm{l}_j {=} m_j$, more generally called fields, for pairwise distinct labels $\mathrm{l}_j$.

In a method $\varsigma(s, \tilde{x})b$ the letter $\varsigma$ is a binder for the self variable $s$ and argument variables $\tilde{x}$ (a tuple $x_1 \mathinner{..} x_n$) within the method body $b$. In order to invoke a method, we are required to supply a number of actual parameters: usually, $a.\mathrm{l}\langle\, a_1 \mathinner{..} a_n \,\rangle$ with field $\mathrm{l}$ containing the method $\varsigma(s, \tilde{x})b$ results in the body $b$ with the enclosing object $a$ bound to the

$$
\begin{array}{lll}
a, b ::= \ldots \mid v \mid \mathsf{wait} & r ::= \mathbb{O} & e[\cdot] ::= [\cdot] \\
 & \quad \mid o.\mathrm{l}\langle \tilde{v} \rangle & \quad \mid e[\cdot].\mathrm{l}\langle \tilde{a} \rangle \quad \mid o.\mathrm{l}\langle \tilde{v}, e[\cdot], \tilde{a} \rangle \\
v ::= o \mid t & \quad \mid o.\mathrm{l}\Leftarrow m & \quad \mid e[\cdot].\mathrm{l}\Leftarrow m \\
 & \quad \mid o.\mathsf{alias}\langle o' \rangle & \quad \mid e[\cdot].\mathsf{alias}\langle b \rangle \quad \mid o.\mathsf{alias}\langle e[\cdot] \rangle \\
m ::= \varsigma(s, \tilde{x})b & \quad \mid o.\mathsf{clone} & \quad \mid e[\cdot].\mathsf{clone} \\
\mathbb{O} ::= [\mathrm{l}_j = m_j]_{j \in J} & \quad \mid \mathsf{let}\, x = v \,\mathsf{in}\, b & \quad \mid \mathsf{let}\, x = e[\cdot] \,\mathsf{in}\, b \\
 & \quad \mid \mathsf{fork}(a) & \\
 & \quad \mid \mathsf{join}(t) & \quad \mid \mathsf{join}(e[\cdot]) \\
 & \quad \mid \mathsf{wait} &
\end{array}
$$

Figure 2: Syntax of Øjeblik run-time expressions

self variable $s$, and the actual parameters $a_1 \ldots a_n$ of the invocation bound to the formal parameters $\tilde{x}$. The expression $a.\mathrm{l}\Leftarrow m$ updates the content of the named field l in $a$ with method $m$ and evaluates to the modified object.

Every object in Øjeblik comes equipped with special methods for cloning and aliasing, which cannot be overwritten by the update operation. The operation $a.\mathsf{clone}$ creates an object with the same fields as the original object and initializes the fields to the same values as in the original object. The operation $a.\mathsf{alias}\langle b\rangle$ replaces the object $a$ with a pointer to $b$, regardless of whether $a$ is already a pointer or still an object record[2]. Thus, like cloning, the aliasing operation itself is not subject to aliasing. Consequently, the behavior of an Øjeblik-object can only be changed directly via method update or else indirectly by aliasing.

As usual, an expression $\mathsf{let}\, x = a \,\mathsf{in}\, b$ (only non-recursive) first evaluates $a$, binding the result to $x$, and then evaluates $b$ within the scope of the new binding. Moreover, $a; b$ abbreviates $\mathsf{let}\, x = a \,\mathsf{in}\, b$, where $x$ does not occur free in $b$.

To create a new concurrent thread we use the fork command. The expression $\mathsf{fork}(a)$ returns a thread identifier to denote a new thread evaluating $a$. To get the result of a computation in a fork'ed thread the join command is used. If $a$ evaluates to a thread identifier, then $\mathsf{join}(a)$ either returns the value that the thread denoted by $a$ has evaluated to, blocks until the thread finishes and then returns the resulting value, or blocks forever if a join on the $a$ thread has already been performed[3].

**Self-Infliction** The *current method* of the current thread is the last method invoked in it that has not yet completed. The *current self* is the self of the current method. Operations on Øjeblik-objects can happen in two ways: a *self-inflicted operation* is an operation that is performed on the current self; an operation is *external* if it is not self-inflicted.

**Serialization** In concurrent object-based settings, the invariant that at most one thread at a time may be active within an object is often called *serialization*. The simplest way to ensure serialization is to associate with an object a mutex that is locked when a thread enters the object and released when the thread exits the object. However, this approach is too restrictive—it prevents recursion. Based on the notion of *thread* as a strand of activity, so-called *reentrant* mutexes can be used to allow an operation to re-enter an object under the assumption that this operation belongs to the same thread as the operation that is currently active in the object. In Obliq, however, the more cautious idea of *self-serialization* 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 mutual recursion, where a method in an object $a$ calls a method in another object $b$, which then tries to 'call back' another method in $a$.

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

Note that in Obliq, the programmer specifies by means of keywords whether an object is protected or serialized. In this document, however, we assume for simplicity that all Øjeblik-objects are both protected and serialized.

## 3 An operational semantics for Øjeblik

We give a transitional semantics for Øjeblik terms that closely follows the one sketched by Talcott [Tal96]. Her semantics addresses a larger subset of Obliq than we do with Øjeblik, in particular including distribution concepts, but nevertheless excludes, for example, migration and join. We adapt the basic setup to our restricted variant Øjeblik.

### 3.1 Concepts

The semantics performs changes on run-time configurations, which are mappings from references $\mathcal{R}$ to run-time entities. More precisely, a configuration $\mathfrak{C}$ maps task references $t \in \mathcal{R}_\mathcal{T}$ and object references $o \in \mathcal{R}_\mathcal{O}$ to tasks $\mathcal{T}$ and objects $\mathcal{O}$, respectively, which we introduce below. We write $t, o \in \mathfrak{C}$, if $t, o$ are in the domain of $\mathfrak{C}$, and $\uparrow$ for undefined references.

A run-time expression $a$ is generated from the extended Øjeblik grammar as in Figure 2, where we introduce references as *values* $v$, as well as an additional construct wait, whose meaning will become clear from its use later on. Let us refer to this extended set of terms as $\mathcal{L}_\mathcal{R}$.

[2] Note that this is consistent with re-aliasing in Obliq.

[3] In Obliq, an exception will be raised.

A run-time object $O \in \mathcal{O}$ is either an object record $\mathbb{O}$ as generated by the extended grammar, or a pointer $\gg o$ to an object reference $o \in \mathcal{R}_{\mathcal{O}}$.

A run-time task $T$ is a triple $\langle f, s, a \rangle \in \mathcal{R}_{\mathcal{T}} \times \mathcal{R}_{\mathcal{O}} \times \mathcal{L}_{\mathcal{R}}$ that refers to a *father* $f$, a current *self* $s$, and a run-time Øjeblik expression $a$ that remains to be evaluated. By the partial functions $\mathrm{s}_{\mathfrak{C}}(t)$ and $\mathrm{f}_{\mathfrak{C}}(t)$, we refer to the—also possibly undefined—current self and father of the task associated with reference $t$ in $\mathfrak{C}$. As a well-formedness condition, we will only consider configurations, where each task is the father of at most one other task (fathers *and* children are unique), and if a task has a father reference, then this father is associated with a task within the configuration.

**Alias chains** Let the partial function $\mathrm{ali}_{\mathfrak{C}} : \mathcal{R}_{\mathcal{O}} \rightharpoonup \mathcal{R}_{\mathcal{O}}$ with

$$\mathrm{ali}_{\mathfrak{C}}(o) \stackrel{\mathrm{def}}{=} \begin{cases} \uparrow & \text{if } \mathfrak{C}(o) = \uparrow \\ o & \text{if } \mathfrak{C}(o) = \mathbb{O} \\ \mathrm{ali}_{\mathfrak{C}}(o') & \text{if } \mathfrak{C}(o) = \gg o' \end{cases}$$

compute the endpoint of an alias chain (starting at reference $o$), which is associated with an object record if it exists.

**Threads** We formalize the notion of *thread* derived from a task as the task's ancestors, similar to a call-stack in an implementation. Let $\mathrm{his}_{\mathfrak{C}} : \mathcal{R}_{\mathcal{T}} \to \mathcal{R}_{\mathcal{T}}{}^*$ with

$$\mathrm{his}_{\mathfrak{C}}(t) \stackrel{\mathrm{def}}{=} \begin{cases} \epsilon & \text{if } \mathfrak{C}(t) = \uparrow \\ t & \text{if } t \in \mathfrak{C} \text{ and } \mathrm{s}_{\mathfrak{C}}(t) = \uparrow \\ t \cdot \mathrm{his}_{\mathfrak{C}}(\mathrm{f}_{\mathfrak{C}}(t)) & \text{otherwise} \end{cases}$$

be the *history* of a task $t$, where $\cdot$ denotes concatenation of strings of references, and $\epsilon$ the empty string.

Furthermore, let $\mathrm{trc}_{\mathfrak{C}} : \mathcal{R}_{\mathcal{T}} \to \mathcal{R}_{\mathcal{O}}{}^*$ with

$$\mathrm{trc}_{\mathfrak{C}}(t) \stackrel{\mathrm{def}}{=} \begin{cases} \epsilon & \text{if } \mathfrak{C}(t) = \uparrow \\ \epsilon & \text{if } t \in \mathfrak{C} \text{ and } \mathrm{s}_{\mathfrak{C}}(t) = \uparrow \\ \mathrm{s}_{\mathfrak{C}}(t) \cdot \mathrm{trc}_{\mathfrak{C}}(\mathrm{f}_{\mathfrak{C}}(t)) & \text{otherwise} \end{cases}$$

be the *trace* of a task, which represents the string of all object references that occur as the current self of the task and its ancestors. For example, with

$$\mathfrak{C} = \{t_0 := \langle \uparrow, \uparrow, a_0 \rangle, t_1 := \langle t_0, s_1, a_1 \rangle, t_2 := \langle \uparrow, \uparrow, a_2 \rangle\},$$

we get $\mathrm{his}_{\mathfrak{C}}(t_1) = t_1 t_0$ and $\mathrm{trc}_{\mathfrak{C}}(t_1) = s_1$. A task $\hat{t}$ is *current* in a configuration $\mathfrak{C}$, if it is defined in $\mathfrak{C}$, but not the father of any other task in $\mathfrak{C}$. The threads of a configuration $\mathfrak{C}$ are identified as the set of histories of the current tasks in $\mathfrak{C}$:

$$\begin{aligned} \mathrm{cur}(\mathfrak{C}) &\stackrel{\mathrm{def}}{=} \{\ \hat{t} \in \mathcal{R}_{\mathcal{T}} \mid \hat{t} \text{ is current in } \mathfrak{C}\ \} \\ \mathrm{thr}(\mathfrak{C}) &\stackrel{\mathrm{def}}{=} \{\ \mathrm{his}_{\mathfrak{C}}(\hat{t}) \mid \hat{t} \in \mathrm{cur}(\mathfrak{C})\ \} \end{aligned}$$

In the above example, there are just two threads $\{t_1 t_0, t_2\}$. Finally, a task is *active*, if its expression is not a value.

**Self-Infliction** According to the need to test for the either self-inflicted or external character of operations on objects, we introduce some suitable notation. An object reference $o$ is *idle* in $\mathfrak{C}$, if it is not the current self of any task in $\mathfrak{C}$.

$$\begin{aligned} \mathrm{Idle}_{\mathfrak{C}}(o) &\stackrel{\mathrm{def}}{=} \textstyle\bigwedge_{t \in \mathcal{R}_{\mathcal{T}} \cap \mathfrak{C}} \ (o \neq \mathrm{s}_{\mathfrak{C}}(t)) \\ \mathrm{Avail}_{\mathfrak{C}}(o, t) &\stackrel{\mathrm{def}}{=} \mathrm{Idle}_{\mathfrak{C}}(o) \vee (o = \mathrm{s}_{\mathfrak{C}}(t)) \end{aligned}$$

An object reference $o$ is *available* for task $t$ in $\mathfrak{C}$, if $o$ is either idle or identical with the current self of task $t$.

**Evaluation** Figure 2 contains grammars for generating *redexes* $r$ and *evaluation contexts* $e[\cdot]$, which we use to control the evaluation of (the expression part of) run-time tasks. The contexts are designed such that evaluation proceeds leftmost-innermost. A simple algorithm can compute for every run-time expression $a \notin \mathcal{R}$ a unique pair of redex $r$ and context $e[\cdot]$ such that $a = e[r]$.

**Behaviors** The semantics $[\![\, a \,]\!]$ of a *closed* Øjeblik term $a \in \mathcal{L}$ is given by assigning $\{t_{\mathrm{m}} := \langle \uparrow, \uparrow, a \rangle\}$ as its initial configuration. Note that the task associated with $t_{\mathrm{m}}$ is both current and active; it represents the start of a *main* thread. The behavior of configurations is generated from the transition rules in Figures 3–6. In each case we pick some task and object references in a particular configuration $\mathfrak{C}$, which, under the respective conditions may enable a transition to take place in $\mathfrak{C}$. In the premises, note that the expressions of tasks are always in unique context-redex decomposed form. In the conclusions of the rules, the notation $\mathfrak{C}\{t := T, o := O\}$ means that the mapping $\mathfrak{C}$ is either extended or overwritten with the association of task reference $t$ with task $T$, and object reference $o$ with run-time object $O$.

The rules in Figure 3 describe the local activity in a single task $t$ in a straightforward manner; recall that let is not recursive and that the value $v$ is either a task or an object reference whose actual run-time entity is accessible through $\mathfrak{C}$, when needed.

The rules in Figure 4 have in common that they address protected operations, which can only happen if they are self-inflicted—so the premise $o = s$ is required. Note also that the effects of (Ali) and (Cln) do not depend on whether the current object is aliased or non-aliased, while this is important for (Upd), where the check for self-infliction is only carried out on the endpoint $\hat{o}$ of a (possible) alias chain starting with $o$. The intermediate nodes are thus treated in a rather liberal manner w.r.t. protection and serialization (cf. the detailed discussion in § 4 below).

The rules in Figure 5 formalize the protocol of synchronous method invocation, where each call creates a subtask as a child of the current task and blocks the father—by means of wait—until the subtask $t'$ signals its completion to its father $t$ in rule (Ret). Note the use of the predicate $\mathrm{Avail}_{\mathfrak{C}}(o)$ in rule (Inv) which is capturing the fact that a method call can be enabled—again checked for the endpoint $\hat{o}$ of an alias chain—either externally or by means of self-infliction.

The rules in Figure 6 exhibit that the interplay of fork-and-join is dual to the invoke-and-return game: in rule (Join), it is the father who knows his child by name, while in rule (Ret), it is the child knowing his father by name. This distinction is crucial since we derive the notion of thread by collecting references along the father pointers. Forked tasks do not know their father, so they represent initial tasks of new threads.

Finally, note that for simplicity we have not generated run-time errors in the above semantics for the cases of invalid access to protected operations; here, the calls to such operations will just block forever.

### 3.2 Theory

Based on the operational semantics, we may now formally define—and prove—in a straightforward manner the concept of self-serialization for Øjeblik expressions. We may also define a behavioral notion of program equivalence.

$$\frac{\mathfrak{C}(t) = \langle\, f, s, e[\mathsf{let}\, x = v\, \mathsf{in}\, b]\,\rangle}{\mathfrak{C} \;\to \mathfrak{C}\{t := \langle\, f, s, e[b\{^{v}/_{x}\}]\,\rangle\}} \quad (\text{Let})$$

$$\frac{\mathfrak{C}(t) = \langle\, f, s, e[\mathbb{O}]\,\rangle \qquad o \notin \mathfrak{C}}{\mathfrak{C} \;\to \mathfrak{C}\{t := \langle\, f, s, e[o]\,\rangle, o := \mathbb{O}\}} \quad (\text{New})$$

Figure 3: Local transitions

$$\frac{\begin{array}{c}\mathfrak{C}(t) = \langle\, f, s, e[o.\mathsf{alias}\langle o'\rangle]\,\rangle \qquad s = o \\ o \in \mathfrak{C} \qquad o' \in \mathfrak{C}\end{array}}{\mathfrak{C} \;\to \mathfrak{C}\{t := \langle\, f, s, e[o']\,\rangle, o := \gg o'\}} \quad (\text{Ali})$$

$$\frac{\begin{array}{c}\mathfrak{C}(t) = \langle\, f, s, e[o.\mathsf{clone}]\,\rangle \qquad s = o \\ o \in \mathfrak{C} \qquad o' \notin \mathfrak{C}\end{array}}{\mathfrak{C} \;\to \mathfrak{C}\{t := \langle\, f, s, e[o']\,\rangle, o' := \mathfrak{C}(o)\}} \quad (\text{Cln})$$

$$\frac{\begin{array}{ll}\mathfrak{C}(t) = \langle\, f, s, e[o.\mathrm{l}{\Leftarrow}m]\,\rangle & \mathrm{ali}_{\mathfrak{C}}(o) = \hat{o} = s \\ \mathfrak{C}(\hat{o}) = [\mathrm{l}_j{=}m_j]_{j\in J} & \mathrm{l} = \mathrm{l}_k \text{ for } k{\in}J\end{array}}{\begin{array}{l}\mathfrak{C} \;\to \mathfrak{C}\{\, t := \langle\, f, s, e[\hat{o}]\,\rangle \\ \qquad\qquad o := [\mathrm{l}_k{=}m, \mathrm{l}_{j\neq k}{=}m_j]_{j\in J}\}\end{array}} \quad (\text{Upd})$$

Figure 4: Protected transitions

$$\frac{\begin{array}{ll}\mathfrak{C}(t) = \langle\, f, s, e[o.\mathrm{l}\langle\,\tilde{v}\,\rangle]\,\rangle & \mathrm{ali}_{\mathfrak{C}}(o) = \hat{o} \\ \mathfrak{C}(\hat{o}) = [\mathrm{l}_j{=}\varsigma(s_j, \tilde{x})b_j]_{j\in J} & \mathrm{l} = \mathrm{l}_k \text{ for } k{\in}J \\ \mathrm{Avail}_{\mathfrak{C}}(\hat{o}, s) & t' \notin \mathfrak{C}\end{array}}{\begin{array}{l}\mathfrak{C} \;\to \mathfrak{C}\{\, t := \langle\, f, s, e[\mathsf{wait}]\,\rangle \\ \qquad\qquad t' := \langle\, t, \hat{o}, b_k\{^{\hat{o}\tilde{v}}/_{s\tilde{x}}\}\,\rangle\}\end{array}} \quad (\text{Inv})$$

$$\frac{\begin{array}{c}\mathfrak{C}(t) = \langle\, f, s, e[\mathsf{wait}]\,\rangle \\ \mathfrak{C}(t') = \langle\, t, s', v\,\rangle\end{array}}{\mathfrak{C} \;\to \mathfrak{C}\{t := \langle\, f, s, e[v]\,\rangle, t' := \uparrow\}} \quad (\text{Ret})$$

Figure 5: Method transitions

$$\frac{\mathfrak{C}(t) = \langle\, f, s, e[\mathsf{fork}(a)]\,\rangle \qquad t' \notin \mathfrak{C}}{\mathfrak{C} \;\to \mathfrak{C}\{t := \langle\, f, s, e[t']\,\rangle, t' := \langle\, \uparrow, \uparrow, a\,\rangle\}} \quad (\text{Fork})$$

$$\frac{\begin{array}{c}\mathfrak{C}(t) = \langle\, f, s, e[\mathsf{join}(t')]\,\rangle \\ \mathfrak{C}(t') = \langle\, \uparrow, \uparrow, v\,\rangle\end{array}}{\mathfrak{C} \;\to \mathfrak{C}\{t := \langle\, f, s, e[v]\,\rangle, t' := \uparrow\}} \quad (\text{Join})$$

Figure 6: Concurrency transitions

**Definition 3.1 (Serializability).** *A configuration $\mathfrak{C}$ is*

- serialized, *if for all $\widehat{t_1}, \widehat{t_2} \in \mathrm{cur}(\mathfrak{C})$:*
  *if $\mathrm{trc}_{\mathfrak{C}}(\widehat{t_1}) \cap \mathrm{trc}_{\mathfrak{C}}(\widehat{t_2}) \neq \emptyset$, then $\widehat{t_1} = \widehat{t_2}$.*
- self-serialized, *if, in addition, for all $t_n \cdots\cdot t_0 \in \mathrm{thr}(\mathfrak{C})$:*
  *if $\mathrm{s}_{\mathfrak{C}}(t_i) = \mathrm{s}_{\mathfrak{C}}(t_j)$ for $0 \leq i < j \leq n$,*
  *then $\mathrm{s}_{\mathfrak{C}}(t_i) = \mathrm{s}_{\mathfrak{C}}(t_{i+1}) = \ldots = \mathrm{s}_{\mathfrak{C}}(t_{j-1})$.*

Serialization means that each object is inhabited by at most one thread. Self-serialization adds the requirement that whenever a thread successfully re-calls an object it has actually never left the object since its first visit.

Let $\to^*$ denote the reflexive-transitive closure of the transition relation on Øjeblik-configurations.

**Proposition 3.2 (Soundness).** *For each Øjeblik term $a$, whenever $[\![\, a \,]\!] \to^* \mathfrak{C}$, then $\mathfrak{C}$ is self-serialized.*

*Proof.* By induction on the length of transition sequences, exploiting the premises concerning self-infliction and availability of objects. The fact that the self or father of a task once added in a configuration is never changed, is also used.

The only rule cases of interest are, when tasks are *added*, which is for (Inv) and (Fork), because only then the invariant may be broken. The actual proof in these cases is mere algebra.

If tasks are *removed* from the current configuration, as with (Ret) and (Join), then tasks are always only removed from the top of threads. Moreover, threads are never split up into two, so there is no danger of possibly invalidating the invariant that way.

If we neither add nor remove tasks, then the invariant holds trivially. □

Based on a *may*-variant[4] of convergence [Mor68], we will define a contextual notion of program equivalence.

**Definition 3.3 (Convergence).** *If $a$ is an Øjeblik term, then $a{\Downarrow}$ if $[\![\, a \,]\!] \to^* \mathfrak{C}$ with $\mathfrak{C}(t_{\mathrm{m}}) = \langle\, \uparrow, \uparrow, v\,\rangle$ for some $v$.*

Note that this notion of convergence does not mean that the whole computation of $a$ terminates, but rather that the main thread $t_{\mathrm{m}}$ does so: the evaluation of $a$ may converge to a value $v$ and can be reached in finite time within $t_{\mathrm{m}}$. Note that there might be other fork'ed computations that were not join'ed and might still be running, possibly forever.

In order to define a contextual notion of program equivalence, we also need a general notion of program context that differs from the notion of evaluation context given earlier. More specifically, according to Figure 3.2, an *object context* $C[\cdot]$ has a single hole $[\cdot]$ that may be filled with an Øjeblik term that possibly evaluates to an object; in particular, the hole must not be filled with a term that evaluates to a thread identifier. So, a composition $C[a]$ of an object context $C[\cdot]$ and an object expression $a$ is *well-formed*[5], if $a$ does not evaluate to a thread identifier. In the following, we implicitly consider only well-formed compositions.

**Definition 3.4 (Equivalence).** *Two terms $a, b \in \mathcal{L}$ are observationally equivalent, written $a \cong b$, if for all contexts $C[\cdot]$ with $\mathrm{fv}(C[a]){=}\mathrm{fv}(C[b]){=}\emptyset$ : $C[a] \Downarrow$ iff $C[b] \Downarrow$.*

[4] In the context of a concurrent language with fork, where threads may nondeterministically affect the outcome and convergence of evaluation, and with respect to our goal of reasoning about surrogation, we regard *must*-variants of convergence as too strong.

[5] In a typed variant of Øjeblik, *well-typed* composition takes over.

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

Figure 7: Øjeblik object contexts

### 3.3 Examples

As an abbreviation, we use the method definitions l=id and k=Ω to denote $\mathrm{l}{=}\varsigma(s)s$ and $\mathrm{k}{=}\varsigma(s)s.\mathrm{k}$, respectively, which obviously satisfy the properties $[\,\mathrm{l}{=}\mathsf{id}\,].\mathrm{l}\Downarrow$ and $[\,\mathrm{k}{=}\Omega\,].\mathrm{k}\not\Downarrow$.

**Cycles** Note that Øjeblik does not prevent the programmer from introducing self-aliases or alias chains with cycles, e.g.

$$
\begin{array}{l}
\mathsf{let}\ x = [\,\mathrm{k}{=}\mathsf{id}\,,\ \mathrm{l}{=}\varsigma(s,z)s.\mathsf{alias}\langle z\rangle\,]\ \mathsf{in}\\
\mathsf{let}\ y = x\ \mathsf{in}\\
x.\mathrm{l}\langle y\rangle; x.\mathrm{k}
\end{array}
\tag{1}
$$

In the semantics, after carrying out the aliasing operation, by means of the call to $x.\mathrm{l}\langle y\rangle$, yielding configuration $\mathfrak{C}$, the call to $x.\mathrm{k}$ results in the function $\mathrm{ali}_{\mathfrak{C}}$ not returning an argument. As a consequence, every operation on an object in an alias chain with cycles will block unless one of the objects breaks the cycle by means of re-aliasing (see also § 4).

**Self-infliction via substitution** Protected operations can be called from within methods not only literally on the self variable $s$, but also indirectly by an expression—for example an object variable—that evaluates to the object itself:

$$
\begin{array}{rl}
C[\cdot] := & \mathsf{let}\ x\ = [\,\mathrm{l}{=}\varsigma(s,z)z.\mathsf{clone}\,]\ \mathsf{in}\\
& \mathsf{let}\ y\ = [\cdot]\ \mathsf{in}\\
& x.\mathrm{l}\langle y\rangle
\end{array}
\tag{2}
$$

The behavior of $C[x]$ is error-free: the cloning operation may take place since $z$ is replaced with $x$ such that the call $z.\mathsf{clone}$ is self-inflicted (see also § 5.2).

## 4 Models for alias chains

With respect to the semantics of the alias operation, there is some freedom on how to precisely model serialization and protection in the aliased object. There are several possible variants, some of which we list in order of strength:

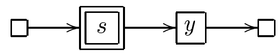
- a *conservative model* that keeps all properties,
- a *relaxed model* that only partially keeps them,
- an *ignorant model* that ignores them completely.

According to Cardelli's intuitive semantics [Car95, Car98], where object aliasing is derived from field aliasing, Obliq adopts the conservative model. A reasonable explanation is that, there, the aliased object still exists unchanged, but only accesses to its fields are redirected. The remainder of this section shall provide some operational understanding of the underlying concepts and differences of the above models.

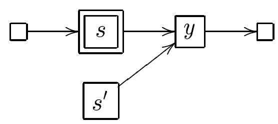
It is helpful to clarify the structure and requirements of aliased objects in an alias chain. In particular, we try to distinguish the cases of external vs internal requests that need to be dealt with in aliased objects. Note that, in Obliq, there is no doubt about the forwarding of updates and invocations to the alias target, being them internal or external.

### 4.1 Inside alias chains

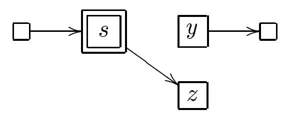
Considering the case of internal requests actually means that we look at an alias chain 'under construction'. In this case, there is at least one non-endpoint node in the chain, where some task is running and has just, in a self-inflicted way, connected itself by means of an alias operation to its current successor. Let this be the case for $s$ in



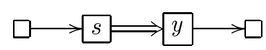

where it has just connected itself to $y$. Note the behavior of further possible self-inflicted actions of the current method in $s$: no update on $s$ may take place since it would already be forwarded to the $y$ (and maybe further). Invocations on $s$ would be treated in the same way, such that they will possibly be serialized in $y$ or beyond. However, self-inflicted cloning and aliasing on $s$ are successful, leading to one of




or




and such changes could continue further as long as the current method remains active. Once the method current in $s$ terminates, the alias $s{\gg}y$ becomes *stable*, as it can never be changed again in future computations.




Indeed, after an aliasing node has terminated its connection, every future incoming request will necessarily be external:

- for invocations and updates this is immediate, and
- for cloning and aliasing, this follows from the fact that the only possibility to start a self-inflicted cloning or aliasing—after the current method has finished—would be just within a method invoked *at* $s$, but any such will be forwarded to a successor of $s$, so it cannot affect $s$ itself.

### 4.2 Protection: Countering to the ignorant model

For external protection-critical access, like aliasing or cloning, in aliased objects, both the relaxed and conservative model produce run-time errors. For example, in the case of an external aliasing request to an already aliased object

$$
\begin{array}{l}
\mathsf{let}\ y_1\ = [\,\mathrm{l}{=}\mathsf{id}\,]\ \mathsf{in}\\
\mathsf{let}\ y_2\ = [\,\mathrm{l}{=}\mathsf{id}\,]\ \mathsf{in}\\
\mathsf{let}\ \ x\ \ = [\,\mathrm{l}{=}\varsigma(s,z)s.\mathsf{alias}\langle z\rangle\,]\ \mathsf{in}\\
x.\mathrm{l}\langle y_1\rangle; x.\mathsf{alias}\langle y_2\rangle
\end{array}
\tag{3}
$$

observe that the first call to $x.\mathrm{l}\langle y_1\rangle$ causes an internal aliasing operation on $x$, while the second afterwards tries to realias from the outside. While both the conservative and the relaxed model would—intuitively correct—block access to the call to $x.\mathsf{alias}\langle y_2\rangle$, the ignorant model would grant this external aliasing. This argument clearly suggests that protection should not be dropped completely, thus not to use the ignorant model, so we do not consider it further.

### 4.3 Serialization

The reason why one should keep serialization at all in aliased objects is, compared to protection, less obvious.

What may happen, when an external request enters a potentially serialized aliased object? Recall that, in Obliq, invocations and updates are always forwarded, while cloning and re-aliasing try to run where they are received. For the latter operations, the conservative and the relaxed model coincide and yield run-time errors. For the former operations, where forwarding is involved, the conservative and the relaxed model differ. It is crucial to clarify the interference between the act of forwarding that takes place in the source and the notion of current method for the moment when the forwarded request arrives at the target.

**Forwarding vs current self** For handling *external* requests that are to be forwarded by aliased objects, there are basically three strategies, which essentially rely on different underlying implementations of the object's mutex:

1. If the request is required to *grab* the mutex of the aliased object before being forwarded, then we should also assume that, afterwards, the current method is the one holding the mutex in the source.
2. If the request is *not* required to grab the mutex, then it is natural to assume that the current method after forwarding is the one that sent the request to the source.
3. If the request is just required to be able to *touch* the mutex, which amounts to grab and immediately release it, then it seems also natural to assume that the current method after forwarding is like in strategy 2.

Strategy 2 may be considered as safe under the assumption that aliasing is only carried out if the set of properties of the target subsume the ones of the source such that serialization and protection may be carried out there, if necessary. Strategy 3 lies in between the other two (cf. Example (5)).

Note further that if a request is self-inflicted at the source, then the current method should be as in strategy 2.

**Conservative vs relaxed model** Example (4) shows the case of an invocation, which is subject to serialization only:

$$\begin{array}{l}\mathsf{let}\ y\ = [\,\mathrm{k}{=}\mathsf{id}\,,\mathrm{l}{=}\varsigma(s,z)z.\mathrm{k}\,]\ \mathsf{in}\\ \mathsf{let}\ x\ = [\,\mathrm{k}{=}\mathsf{id}\,,\mathrm{l}{=}\varsigma(s,z)s.\mathsf{alias}\langle z\rangle\,]\ \mathsf{in}\\ x.\mathrm{l}\langle y\rangle; y.\mathrm{l}\langle x\rangle\end{array} \tag{4}$$

Observe that, in both models, the call $x.\mathrm{l}\langle y\rangle$ results in $x$ turning itself into an alias to $y$. Then, the call $y.\mathrm{l}\langle x\rangle$ results in a call to $x.\mathrm{k}$, which in turn is forwarded by $x$, now being an alias, back to $y$. There, it is treated differently:

- In the conservative model, which adopts strategy 1, the request arriving back at object $y$ would be blocked because there is already a method active in it, which is not the current one.
- In the relaxed model, which adopts strategy 2 (cf. rules (Ali), (Fig), and (Upd) in Figures 4 and 5), the request forwarded to $y$ is recognized as being self-inflicted since the current method is still the one active in $y$.

There may be reasons to prefer either of the possible design decisions. In our semantics, we chose the relaxed model to contrast it with Obliq's conservative model, over which it has some advantages, as will be explored later on. Yet, also the relaxed model has some deficiencies.

**A critique on the relaxed model** In the operational semantics of § 3, reminiscent of Talcott [Tal96], we mimic the relaxed model. According to the rules (Inv) and (Upd), invocations and updates are forwarded directly to the endpoint of an alias chain disregarding activities that potentially go on in intermediate nodes, only checking for serialization and protection (for update) in the endpoint. For example, in

$$\begin{array}{l}\mathsf{let}\ y\ = [\,\mathrm{l}{=}\mathsf{id}\,,\mathrm{k}{=}\mathsf{id}\,]\ \mathsf{in}\\ \mathsf{let}\ x\ = [\,\mathrm{l}{=}\mathsf{id}\,,\mathrm{k}{=}s.\mathsf{alias}\langle b\rangle; \ldots s \ldots]\ \mathsf{in}\\ \mathsf{fork}(x.\mathrm{l}); x.\mathrm{a}\end{array} \tag{5}$$

we concurrently call two different methods on $x$, both of which have the same chance of entering the object $x$.

If the call to $x.\mathrm{l}$ is first to enter, then there is no problem with the $x.\mathrm{k}$ getting in afterwards.

If, however, $x.\mathrm{k}$ is the first to enter, then after performing the aliasing operation resulting in a 'chain-under-construction' situation outlined in the beginning of § 4.1, where the other call to $x.\mathrm{l}$ might knock on $x$'s door. If there is no serialization going on in the *now* aliased object $x$, then this call might already (enter and) be forwarded, while $x$ is still active in performing operations of the current method k. These operations may be self-inflicted and the programmer of method k might have had in mind that no other method might enter or pass $x$, as long as method k is active, so there might be inconsistent intermediate states of $x$ that no other thread should be confronted with. To conclude, if the concept of serialization is to be interpreted rather strictly, external requests should be prevented from being processed in an aliased object as long as some computation is going on in it. So, the relaxed model of our semantics is not purely serialized (the semantics is too coarse-grained to exhibit this lack of serialization in a statement as of Definition 3.1).

The above discussion suggests that external requests should only be allowed to enter or pass an aliased object, when it has become stable, i.e., where requests are not in danger of passing the alias 'under construction'. It is this idea we had in mind when introducing strategy 3. Once the alias is stable, external requests can always touch-and-go.

**An relaxed interpretation: preentrant mutexes** The relaxed model ignores serialization in intermediate nodes of alias chains since an invocation or update to any of them is directly forwarded to the endpoint of the chain.

Regarding protection, the situation is slightly different. Whereas cloning and aliasing are checked and carried out in the source, updates are forwarded to be checked and carried out at the endpoint. We update Example (4) to this case:

$$\begin{array}{l}\mathsf{let}\ y\ = [\,\mathrm{k}{=}\mathsf{id}\,,\mathrm{l}{=}\varsigma(s,z)z.\mathrm{k}{\Leftarrow}\mathsf{id}\,]\ \mathsf{in}\\ \mathsf{let}\ x\ = [\,\mathrm{k}{=}\mathsf{id}\,,\mathrm{l}{=}\varsigma(s,z)s.\mathsf{alias}\langle z\rangle\,]\ \mathsf{in}\\ x.\mathrm{l}\langle y\rangle; y.\mathrm{l}\langle x\rangle\end{array} \tag{6}$$

In the conservative model, a forwarded update causes a run-time error. In the relaxed model, the update is forwarded to the endpoint where it is accepted as self-inflicted.

The behavior of the relaxed model in the above two examples could be interpreted as a relaxation of the concept of self-infliction towards reentrant mutexes. A thread may leave its current self behind to enter another object preceding it in an alias chain, from where it is forwarded to the endpoint of the chain. If the endpoint *is* the current self, then the thread may re-enter. Mutexes that allow this behavior may be called *pre-entrant* since they allow for self-invocation on endpoints via predecessors in an alias chain.

## 5 Surrogation in Øjeblik

In Øjeblik-objects, we use the special method $\mathsf{surrogate}$ as our abstraction for migration: by calling it, an object is turned into a (local) proxy for a (remote) copy of itself.

$$a, b, c \quad ::= \quad \ldots \mid a.\mathsf{surrogate}$$

According to our abstraction from migration to surrogation, Cardelli [Car95] would suggest that the response of Øjeblik objects to the call to $a.\mathsf{surrogate}$ can be implemented precisely and uniformly as the method $\varsigma(s)\mathsf{SUR}\langle s\rangle$ where:

$$\mathsf{SUR}\langle s\rangle \quad \stackrel{\text{def}}{=} \quad \mathsf{let}\, x = s.\mathsf{clone}\,\mathsf{in}\, s.\mathsf{alias}\langle x\rangle$$

Since left rather implicit in [Car95], we list the basic properties of surrogation, when implemented as a uniform method:

1. Surrogate methods shall not be updatable.
2. Surrogation shall be permitted for external requests.
3. Surrogation shall be forwarded by aliased objects.

Property 1 is reminiscent of cloning, aliasing, and updating, while properties 2 and 3 rather make it resemble invocation. Their justification is that surrogation mimics migration (although without resorting to explicit distribution), so an object should be surrogatable more than once. In this realm, double-surrogation $a.\mathsf{surrogate}; a.\mathsf{surrogate}$ should obviously be equivalent to $a.\mathsf{surrogate}.\mathsf{surrogate}$. Without forwarding, the migration of an already migrated object would mistakenly migrate the proxy.

$$\frac{\mathfrak{C}(t) = \langle\, f, s, e[o.\mathsf{surrogate}]\,\rangle \quad \mathrm{ali}_{\mathfrak{C}}(o) = \hat{o} \quad \mathrm{Avail}_{\mathfrak{C}}(\hat{o}, s) \quad t' \notin \mathfrak{C}}{\mathfrak{C} \;\to \mathfrak{C}\{\, t := \langle\, f, s, e[\mathsf{wait}]\,\rangle \;\; t' := \langle\, t, \hat{o}, \mathsf{SUR}\langle\hat{o}\rangle\,\rangle\}} \;(\textsc{Sur})$$

Figure 8: Surrogate transitions

The rule in Figure 8 precisely formalizes the semantics of $\mathsf{surrogate}$ according to the above properties as a uniform partially protected method in objects. The overall semantics of Øjeblik is still sound and also the definition of convergence and equivalence carries over smoothly.

The following statement is a first attempt to rephrase the title of this paper in terms of a formal semantics. The idea is that an object should behave the same in all contexts before and after surrogation, so we state an equation using the previously introduced notion of program equivalence.

**Guess 5.1 (Safety of surrogation).** *Let $a$ be an Øjeblik term. Then $a \cong a.\mathsf{surrogate}$.*

It turns out that this guess is rather naive and indeed wrong with our current semantics, so in the following paragraphs we narrow down the above equation in a way such that it becomes true, and provably so. Along the way, we give a few examples that highlight various problems of surrogation. Note that the following discussion also generalizes to *migration* in a distributed lexically-scoped setting, like Obliq.

The simplest case of Guess 5.1 is, where $a$ is an Øjeblik object $\mathbb{O}$. In this case the surrogation is surely safe because (1) the process of surrogation is carried out correctly since only the surrogation thread can interact with the object $\mathbb{O}$, i.e., there cannot be any interference with another thread or activity, and (2) every interaction with $\mathbb{O}$ is mimicked in exactly the same way by $\mathbb{O}.\mathsf{surrogate}$ and after surrogation nobody has access to the previous $\mathbb{O}$.

In the general case, however, neither of the two above arguments holds. In order to simplify the discussion, note that by means of $a \cong \mathsf{let}\, x = a\, \mathsf{in}\, x$ and the fact that the notion of equivalence takes all Øjeblik contexts into account, the statement of Guess 5.1 can be reduced to the equation $x \cong x.\mathsf{surrogate}$, thus reducing the problem to surrogation on variables. It is easy to see that $x\Downarrow$ iff $x.\mathsf{surrogate}\Downarrow$ since neither of them converges. Similarly, $C[x]\Downarrow$ iff $C[x.\mathsf{surrogate}]\Downarrow$ for all contexts $C[\cdot]$ that do not bind $x$. This implies that the interesting contexts are the ones binding $x$.

For the safety of surrogation, it is decisive, whether or not the call $x.\mathsf{surrogate}$ is an external or a self-inflicted one. Note that this is an undecidable problem, so we can only at run-time observe which case applies. In the following series of examples, we consider both cases separately.

### 5.1 External surrogation

The main message of this subsection is that, even with respect to our weak semantics based on may-convergence:

> External surrogation in Øjeblik is *not* safe!
> Migration in the Obliq [Car] is *not* safe!

... at least if the semantics or implementation is based on either the conservative or the relaxed alias chain model.

**Countering to the conservative alias model** The following object context $C[\cdot]$, a variant of Example (4), distinguishes an object variable $x$ and its surrogated counterpart $x.\mathsf{surrogate}$ in Cardelli's conservative alias model for Obliq:

$$\begin{aligned} C[\cdot] := \;& \mathsf{let}\ x \;= [\,\mathrm{k}{=}\mathsf{id}\,, \mathrm{l}{=}\varsigma(s,z)z.\mathrm{k}\,]\ \mathsf{in} \\ & \mathsf{let}\ y \;= [\cdot]\ \mathsf{in} \\ & y.\mathrm{l}\langle x\rangle \end{aligned} \qquad (7)$$

Regarding $C[x]$ both the conservative and the relaxed model yield a (even must-) convergent computation since the call to $y.\mathrm{l}\langle x\rangle$ is transformed to a self-inflicted call to $x.\mathrm{k}$.

In contrast, for $C[x.\mathsf{surrogate}]$, then this call arises from object $y$ different from object $x$ and comes back to $y$ as a forwarded call. For the same reason as in Equation (4), the relaxed model still yields (must-) convergence, in accordance with $C[x]$, while the conservative model blocks this call, always preventing the whole term from proper termination.

**Countering to the relaxed alias model** The following example, a variant of Example (2), distinguishes an object variable $x$ and its surrogated counterpart $x.\mathsf{surrogate}$ in Talcott's relaxed alias model for Obliq:

$$\begin{aligned} C[\cdot] := \;& \mathsf{let}\ x \;= [\,\mathrm{l}{=}\varsigma(s,z)z.\mathsf{clone}\,]\ \mathsf{in} \\ & \mathsf{let}\ y \;= [\cdot]\ \mathsf{in} \\ & y.\mathrm{l}\langle x\rangle \end{aligned} \qquad (8)$$

As already motivated in Example (2), the term $C[x]$ converges since the cloning inside $x$ is self-inflicted. This, however, is not the case for the term $C[x.\mathsf{surrogate}]$, for reasons similar to the previous Example (7): The call in the surrogation target $y$ to clone its predecessor $x$ is external to $x$.

**Promoting the forwarder model** The previous counterexamples obviously suggest that aliasing of objects should yield *pure forwarders for external requests*, not only for invocations, surrogations, and updates, but also—instead of generating run-time errors—for cloning and aliasing. In particular, like in the relaxed model, these external requests should be forwarded without affecting the identity of the current method. Otherwise, a surrounding context can be constructed that distinguishes variables $x$ from their externally surrogated counterparts $x.\mathsf{surrogate}$.

In comparison with the relaxed model, the forwarder model represents a proper generalization for the handling of external requests. For internal requests that enter aliased objects, i.e., as sketched at the end of Section 4, the situation is not different compared to the relaxed model. Note that the critique on the relaxed model that we articulated in § 4.3, and its correction by adopting forwarding strategy 3, is orthogonal to the generalization to the forwarder model, so the critique also applies to the latter.

If we want to adapt the operational semantics of § 3 to the forwarder model with strategy 3, we have to model

1. that for (Cln) and (Ali) forwarding to a non-endpoint successor—the current self—must be considered, and
2. that forwarding in (Cln), (Ali), (Upd), and (Inv), is only allowed via *stable* alias nodes.

We omit the straightforward formalization.

### 5.2 Self-inflicted surrogation

A particular class of examples is represented by objects that perform surrogation in a self-inflicted way such that they afterwards—as long as the current method is active—still can perform self-inflicted operations on the surrogated object. Due to these examples surrogation is *not completely* safe, even in a forwarder model. We classify two different sets of examples depending on whether they exhibit problems with access to a self-surrogated source or the target thereof.

**Target problems** A rather simple immediate source of problems is due to the incorrect external use of the surrogation target by means of protected operations, e.g., cloning:

$$C[\cdot] := [\,\mathrm{k}{=}\varsigma(s)\mathsf{let}\,y=[\cdot]\,\mathsf{in}\,y.\mathsf{clone}\,].\mathrm{k} \quad \text{yields} \quad C[s]\Downarrow \quad \text{and} \quad C[s.\mathsf{surrogate}]\not\Downarrow. \tag{9}$$

In $C[s]$ the cloning of $y$ is allowed, while in $C[s.\mathsf{surrogate}]$ the corresponding call is blocked due to a run-time error.

**Source problems** Similar to the previous example, a problem arises by externally sending a request to the surrogation target, but now via the surrogation source, e.g., for updates:

$$C[\cdot] := [\,\mathrm{k}{=}\varsigma(s)\mathsf{let}\,y=[\cdot]\,\mathsf{in}\,s.\mathrm{k}{\Leftarrow}\mathsf{id}\,].\mathrm{k} \quad \text{yields} \quad C[s]\Downarrow \quad \text{and} \quad C[s.\mathsf{surrogate}]\not\Downarrow. \tag{10}$$

By sending an update to itself after having surrogated, as in $C[s.\mathsf{surrogate}]$, the update of $y$ is blocked and results in a run-time error, while without previous surrogation the call succeeds and the whole term converges.

The next (and final) example is intended to exhibit the effect of re-aliasing after self-inflicted surrogation.

$$\begin{array}{l} C[\cdot] := \mathsf{let}\ x = [\,\mathrm{l}{=}\Omega\,,\mathrm{k}{=}\Omega\,]\ \mathsf{in} \\ \qquad\quad \mathsf{let}\ z = [\,\mathrm{l}{=}\mathsf{id}\,,\mathrm{k}{=}\varsigma(s)\ \mathsf{let}\ y=[\cdot] \\ \qquad\qquad\qquad\qquad\qquad\quad \mathsf{in}\ \ s.\mathsf{alias}\langle x\rangle; y.\mathrm{l}\,].\mathrm{k} \\ \qquad\qquad\qquad \text{yields} \\ \qquad C[s]\not\Downarrow \quad \text{and} \quad C[s.\mathsf{surrogate}]\Downarrow. \end{array} \tag{11}$$

Whereas $C[s]$ diverges since the alias call to $s$ also affects $y$ in that case, the counterpart $C[s.\mathsf{surrogate}]$ converges since the re-aliasing of $s$ does not affect the target $y$.

Note that it is the programmer of an object who is responsible for potential problems caused by self-inflicted surrogation. A programmer will hardly cause problems using self-surrogation if, in the current method, the self-variable $s$ is neither copied (to prevent from target problems), nor used after surrogation in a self-inflicted way for calls to state-changing methods (to prevent from source problems). The safest way to prevent from problems of self-inflicted surrogation is to use the self-variable in the current method for nothing else but surrogation.

### 5.3 A safety theorem

Apparently, since internal surrogation in Øjeblik cannot be safe in general, we concentrate our efforts on $x \cong x.\mathsf{surrogate}$ for the case that the surrogate operation is external to $x$. According to Definition 3.3, this means that the best safety theorem we can expect in Øjeblik is:

$$\begin{array}{c} \text{for all } C[\cdot] \ \text{ with } \ [\cdot] \text{ ``external for'' x:} \\ C[x]\Downarrow \ \text{ iff } \ C[x.\mathsf{surrogate}]\Downarrow. \end{array} \tag{12}$$

Since self-infliction of method calls is statically undecidable, unless the hole $[\cdot]$ is at top-level (i.e., in redex position), how can we state that the contexts under consideration shall be the ones that render the call to $x.\mathsf{surrogate}$ external?

1. Give a "dynamic" definition based on evaluation of contexts according to an operational semantics.
2. Give a sound syntactic definition of "external" contexts that provides a reasonably complete approximation.

Both attempts will fail based on our current operational semantics. Even if we manage to cut down the number of contexts under consideration, we do not see that a proof of the safety of surrogation based on our definition of convergence in the operational semantics is tractable. The main reasons are: (1) this semantics only works for closed terms, so only convergence of these can be investigated; (2) the semantics is not sufficiently compositional; (3) the semantics is in a non-syntactical configuration style such that derived configurations after some steps do not allow to reconstruct a corresponding term, thus it is impossible to describe the evaluation of contexts.

In previous work on Abadi and Cardelli's Imperative Object Calculus (IOC) [AC96a], equivalence between IOC terms is defined in a contextual way [GHL97] similar to Definition 3.3. It turned out that in many cases it is simpler to use a semantics by translation into a $\pi$-calculus to establish the equivalence between terms. The main advantage is the large number of equivalences and algebraic laws to reason about expressions. Building on these experiences we choose a similar path—IOC is a concurrency-free subset of Øjeblik—for establishing the safety of external surrogation.

## 6 $\pi$-calculus background

In the next section we give the semantics of Øjeblik as an encoding into some $\pi$-calculus. In Figure 9, we introduce a monadic asynchronous $\pi$-calculus [HT92, Bou92], equipped with: (i) name testing; (ii) labeled values, called *variants*, used recently for encoding objects [San98]; (iii) a case destructor construct over variants values; (iv) a wrong construct used to indicate that a run-time error has occurred.

| | |
|---|---|
| *Names* | |
| $p, q, r, x, y, z, m, s, i, t$ | |
| *Variant Tags* | |
| $\mathrm{l}$ | |
| *Values* | |
| $v \ ::= \ x$ | name |
| $\mid \ \mathrm{l}\_v$ | variant value |
| *Processes* | |
| $P \ ::= \ \mathbf{0}$ | nil process |
| $\mid \ p(x).\, P$ | input |
| $\mid \ \overline{p}v$ | output |
| $\mid \ P \mid P$ | parallel |
| $\mid \ (\boldsymbol{\nu} x)\, P$ | restriction |
| $\mid \ !\, p(x).\, P$ | replicated input |
| $\mid \ \mathsf{if}\ [p{=}q]\ \mathsf{then}\ P\ \mathsf{else}\ Q$ | name testing |
| $\mid \ \mathsf{case}\ v\ \mathsf{of}\ \{\mathrm{l}\_(x) : P\}^*$ | case destructor |
| $\mid \ \mathsf{wrong}$ | wrong |

Figure 9: Syntax of $\pi$-calculus

*Substitutions*, ranged over by $\sigma$, are functions from names to values; for an expression $e$ which could be either a process or a value, $e\sigma$ is the result of applying $\sigma$ onto $e$, with the usual renaming to avoid captures of bound names. Substitutions have a tighter syntactic precedence than the process operators. Parallel composition has the lowest precedence among the operators.

The labeled transition system is the usual one in the early style [ACS98], extended with straightforward rules for case analysis and wrong [KS98].

Transitions are of the form $P \xrightarrow{\mu} P'$, where *action* $\mu$ is: $\tau$ (interaction), $pv$ (free input), $\overline{p}v$ (free output), or $\overline{p}\langle(\boldsymbol{\nu} x)\, v\rangle$ (bound output, i.e., the emission of value $v$ containing a private name $x$ at $p$). In these actions, $p$ is the *subject* and $v$ the *object* part. Free and bound names (fn, bn) of actions and processes are defined as usual.

The relation $\Longrightarrow$ is the reflexive-transitive closure of $\xrightarrow{\tau}$. Moreover, we recall standard notions for weak transitions:

$$\xrightarrow{\widehat{\mu}} \stackrel{\text{def}}{=} \begin{cases} \xrightarrow{\mu} & \text{if } \mu \neq \tau \\ \xrightarrow{\tau} \cup = & \text{if } \mu = \tau \end{cases} \qquad \stackrel{\widehat{\mu}}{\Longrightarrow} \stackrel{\text{def}}{=} \Rightarrow \xrightarrow{\widehat{\mu}} \Rightarrow \qquad \stackrel{\mu}{\Longrightarrow} \stackrel{\text{def}}{=} \Rightarrow \xrightarrow{\mu} \Rightarrow$$

They are direct generalizations to our extended calculus.

### 6.1 Behavioral Equivalences

In an asynchronous scenario, it makes sense to provide a notion of observability that considers only output actions [ACS98]. Process $P$ has a *barb* at name $q$, written $P\downarrow_q$, if $P$ can perform an output action whose subject is name $q$. We write $P\Downarrow_q$, if there exists a $P'$ with $P \Longrightarrow P'$ and $P\downarrow_q$.

As regards behavioral notions of equivalence, we focus on bisimulation-based behavioral equivalences, precisely on *barbed bisimulation* [MS92]. It is well-known that barbed bisimilarity represents a uniform mechanism for defining behavioral equivalences in any process calculus possessing (i) a *reduction relation* and (ii) an *observability predicate* which simply detects the possibility of performing some observable action. Barbed bisimulation equips a global observer with a minimal ability to observe actions and/or process states but it is not a congruence. By closing barbed bisimulation under contexts we obtain a much finer relation.

**Definition 6.1 (Barbed bisimulation/congruence).** Barbed bisimulation*, written $\dot{\approx}$, is the largest symmetric relation on processes s.t. $P \dot{\approx} Q$ implies:*

1. *If $P \xrightarrow{\tau} P'$, then $\exists\, Q'$ with $Q \Longrightarrow Q'$ and $P' \dot{\approx} Q'$.*
2. *If $P\downarrow_p$ then $Q\Downarrow_p$.*

*Processes $P$ and $Q$ are* barbed congruent*, written $P \cong_{bc} Q$, if for each context $C[\cdot]$ it holds that $C[P] \dot{\approx} C[Q]$.*

The main inconvenience of barbed congruence is that it uses quantification over contexts in the definition, and this can make proofs of process equalities heavy. However, there exist several versions of labeled bisimulations, which are defined without context quantification, but which imply barbed congruence, especially in an asynchronous setting [ACS98].

**Definition 6.2 (Early bisimulation).** *Weak early bisimulation, written $\approx$, is the largest symmetric relation on processes s.t. if $P \approx Q$ and $P \xrightarrow{\mu} P'$, with $\mathsf{bn}(\mu) \cap \mathsf{fn}(Q) = \emptyset$, then there exists $Q'$ s.t. $Q \stackrel{\widehat{\mu}}{\Longrightarrow} Q'$ and $P' \approx Q'$.*

### 6.2 L$\pi$ laws

In the technical part of the paper, we make extensive use of some algebraic properties of the local asynchronous $\pi$-calculus, L$\pi$, due to Merro and Sangiorgi [MS98]. L$\pi$ is a variant of the asynchronous $\pi$-calculus, where only the output capability of names may be transmitted, that is, the recipient of a name may use it only in output actions.

Asynchronous names are L$\pi$ names if they cannot be tested and, whenever passed, they may be used by recipients only in output actions. L$\pi$ names have several interesting algebraic properties, of which the following two, Laws (13) and (14), will be useful in our proofs.

Let $C[\cdot]$ be a standard $\pi$-calculus process context. Let $p$ and $q$ be two L$\pi$ names with $p \neq q$, and $p$ does not appear free in input both in $P$ and $C[\cdot]$. Then

$$(\boldsymbol{\nu} p)(!p(x).\,\overline{q}x \mid P) \quad = \quad P\{^q/_p\} \qquad (13)$$

$$(\boldsymbol{\nu} p)(!p(x).\,P \mid C[\overline{p}v]) \quad = \quad (\boldsymbol{\nu} p)(!p(x).\,P \mid C[P\{^v/_x\}]) \qquad (14)$$

Law 13 is an optimization that allows us to replace a restricted forwarder $!\,p(x).\,\overline{q}x$ with a substitution. Such optimization laws are very useful in compiler environments. Law 14 resembles *inline expansion*, an optimization technique for functional languages that replaces a function call with a copy of the function body.

As already pointed out in [MS98], the algebraic properties of L$\pi$ can be applied also in calculi where the usage of some names goes beyond the syntax of L$\pi$. For instance, there could be synchronous names, or names that can be tested for identity. A type system can be used to distinguish between L$\pi$ names and the other names, and the theory of L$\pi$ can then be applied to the former names.

## 7 A $\pi$-calculus semantics for Øjeblik

In addition to the core $\pi$-calculus, we introduce some well-known syntactic sugar for presenting the encoding: we use (i) tuples $\tilde{x}$, also in labeled values $\mathrm{l}\_\langle\tilde{z}\rangle$ and patterns $\mathrm{l}\_(\tilde{z})$, and (ii) parameterized recursive definitions. Both can be faithfully represented in our core calculus, tuples in a type-safe manner by means of variants [San98], and recursion up to weak bisimulation in terms of replication [Mil93].

### 7.1 Concepts

In Figures 10–13, we present an encoding from Øjeblik-terms into $\pi$-calculus terms parameterized on two names. In an encoded Øjeblik term $[\![a]\!]_p^{i_\mathbf{c}}$, the parameters are used for returning its result ($p$), and for book-keeping its current self ($i_\mathbf{c}$).

**Objects** The basic structure of the encoding is similar to the one for the imperative object calculus IOC by Kleist and Sangiorgi [KS98]: the translation of an object $\mathbb{O}$, as shown in Figure 10, consists of a message that returns the object's reference $s$ on some result channel $p$, a composition of replicated processes that give access to the method bodies $[\![b_j]\!]_r^{i_\mathbf{c}}$, and a new object process $\mathsf{new.O}_\mathbb{O}\langle\, s, \tilde{t}\,\rangle$ that connects invocations on $s$ from the outside—via a series of *managers*—to the method bodies, which are invoked by the trigger names $\tilde{t}$.

**Self-Infliction** The main new idea of the translation is the distinction between two different self identifiers: an external self $s$ to receive requests, and an internal self $i$ to check for self-infliction. The crux for handling the latter resides on the protocol that each incoming request $\mathbf{c}$ carries with it a parameter $i_\mathbf{c}$ that identifies its current self, i.e., the internal self of the object from where the request originated. This parameter can then be matched against the internal self $i$ of the current object: if $[i_\mathbf{c}{\neq}i]$, then the request is regarded as external, i.e., originating from a different thread of control and, therefore, will be required to pass some serialization and protection mechanisms.

**Serialization** The mutual exclusion of activities within an object is implemented by a standard technique: a lock $m$, i.e., a message on a local *mutex* channel $m$ that obeys the invariant that at any time there is at most one message on it available. Whenever a new object $\mathsf{new.O}$ is created, a new mutex is also created, and initialized.

External requests ($[i_\mathbf{c}{\neq}i]$) are forced to grab the mutex before they are forwarded on $f$ to an object manager. After initialization, mutexes are only grabbed by serialization managers $\mathsf{SM}$ and released by the managers $\mathsf{OM}$ or $\mathsf{AM}$.

We distinguish between object managers ($\mathsf{OM}$) and alias managers ($\mathsf{AM}$) since they behave differently. Before we get into their functionality, we have a look on how standard *requests* are generated by the clients of objects.

**Clients** In Figure 11, the current-self parameter $i_\mathbf{c}$ of encoded terms is 'used'—just in object position—when clients pass it on together with their requests. In each case, the responsibility for returning a result on channel $p$ is forwarded to the respective object manager. Furthermore, each of the translations obeys the same idea: the involved expressions are evaluated at private locations $q$, their results are grabbed and then used to forward low-level requests to the corresponding object managers. We chose a sequential leftmost-innermost evaluation order for objects and parameters, which also guarantees that $i_\mathbf{c}$ can only be 'used' once at a time.

**Object managers** We first consider the case of self-inflicted requests $\mathbf{c}$ (where $[i_\mathbf{c}{=}i]$) that arrive in $\mathsf{OM}$ along name $f$, where no serialization or protection is required.

The basic functionality of the manager is to invoke the appropriate instances of method bodies (case $\mathrm{l}_j\_\mathsf{inv}$: activate the method body bound to $\mathrm{l}_j$ along trigger name $t_j$), and to carefully administrate updating (case $\mathrm{l}_j\_\mathsf{upd}$: install a new trigger name $t'$). Cloning (case $\mathsf{cln}$) results in the restart of the current object manager in parallel with a new object $\mathsf{new.O}_\mathbb{O}\langle\, s', \tilde{t}\,\rangle$, which can be considered a copy of the former since it uses the same method bodies $\tilde{t}$, but is accessible through a fresh reference $s'$. Aliasing (case $\mathsf{ali}$) is encoded by starting an appropriate aliased object manager $\mathsf{AM}$ instead of re-starting a previous non-aliased $\mathsf{OM}$. Requests for surrogation (case $\mathsf{sur}$) is simply translated in a compositional way as cloning followed by aliasing. Finally, also $\mathsf{ping}$-requests can be handled; they are not generated from clients in Figure 11, but will be introduced later on with the encoding of variables in Figure 12.

If serialization is required, i.e., in the case of external invocation and surrogation, we create a fresh return channel $r$ before we pass on the whole request, with the former result channel $p$ replaced by $r$. By that, we may block all further requests that need to serialize, until some result $y$ due to the current request is coming back on $r$. Only then, we are allowed to proceed by forwarding the result $y$ to the intended result channel $p$ and by releasing a signal on the mutex channel $m$ in order to let the next serialized request become active in the object. Note that a triggered method body $[\![b_j]\!]_r^{i}$ and the externally invoked surrogation body $[\![\mathsf{SUR}\langle s\rangle]\!]_r^{i}$ are both run in the context of the current internal self $i$, so their further calls to $s$ will be self-inflicted. This is of course essential for surrogation, since cloning and aliasing are only allowed internally.

If protection is required, i.e., in the case of cloning, aliasing, and update, we simply indicate a run-time error, whenever a modification request is attempted externally, and restart the object manager.

**Aliased managers** In accordance to our discussion in § 4 and 5, we intend to implement a forwarder model with forwarding strategy 3. An encoding in terms of the $\pi$-calculus provides just the right amount of granularity to make these ideas operationally precise. All *external* requests to an alias manager are forwarded—without modification of $i_\mathbf{c}$—to the aliasing target $s_\mathrm{a}$, its immediate successor.

In the self-inflicted cases, which represent an alias node 'under construction', cloning and aliasing are allowed and behave similar to the respective clauses in object managers, using $\mathsf{new.A}$ instead of $\mathsf{new.O}$. The requests for update, invocation, and surrogation are forwarded *as is* to the local alias target $s_\mathrm{a}$, where they will be considered external, since they are already self-inflicted in the current alias object.

**Variables** While the $\mathsf{let}$-construct is encoded in a standard manner (see [KS98]), variables are dealt with in a non-standard way: instead of mapping $[\![x]\!]_p^{i_\mathbf{c}}$ to $\overline{p}\langle x\rangle$, signaling immediately where the object associated with $x$ can be

$$
\begin{array}{rcl}
\mathbb{O} & \stackrel{\text{def}}{=} & [\mathrm{l}_j {=} \varsigma(s_j, \tilde{x}) b_j]_{j \in J} \\
\mathsf{SUR}\langle s \rangle & \stackrel{\text{def}}{=} & \mathsf{let}\, x = s.\,\mathsf{clone}\,\mathsf{in}\, s.\,\mathsf{alias}(x) \\
\hline
[\![\mathbb{O}]\!]_p^{i_{\mathbf{c}}} & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} s \tilde{t}) \left( \,\overline{p}\langle s \rangle \mid \mathsf{new.O}_{\mathbb{O}}\langle\, s, \tilde{t}\,\rangle \mid \prod_{j \in J} !\, t_j(r, i, s_j, \tilde{x}).\, [\![b_j]\!]_r^i \right) \\
\hline
\mathsf{new.O}_{\mathbb{O}}\langle\, s, \tilde{t}\,\rangle & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} imf) \left( \,\overline{m} \mid \mathsf{SM}_{\mathbb{O}}\langle\, s, i, m, f\,\rangle \mid \mathsf{OM}_{\mathbb{O}}\langle\, s, i, m, f, \tilde{t}\,\rangle \right) \\
\mathsf{new.A}_{\mathbb{O}}\langle\, s, s_{\mathrm{a}}\,\rangle & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} imf) \left( \,\overline{m} \mid \mathsf{SM}_{\mathbb{O}}\langle\, s, i, m, f\,\rangle \mid \mathsf{AM}_{\mathbb{O}}\langle\, s, i, m, f, s_{\mathrm{a}}\,\rangle \right) \\
\hline
\mathsf{SM}_{\mathbb{O}}\langle\, s, i, m, f\,\rangle & \stackrel{\text{def}}{=} & !\, s(\mathbf{c}).\, \mathsf{if}\ [i_{\mathbf{c}} {=} i]\ \mathsf{then}\ \overline{f}\langle \mathbf{c} \rangle\ \mathsf{else}\ m.\, \overline{f}\langle \mathbf{c} \rangle
\end{array}
$$

$$
\begin{array}{l}
\mathsf{OM}_{\mathbb{O}}\langle\, s, i, m, f, \tilde{t}\,\rangle \stackrel{\text{def}}{=} f(\mathbf{c}).\, \mathsf{if}\ [i_{\mathbf{c}} {=} i] \\
\quad \mathsf{then}\ \mathsf{case}\ \mathbf{c}\ \mathsf{of}\ \begin{array}[t]{rcl}
\mathsf{cln\_}(p_{\mathrm{c}}, i_{\mathbf{c}}) & : & \mathsf{OM}_{\mathbb{O}}\langle\, s, i, m, f, \tilde{t}\,\rangle \mid (\boldsymbol{\nu} s')\, (\,\overline{p_{\mathrm{c}}}\langle s' \rangle \mid \mathsf{new.O}_{\mathbb{O}}\langle\, s', \tilde{t}\,\rangle\,) \\
\mathsf{ali\_}(s_{\mathrm{a}}, p_{\mathrm{c}}, i_{\mathbf{c}}) & : & \mathsf{AM}_{\mathbb{O}}\langle\, s, i, m, f, s_{\mathrm{a}}\,\rangle \mid \overline{p_{\mathrm{c}}}\langle s_{\mathrm{a}} \rangle \\
\mathrm{l}_j\mathsf{\_upd\_}(t', p_{\mathrm{c}}, i_{\mathbf{c}}) & : & \mathsf{OM}_{\mathbb{O}}\langle\, s, i, m, t_1 \ldots t_{j-1}, t', t_{j+1} \ldots t_n\,\rangle \mid \overline{p_{\mathrm{c}}}\langle s \rangle \\
\mathrm{l}_j\mathsf{\_inv\_}(\tilde{x}, p_{\mathrm{c}}, i_{\mathbf{c}}) & : & \mathsf{OM}_{\mathbb{O}}\langle\, s, i, m, f, \tilde{t}\,\rangle \mid \overline{t_j}\langle p_{\mathrm{c}}, i, s, \tilde{x} \rangle \\
\mathsf{sur\_}(p_{\mathrm{c}}, i_{\mathbf{c}}) & : & \mathsf{OM}_{\mathbb{O}}\langle\, s, i, m, f, \tilde{t}\,\rangle \mid [\![\mathsf{SUR}\langle s \rangle]\!]_{p_{\mathrm{c}}}^i \\
\mathsf{ping\_}(p_{\mathrm{c}}, i_{\mathbf{c}}) & : & \mathsf{OM}_{\mathbb{O}}\langle\, s, i, m, f, \tilde{t}\,\rangle \mid \overline{p_{\mathrm{c}}}\langle s \rangle
\end{array} \\
\quad \mathsf{else}\ \ \mathsf{OM}_{\mathbb{O}}\langle\, s, i, m, f, \tilde{t}\,\rangle \mid \mathsf{case}\ \mathbf{c}\ \mathsf{of}\ \begin{array}[t]{rcl}
\mathsf{cln\_}(p_{\mathrm{c}}, i_{\mathbf{c}}) & : & \mathsf{wrong} \mid \overline{m} \\
\mathsf{ali\_}(s_{\mathrm{a}}, p_{\mathrm{c}}, i_{\mathbf{c}}) & : & \mathsf{wrong} \mid \overline{m} \\
\mathrm{l}_j\mathsf{\_upd\_}(t', p_{\mathrm{c}}, i_{\mathbf{c}}) & : & \mathsf{wrong} \mid \overline{m} \\
\mathrm{l}_j\mathsf{\_inv\_}(\tilde{x}, p_{\mathrm{c}}, i_{\mathbf{c}}) & : & (\boldsymbol{\nu} r)(\ \overline{t_j}\langle r, i, s, \tilde{x} \rangle \mid r(y).\, (\,\overline{p_{\mathrm{c}}}\langle y \rangle \mid \overline{m}\,)\,) \\
\mathsf{sur\_}(p_{\mathrm{c}}, i_{\mathbf{c}}) & : & (\boldsymbol{\nu} r)(\ [\![\mathsf{SUR}\langle s \rangle]\!]_r^i \mid r(y).\, (\,\overline{p_{\mathrm{c}}}\langle y \rangle \mid \overline{m}\,)\,) \\
\mathsf{ping\_}(p_{\mathrm{c}}, i_{\mathbf{c}}) & : & \overline{p_{\mathrm{c}}}\langle s \rangle \mid \overline{m}
\end{array} \\
\hline
\mathsf{AM}_{\mathbb{O}}\langle\, s, i, m, f, s_{\mathrm{a}}\,\rangle \stackrel{\text{def}}{=} f(\mathbf{c}).\, \mathsf{if}\ [i_{\mathbf{c}} {=} i] \\
\quad \mathsf{then}\ \ \mathsf{case}\ \mathbf{c}\ \mathsf{of}\ \begin{array}[t]{rcl}
\mathsf{cln\_}(p_{\mathrm{c}}, i_{\mathbf{c}}) & : & \mathsf{AM}_{\mathbb{O}}\langle\, s, i, m, f, s_{\mathrm{a}}\,\rangle \mid (\boldsymbol{\nu} s')\, (\,\overline{p_{\mathrm{c}}}\langle s' \rangle \mid \mathsf{new.A}_{\mathbb{O}}\langle\, s', s_{\mathrm{a}}\,\rangle\,) \\
\mathsf{ali\_}(s'_{\mathrm{a}}, p_{\mathrm{c}}, i_{\mathbf{c}}) & : & \mathsf{AM}_{\mathbb{O}}\langle\, s, i, m, f, s'_{\mathrm{a}}\,\rangle \mid \overline{p_{\mathrm{c}}}\langle s'_{\mathrm{a}} \rangle \\
\mathrm{l}_j\mathsf{\_upd\_}(t', p_{\mathrm{c}}, i_{\mathbf{c}}) & : & \mathsf{AM}_{\mathbb{O}}\langle\, s, i, m, f, s_{\mathrm{a}}\,\rangle \mid \overline{s_{\mathrm{a}}}\langle \mathbf{c} \rangle \\
\mathrm{l}_j\mathsf{\_inv\_}(\tilde{x}, p_{\mathrm{c}}, i_{\mathbf{c}}) & : & \mathsf{AM}_{\mathbb{O}}\langle\, s, i, m, f, s_{\mathrm{a}}\,\rangle \mid \overline{s_{\mathrm{a}}}\langle \mathbf{c} \rangle \\
\mathsf{sur\_}(p_{\mathrm{c}}, i_{\mathbf{c}}) & : & \mathsf{AM}_{\mathbb{O}}\langle\, s, i, m, f, s_{\mathrm{a}}\,\rangle \mid \overline{s_{\mathrm{a}}}\langle \mathbf{c} \rangle \\
\mathsf{ping\_}(p_{\mathrm{c}}, i_{\mathbf{c}}) & : & \mathsf{AM}_{\mathbb{O}}\langle\, s, i, m, f, s_{\mathrm{a}}\,\rangle \mid \overline{p_{\mathrm{c}}}\langle s \rangle
\end{array} \\
\quad \mathsf{else}\ \ \mathsf{AM}_{\mathbb{O}}\langle\, s, i, m, f, s_{\mathrm{a}}\,\rangle \mid \overline{s_{\mathrm{a}}}\langle \mathbf{c} \rangle \mid \overline{m}
\end{array}
$$

Figure 10: Encoding of Øjeblik-objects

$$
\begin{array}{rcl}
[\![a.\,\mathsf{clone}]\!]_p^{i_{\mathbf{c}}} & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} q)\, (\, [\![a]\!]_q^{i_{\mathbf{c}}} \mid q(y)\,.\, \overline{y}\, \mathsf{cln\_}\langle p, i_{\mathbf{c}} \rangle\,) \\
[\![a.\,\mathsf{alias}(b)]\!]_p^{i_{\mathbf{c}}} & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} q_x q_y)\, (\, [\![a]\!]_{q_y}^{i_{\mathbf{c}}} \mid q_y(y)\,.\, ([\![b]\!]_{q_x}^{i_{\mathbf{c}}} \mid q_x(x)\,.\, \overline{y}\, \mathsf{ali\_}\langle x, p, i_{\mathbf{c}} \rangle)\,) \\
[\![a.\,\mathrm{l}_j \Leftarrow \varsigma(s, \tilde{x}) b]\!]_p^{i_{\mathbf{c}}} & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} q) \left( [\![a]\!]_q^{i_{\mathbf{c}}} \mid q(y)\,.\, (\boldsymbol{\nu} t)\, (\, !\, t(r, i, s, \tilde{x}).\, [\![b]\!]_r^i \mid \overline{y}\, \mathrm{l}_j\mathsf{\_upd\_}\langle t, p, i_{\mathbf{c}} \rangle\,)\, \right) \\
[\![a.\,\mathrm{l}_j\langle\, a_1 \ldots a_n\,\rangle]\!]_p^{i_{\mathbf{c}}} & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} q)\, (\boldsymbol{\nu} q_1 \cdots q_n) \left( [\![a]\!]_q^{i_{\mathbf{c}}} \mid q(y).\, ([\![a_1]\!]_{q1}^{i_{\mathbf{c}}} \mid q_1(x_1).\, (\ldots q_n(x_n).\, \overline{y}\, \mathrm{l}_j\mathsf{\_inv\_}\langle x_1 \ldots x_n, p, i_{\mathbf{c}} \rangle))\right) \\
[\![a.\,\mathsf{surrogate}]\!]_p^{i_{\mathbf{c}}} & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} q)\, (\, [\![a]\!]_q^{i_{\mathbf{c}}} \mid q(y)\,.\, \overline{y}\, \mathsf{sur\_}\langle p, i_{\mathbf{c}} \rangle\,)
\end{array}
$$

Figure 11: Encoding of Øjeblik-clients

$$\boxed{\begin{array}{rcl} [\![\mathsf{let}\, x = a\, \mathsf{in}\, b]\!]_p^{i_c} & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} q)\,\big(\, [\![a]\!]_q^{i_c} \;\mid\; q(x)\,.\,[\![b]\!]_p^{i_c}\,\big) \\ [\![x]\!]_p^{i_c} & \stackrel{\text{def}}{=} & \overline{x}\,\mathsf{ping}\_\langle p, i_c\rangle \end{array}}$$

Figure 12: Encoding of Øjeblik-variables

found, in Figure 12, we require variables to briefly interact with the run-time entity that they refer to by sending to it a ping-request. The intuition is that the referred entity simulates the usual signaling protocol, but refines it according to the context of self-infliction and aliasing.

According to our encoding of managers OM and AM, a ping-request is replied immediately if it is self-inflicted, and it is forwarded, when it is external in an alias node. Note that this design possibly delays the signaling protocol for external variable access until the object that it is referring to (via aliasing) has finished its current method. Consequently, it blocks the protocol if this current method diverges.[6] We defer a further explanation to § 7.2.

**Concurrency** To fork a thread means to create a new activity running in parallel with the current one(s). In the $\pi$-calculus, where we have a parallel operator, it suffices in addition to create a fresh self identifier upon thread creation and to implant it as the forked threads' current self. In Figure 13, we use $[\![a]\!]_q^{\boldsymbol{\nu} i}$ to abbreviate $(\boldsymbol{\nu} i)\,([\![a]\!]_q^{i})$.

$$\boxed{\begin{array}{rcl} [\![\mathsf{fork}(a)]\!]_p^{i_c} & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} q, t)\,\big(\, [\![a]\!]_q^{\boldsymbol{\nu} i} \;\mid\; \overline{p}\langle t\rangle \;\mid\; \mathsf{TM}\langle q, t\rangle\,\big) \\ \mathsf{TM}\langle q, t\rangle & \stackrel{\text{def}}{=} & t(x).\,\mathsf{case}\; x\; \mathsf{of} \\ & & \quad \mathsf{ping}\_(p_c, i_c) : \mathsf{TM}\langle q, t\rangle \;\mid\; \overline{p_c}\langle t\rangle \\ & & \quad \mathsf{join}\_(p_c, i_c) : \mathsf{TM}^D\langle t\rangle \;\mid\; q(y).\,\overline{p_c}\langle y\rangle \\ \mathsf{TM}^D\langle t\rangle & \stackrel{\text{def}}{=} & t(x).\,\mathsf{case}\; x\; \mathsf{of} \\ & & \quad \mathsf{ping}\_(p_c, i_c) : \mathsf{TM}^D\langle t\rangle \;\mid\; \overline{p_c}\langle t\rangle \\ & & \quad \mathsf{join}\_(p_c, i_c) : \mathsf{TM}^D\langle t\rangle \;\mid\; \mathsf{wrong} \\ \hline [\![\mathsf{join}(b)]\!]_p^{i_c} & \stackrel{\text{def}}{=} & (\boldsymbol{\nu} q)\,\big(\, [\![b]\!]_q^{i_c} \;\mid\; q(t)\,.\,\overline{t}\,\mathsf{join}\_\langle p, i_c\rangle\,\big) \end{array}}$$

Figure 13: Encoding of Øjeblik-concurrency

Note that a fork is never blocking: we immediately return a private name $t$, which can be used to get back some result (the evaluation of the forked expression $a$) from the forked thread through interaction with a thread manager TM.

The thread manager is inquired in the translation of join($b$) by sending its continuation $p$ to the thread manager along $t$. Although term $b$ may directly evaluate to some thread identifier, it may also evaluate to a variable that is bound to the thread identifier. In the latter case, the variable does not itself reply along $q$, but instead is sending a ping-request along $t$. Therefore, the thread manager has to simulate the variable's signaling protocol, and then to tell the join'ing thread where to send its continuation to.

[6]Thus, a variable *looks ahead*, before it is used in a computation thread: if its referred object is ping'able now, then only concurrent (i.e., fork'ed) computations may prevent its subsequent use for sending requests—in the realm of may-convergence, this tells us that terminating requests sent to the ping'ed variable may converge.

### 7.2 Theory

Based on the notion of barbs of § 6, and following the intuition that an encoded term $[\![a]\!]_p^{i_c}$ eventually tells its result on name $p$, we may straightforwardly define a new notion of convergence for Øjeblik-terms using the $\pi$-calculus encoding.

**Definition 7.1 (Convergence).** *If $a$ is an Øjeblik term, then $a\Downarrow$ if $[\![a]\!]_p^{\boldsymbol{\nu} i}\Downarrow_p$ for any $p$.*

Accordingly, an Øjeblik term converges, if its translation may report its result on the name $p$. Note that this definition in principle even applies to open Øjeblik terms, e.g., to an Øjeblik variable $x$. However, our $\pi$-calculus semantics is designed to prevent $[\![x]\!]_p^{\boldsymbol{\nu} i}$ from immediately reporting on $p$; it is required to send a ping-request along $x$ such that a convergence signal is only possible if the ping-request reaches an object, alias, or thread manager.

Immediately, by inserting the $\pi$-calculus based notion of convergence into Definition 3.3, we get a $\pi$-calculus based notion of equivalence on Øjeblik terms. We may even go further and extend also this notion to open terms since the encoding treats variables carefully, which is important, when we compare variables to their surrogated counterpart.

For the remainder of the paper, the symbol $\cong$ shall denote the behavioral equivalence based on the notion of convergence that arises from the $\pi$-calculus semantics.

### 7.3 Comparing to the operational semantics

The $\pi$-calculus semantics presented in this paper corrects and simplifies our previous approach [HKNS98], which turned out to be too restrictive for expressions that, after having changed an object, keep working on it in a self-inflicted manner, like the examples of self-inflicted surrogation in § 5.2. The current $\pi$-calculus semantics implements our proposal for an ideal semantics for Øjeblik, based on the *forwarder* model for alias chains, together with forwarding strategy 3, as introduce on page 6.

In general, whenever one has two different semantics at hand that address the same language, one should carefully argue—best in a formal way—for their consistence. In the context of the $\pi$-calculus the notion of *operational correspondence*, a kind of mutual simulation between computation steps in the source and target language of the translation, is often used to this end. In the case of our $\pi$-calculus semantics, in comparison with some appropriate variant of the operational semantics, we expect such a correspondence to be provable, but we have not yet carried out the necessary work. Here's an informal sketch of the correspondence:

- Whereas the operational semantics can only be given for closed terms, also the operational correspondence can only address these.
- For a formal correspondence, the operational semantics should also include error-reporting rules instead of just blocking such that also error-cases can be considered.
- The main obstacle for the correspondence arises from the global-state point of view of the operational semantics that allows it to do several forwarding steps along alias chains at once. In contrast, in the $\pi$-calculus, each individual forwarding step is a proper computation step. This implies that the occurrence of failures may be delayed in the $\pi$-calculus semantics.

Based on the operational correspondence, we conjecture that the two notions of convergence for Øjeblik terms coincide.

## 8 Proving the safety of external surrogation

In § 3, we derived a notion of behavioral equivalence for Øjeblik terms based on a SOS semantics. In § 7, by translation into $\pi$-calculus, we provided a new semantics that closely corresponds to the former. The advantage of a $\pi$-calculus semantics is that we can use established proof techniques for reasoning about Øjeblik programs to prove our conjecture.

**Conjecture 8.1 (Safety).** *If we only consider external surrogations, then* $x \cong x.\mathsf{surrogate}$.

This conjecture is not easy to prove, because $\cong$ still quantifies over Øjeblik contexts, but only the contexts that instantiate $x$ actually need to be considered, as seen in § 5.

A variable $x$ can be bound in three ways: using the $\mathsf{let}$-construct, or using the $\varsigma$-binder, either as a self-variable or as an ordinary variable in a method. We may want to immediately omit the binding as a self-variable since we do not intend our conjecture does not address internal surrogation. However, $x$ may be bound as a self-variable in a nested way as in $C[\cdot] := [\,\mathrm{k}{=}\varsigma(x)[\,\mathrm{l}{=}\varsigma(s)[\cdot]\,]\,].\mathrm{k.l}$, where the surrogate operation would become external. This implies that we should not exclude the binding of $x$ as a self-variable from the set of contexts that we consider for external surrogations.

Consider $C[\cdot] := \mathsf{let}\, x = a \,\mathsf{in}\, [\cdot]$. When the hole $[\cdot]$ is ready to be evaluated, we know that $a$ has reduced to a value. During this evaluation, some concurrent computation could have been started. Note that this computation cannot interfere with the evaluation of $x$ or $x.\mathsf{surrogate}$ inserted into $[\cdot]$, because either (i) these concurrent computations enter the object bound to $x$, then they postpone the evaluation of the hole, or (ii) the hole gets access to the object bound to $x$, then it blocks access from the concurrent computations. This shows that in $\mathsf{let}$-contexts we may safely restrict our attention to just objects $\mathbb{O}$ or other variables $y$ as a binding for $a$, because any computation leading to these will not affect the use of $x$ later on for surrogation; in particular, possibly started concurrent computations of $a$ do not harm.

These consideration lead us to $x$-instantiating contexts.

**Definition 8.2 ($x$-instantiating contexts).** *Let $x$ be a variable, $\mathbb{O}$ an object with $x \notin \mathrm{fv}(\mathbb{O})$, and $C[\cdot]$ an Øjeblik context that does not bind $x$. Let $y$ and $z$ be fresh variables. Then $x$*-instantiating contexts *are defined inductively as:*

$$
\begin{array}{lcl}
C_1^x[\cdot] & ::= & \mathsf{let}\, x = \mathbb{O} \,\mathsf{in}\, C[\cdot] \\
 & \mid & [\,\dots, \mathrm{l}{=}\varsigma(\,..\,x\,..\,)C[\cdot]\,, \dots].\mathrm{l}\langle\,..\,\mathbb{O}\,..\,\rangle \\
C_{n+1}^x[\cdot] & ::= & \mathsf{let}\, y = \mathbb{O} \,\mathsf{in}\, D_n^{y\to x}[\cdot] \\
 & \mid & [\,\dots, \mathrm{l}{=}\varsigma(\,..\,y\,..\,)D_n^{y\to x}[\cdot]\,, \dots].\mathrm{l}\langle\,..\,\mathbb{O}\,..\,\rangle \\
D_1^{y\to x}[\cdot] & ::= & \mathsf{let}\, x = y \,\mathsf{in}\, C[\cdot] \\
 & \mid & [\,\dots, \mathrm{l}{=}\varsigma(\,..\,x\,..\,)C[\cdot]\,, \dots].\mathrm{l}\langle\,..\,y\,..\,\rangle \\
D_{n+1}^{y\to x}[\cdot] & ::= & \mathsf{let}\, z = y \,\mathsf{in}\, D_n^{z\to x}[\cdot] \\
 & \mid & [\,\dots, \mathrm{l}{=}\varsigma(\,..\,z\,..\,)D_n^{z\to x}[\cdot]\,, \dots].\mathrm{l}\langle\,..\,y\,..\,\rangle
\end{array}
$$

The advantage of $x$-instantiating contexts is that they have and inductively defined structure which helps us in proofs.

We get a preciser version of Conjecture 8.1.

**Theorem 8.3 (Safety).** *If we only consider external surrogations, then for all $x$-instantiating Øjeblik contexts $C_n^x[\cdot]$*

$$C_n^x[x] \cong C_n^x[x.\mathsf{surrogate}]$$

*Proof sketch.* (We only mention the crucial points.) At the level of the $\pi$-calculus semantics, the assumption of considering only external surrogations for an $x$-instantiating Øjeblik context amounts to a simple syntactic assumption. Let $\mathbb{O}$ be the object that is bound to $x$ at the moment when the surrogation becomes active, i.e., appears at top-level. Then, at this moment, the mutex of object $\mathbb{O}$ must be available, too, since this means that there is currently no method active in $\mathbb{O}$. We exploit this simple condition in the proof below.

**base case**

We have to prove that (i)

$$\mathsf{let}\, x = \mathbb{O} \,\mathsf{in}\, C[x] \cong \mathsf{let}\, x = \mathbb{O} \,\mathsf{in}\, C[x.\mathsf{surrogate}]$$

and (ii)

$$
\begin{array}{c}
[\,\dots, \mathrm{l}_j{=}\varsigma(s, ..\,x\,..\,)C[x]\,, \dots].\,\mathrm{l}_j\langle\,..\,\mathbb{O}\,..\,\rangle \\
\cong \\
[\,\dots, \mathrm{l}_j{=}\varsigma(s, ..\,x\,..\,)C[x.\mathsf{surrogate}]\,, \dots].\,\mathrm{l}_j\langle\,..\,\mathbb{O}\,..\,\rangle
\end{array}
$$

We consider only case (i), because the proof of (ii) is similar. If we put the two processes in a translated Øjeblik context, then they exhibit the same barbs. Then, we essentially carry out a standard bisimulation 'game' between left and right hand side of the equation according to the identical context $C[\cdot]$ on both side. We repeat this game, until we expose the context's hole $[\cdot]$ at top-level, which renders the access to the variable $x$, and respectively its surrogation, active.

Next, we carry out the surrogation under the assumption that it is external. During this process, we need to exploit some information about serialization, provided by translated contexts, in order to be sure that the surrogation operation is not interfered with by nasty contexts. Here, we exploit the critical fact that the mutex of the object bound to $x$ is currently available, which implies that there can be no other self-inflicted requests arriving from the surrounding derivative of a translated Øjeblik context.

After termination of the two-step surrogation, the overall result follows mainly by applying Laws 13 and 14.

**inductive step**

For using the inductive hypothesis we only need the two following easy results:

$$
\begin{array}{rccl}
1. & \mathsf{let}\, x = y \,\mathsf{in}\, a & \cong & a\{^{y}\!/_{x}\} \\
2. & [\,\dots, \mathrm{l}_j{=}\varsigma(\tilde{x})a\,, \dots].\,\mathrm{l}_j\langle\,\tilde{y}\,\rangle & \cong & a\{^{\tilde{y}}\!/_{\tilde{x}}\}
\end{array}
$$

□

The reason why, so far, we have only been able to prove that external surrogation is safe for a subset of Øjeblik terms is because $\cong$, although expressed in terms of $\pi$-calculus barbs, quantifies over Øjeblik contexts. The ideal situation would be, if we could show $x \cong_{bc} x.\mathsf{surrogate}$, thereby avoiding the quantification over Øjeblik contexts. Unfortunately, this requires that surrogation is safe (meaning transparent) even for $\pi$-calculus contexts, which is not true for the $\pi$-calculus semantics that we give in the present paper. For instance, a $\pi$-calculus term can easily bypass serialization and start concurrent activities in an object by using the internal self of an object concurrently.

We are currently working on an equivalent, but more robust $\pi$-calculus semantics and proving safety even with respect to $\mathrm{L}\pi$ contexts. This allows us to reason using $\mathrm{L}\pi$ laws and labeled characterizations of barbed congruence. Such a result is obviously much stronger than the above-mentioned one, because it tells us that surrogation is safe even with respect to $\mathrm{L}\pi$ attackers, not only translated Øjeblik contexts.

## 9 Conclusion

Our formal analysis of Øjeblik has proven quite fruitful. By a series of examples, we have shown that object surrogation in Øjeblik, and consequently object migration in Obliq, is not as transparent as one might think in the beginning. We have been able to verify, using the Obliq interpreter, that these examples are indeed examples that show problems with surrogation/migration in Obliq.

Most of the examples were discovered, when trying to prove the safety of surrogation using *pi*-calculus translations that implemented, first Obliq's semantics, then Talcott's semantics. These failed attempts were the ones that led us to the final semantics for Øjeblik. Also working with two different semantics, where the operational semantics is a rather high-level semantics and the $\pi$-calculus semantics is closer to an implementation, helped us clarify subtleties in the informal semantics of Obliq, because just the process of keeping two semantics in sync forces one to consider implementation details at different levels of abstraction.

The major "improvement" suggested by our formal semantics is the *forwarder* model for the treatment of aliasing, based on preentrant mutexes. It seems to be necessary in order to ensure transparency of even only external surrogation. This leads us to the definition of a repaired version of Obliq, aka: Repliq, by adopting the forwarder model.

To us, the major lesson learned from the work presented in this paper, is that *concurrent objects need formal analysis.* Not because one necessarily should prove properties, but because the formal analysis is a good debugging tool.

### Current and future work

Our current statement on the 'safety' of migration is not as good as we would like it to be. In particular, we are working on getting rid of $x$-instantiating contexts by the use of a more robust $\pi$-calculus semantics, by which we expect to be able to prove 8.1.

Of course, one more work is required on establishing an operational correspondence between the operational semantics and the $\pi$-calculus semantics. Such a result will definitely increase one's confidence in that the semantics really expresses what we have stated they do.

The $\pi$-calculus that we are using can be equipped with a type-system. One could also consider giving a type-system to Øjeblik. By translating types from Øjeblik to types in the $\pi$-calculus, we expect to get a proof of subject-reduction for Øjeblik types almost for free.

Now that we have shown that internal surrogation is not safe in general, it would be good to able to give a programmer a better syntactic criteria expressing that if she writes programs like this, then surrogation will be safe.

Finally, we would like to use the semantics to show other, perhaps simpler, properties about Øjeblik. For instance, one could consider showing that $\mathsf{join}(\mathsf{fork}(a)) \cong a$ under certain circumstances.

### Related work

Apart from Talcott [Tal96], closest to our work and like ours based on Abadi and Cardelli's object calculus [AC96b], is Gordon and Hankin's *concurrent object calculus* [GH98], where concurrency is introduced by means of (not completely commutative) parallel operators. However, no account on object migration has been addressed in their work.

## References


[AC96a] M. Abadi and L. Cardelli. An Imperative Object Calculus. *Theory and Practice of Object Systems*, 1(13):151–166, 1996.

[AC96b] 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 $\pi$-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 $\pi$-calculus (Note). Rapport de Recherche 1702, INRIA Sofia-Antipolis, May 1992.

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

[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.

[Car98] L. Cardelli. On the Semantics of Obliq. Personal Communication, 1998.

[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.

[HKNS98] H. Hüttel, J. Kleist, U. Nestmann and D. Sangiorgi. Surrogates in Øjeblik: Towards Migration in Obliq. In H. Hüttel and U. Nestmann, eds, *Proceedings of SOAP '98*, volume NS-98-5 of *BRICS Notes Series*, pages 43–50. BRICS, Denmark, 1998.

[HT92] K. Honda and M. Tokoro. On Asynchronous Communication Semantics. In M. Tokoro, O. Nierstrasz and P. Wegner, eds, *Object-Based Concurrent Computing 1991*, volume 612 of *LNCS*, pages 21–51. Springer, 1992.

[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.

[Mil93] R. Milner. The Polyadic $\pi$-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.

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

[Tal96] C. Talcott. Obliq semantics notes. Unpublished note, Jan. 1996.