BRICS

Basic Research in Computer Science

BRICS RS-99-44 Nestmann et al.: Aliasing Models for Mobile Objects

Aliasing Models for Mobile Objects

Uwe Nestmann
Hans Hüttel
Josva Kleist
Massimo Merro

BRICS Report Series **RS-99-44**

ISSN 0909-0878 **December 1999**

See back inner page for a list of recent BRICS Report Series publications. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/99/44/`

Aliasing Models for Mobile Objects*

Uwe Nestmann, Hans Hüttel, Josva Kleist
BRICS†, Aalborg University, Denmark

Massimo Merro‡
INRIA, Sophia-Antipolis, France

Abstract

In Obliq, a lexically scoped, distributed, object-oriented programming language, object migration was suggested as the creation of a copy of an object's state at the target site, followed by turning the object itself into an alias, also called *surrogate*, for the remote copy. We consider the creation of object surrogates as an abstraction of the above-mentioned style of migration. We introduce Øjeblik, a typed distribution-free subset of Obliq, and provide four different configuration-style semantics, which only differ in the respective *aliasing model*. We show that two of the semantics, one of which matches Obliq's implementation, render migration unsafe, while our new proposal allows for safe migration at least for a large class of program contexts. In addition, we propose a type system that allows a programmer to statically guarantee that programs belong to that class. Our work suggests a straightforward repair of Obliq's aliasing model.

*This paper is a revised and extended subset of *Migration = Cloning ; Aliasing* [HKMN99], which appeared in the *Informal Proceedings of FOOL 6*. An extended abstract of this revision, entitled *Aliasing Models for Object Migration*, appeared in the *Proceedings of EUROPAR '99*, LNCS 1685, September 1999.

†Basic Research in Computer Science, Centre of the Danish National Research Foundation.

‡Supported by a Marie Curie fellowhip, No. ERBFMBICT983504, in the EU TMR-programme.

Contents

1 Introduction

Mobile object systems have been investigated for more than one decade, starting out with early representants, like Emerald [JLHB88], and recently a whole series of scientific workshops has been dedicated to mobile object systems [Vit99]. While mobility in object systems has well-known advantages, for example as support for load balancing or mobile agents, there is no common agreement on how mobile objects should actually be realized.

In this paper, our main goal is *not* to argue for one and against another particular way to implement mobile objects, but rather to study in considerable formal detail one proposal that was already suggested for the above-mentioned Emerald: the proposal that object migration can be derived from two other primitives—cloning and aliasing—by just performing them one after the other.

Migration = Cloning ; Aliasing This particular style of programming mobile objects was also advocated by Cardelli within the context of the lexically-scoped distributed programming language Obliq [Car95]. Since we are aiming at a formal investigation into mobile objects, we prefer to study migration in Obliq rather than in Emerald since the former can easily be seen as a proper extension of Abadi and Cardelli's theoretically well-founded object calculi [AC96].

Obliq is well-suited for the above style of migration. 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 execution site. While immutable values can be copied from site to site, mutable values are stationary; when they are exchanged between programs on different sites only network references are transmitted. Accordingly, since objects are mutable, the migration of an object does not physically move the object, but instead creates a *clone* of the object at the target site and then turns the original (local) object into an *alias*—sometimes called *proxy*—for the new (remote) object.

In concurrent and distributed programs, it is important that certain state changes in parts of the running system may happen transparently from the point of view of the rest of the system. Ensuring that the implementation of such state changes is in fact transparent can be a difficult task since the programmer must in principle anticipate all possible execution scenarios. So the question arises naturally, whether object migration in Obliq is transparent to the object's clients, and how that can be stated formally.

Intuitively, migration of an object a to some other site works transparently, or safely, if (1) after the migration a client of a cannot tell that a is now an alias, and if (2) during migration it is not possible to interact with a in a way that prevents the migration operation from proper completion. In Obliq, mobile objects are therefore required to be *protected* and *serialized*: protection guarantees that aliases are persistent; serialization guarantees atomicity of the two-phase migration operation. Unfortunately, Obliq is not equipped with a formal semantics, except for an unpublished note by Talcott [Tal96], which provides a configuration-style semantics for a subset of Obliq excluding migration, so no formal treatment of the safety of migration has been possible up to now. The aim of our project [Nes99] is to remedy the lack of formality and to reason formally about migration.

From Migration to Surrogation We formalize the safety of migration based on an abstraction of it in a distribution-free setting. Indeed, since Obliq is lexically scoped, with respect to the results of Obliq computations we may safely ignore all aspects of distribution unless distribution sites fail. Following this idea, we define Øjeblik as Obliq's concurrent core. On the other hand, Øjeblik can also be seen as a particular concurrent extension of the Imperative Object Calculus [AC96]. In this core language, the *surrogation* of an object a is described as the creation of a clone b of a and then turning a itself into a proxy for b, which forwards future request for methods of a to b. This precisely models migration, except that neither a nor b are attached to any distribution site at all.

Aliasing Models for Mobile Objects For a formal study of surrogation, we have to come up with a formal semantics. However, there are many possible design choices. As in Obliq, the creation of stationary aliases due to surrogation usually results in general *alias chains* in a running system and, as it turns out, it is essential for the safety of migration to get just the modeling of nodes in alias chains right. A detailed discussion and formal presentation of four different aliasing models for Øjeblik represents the core of this paper. Interestingly, one of these models corresponds to the implementation of Obliq [Car94], while another one corresponds to Talcott's formal semantics for a subset of Obliq [Tal96], the only formal semantics for Obliq that we are aware of. Both of these models have severe problems with respect to the transparency of migration, as we shall exhibit formally by means of simple counterexamples.

Outline In Section 2, we present in detail our language Øjeblik for typed protected serialized concurrent objects. In Section 3, we informally develop four different aliasing models trying to emphasize the different design decisions behind them. In Section 4, we then formalize the models in terms of a configuration-style global-view semantics. To strengthen the meaningfulness of the semantics, we also collect a series of run-time properties of configurations and formulate them as invariants. In particular, we have a subject reduction theorem, so the typing of Øjeblik terms is compatible with the operational semantics. In Section 5, we investigate the safety of surrogation in the various aliasing models, where it turns out to be crucial whether an operation on an object occurs internally (requested by some method from within the object itself) or externally (requested from the outside of the object). This distinction enables us to draw several fundamental conclusions: (1) surrogation cannot be generally safe if its internal use is permitted, which is independent of the aliasing model; however, (2) only our so-called *forwarder* aliasing models have a chance to be safe for external surrogation; finally, (3) we provide a type system that statically guarantees that surrogations will always be external. In Section 6, we sum up our observations, we offer some interpretations, and we sketch some future work.

Related work Apart from Talcott [Tal96], closest to our work and like ours based on Abadi and Cardelli's object calculus [AC96] are two *concurrent object calculi*, one by Gordon and Hankin [GH98], and one by Di Blasio and Fisher [DF96]. However, no account on object migration has been addressed in these works. Sekiguchi and Yonezawa present an encoding of coreObliq into their calculus λdist, but they do not consider aliasing [SY97]. Emerald [JLHB88] has many things in common with Obliq, including a similar style of migration by means of cloning and aliasing, as proposed at an early stage. In Distributed Oz [VHB$^+$97] object migration is a primitive notion, so objects are physically mobile and travel according to a provably safe mobile state protocol from site to site, wherever they are needed or intend to go.

2 Typed Protected Serialized Concurrent Objects

In the predecessors of this paper, we presented Øjeblik as an untyped language [HKMN99, NHKM99]. However, types can be added in a straightforward manner, which we do here in order to provide a more complete semantic model of Øjeblik, e.g., by investigating its compatibility with reduction. Another interesting point is that subtyping cannot be straightforwardly introduced due to peculiarities of aliasing (cf. § 2.2).

For the sake of simplicity, in comparison with Obliq [Car95], we omit ground values (like numbers, booleans, strings, etc.), data operations, and procedures, we restrict field selection to method invocation, we restrict multiple cloning to single cloning, we omit flexibility of object attributes, we replace field aliasing with object aliasing, we omit explicit distribution, and we omit exceptions and advanced synchronization, so we get a feasible, but still non-trivial language.

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

Table 1: Syntax of Øjeblik expressions

2.1 Syntax and Informal Semantics

The set $\mathcal{L}$ of typed Øjeblik-expressions is generated as shown in Table 1, where *method labels* l and *variables* s, x, y, z are taken from countable sets $\mathbf{L}$ and $\mathbf{X}$, respectively. Our type system extends the typing rules for the Imperative Object Calculus of [AC96] with a family of *thread types* $\mathrm{Thr}(A)$. Pairs $\tilde{x}_j{:}\tilde{B}_j$ denote sequences $x_{1_j}{:}B_{1_j}, \ldots, x_{n_j}{:}B_{n_j}$.

Function types $A{\rightarrow}B$ do only occur in object types $[\mathrm{l}_j{:}\tilde{B}_j{\rightarrow}C_j]_{j\in J}$, so they are not first-class types. Nevertheless, we sometimes abbreviate such object types by $[\mathrm{l}_j{:}A_j]_{j\in J}$, when we intend to clarify that a type is not a thread type. Whenever appropriate or unambiguous, we deliberately omit the type information in bindings.

The remainder of this subsection presents an informal explanation of the semantics of Øjeblik terms, as suggested for Obliq [Car95], first ignoring aspects of protection and serialization, which are then explained as a second step. In the presentation, we assume that all terms are well-typed according to § 2.2, which formalizes that operations will always be "understandable". Here, we mean that operations are always only requested on objects with a matching interface, but we avoid the term "understood" here, since operations may be invalid due to protection, or delayed due to serialization (see below).

Computation follows the call-by-value (leftmost-innermost) evaluation order of Obliq. In particular, in the following, whenever we use a term a, we implicitly assume that we have first evaluated a to some actual value, i.e., in most cases to an object reference.

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

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

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

The operation $a.\mathsf{clone}$ creates an object with the same fields as the original object and initializes the fields to the same entries as in the original object.

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

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

The operation $a.\mathsf{ping}$ is also implemented by providing a uniform method: $\mathsf{ping}{=}\varsigma(s)s$. Thus, $a.\mathsf{ping}$ returns the "identity" of an object o resulting from the evaluation of a; note that, due to aliasing and forwarding, this could be the "identity" of the current endpoint of an alias chain potentially starting at object o. We add the $a.\mathsf{ping}$ method uniformly to Øjeblik objects, because it allows us to conveniently express an algebraic equation for the correctness of surrogation in Section 5. Furthermore, such a method could be used by clients for garbage collection of references to surrogated servers by interrogating the current identity and using it directly instead of the former indirect reference.

Scoping Apart from the binding of variables in method bodies, Øjeblik also offers explicit scope declarations. An expression $\mathsf{let}\, x = a \,\mathsf{in}\, b$ first evaluates a, binding the result to x, and then evaluates b within the scope of the new binding.

We use the standard inductive definition $\mathrm{fv}(a)$ to denote the free variables of term a with respect to our two forms of binding. Øjeblik only admits non-recursive expressions $\mathsf{let}\, x = a \,\mathsf{in}\, b$, i.e., with $x \notin \mathrm{fv}(a)$. Then, $a; b$ denotes $\mathsf{let}\, x = a \,\mathsf{in}\, b$, where $x \notin \mathrm{fv}(b)$. Finally, $\mathcal{L}_0$ denotes the set of *closed* Øjeblik terms $a \in \mathcal{L}$, i.e., where $\mathrm{fv}(a) = \emptyset$.

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

Self-Infliction, Serialization, Protection The *current method* of a thread is the last method invoked in it that has not yet completed. The *current self* of a thread is the self of its current method. An Øjeblik operation is *self-inflicted*, also called *internal*, if it addresses the current self; an operation is *external* if it is not self-inflicted.

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—for instance, it prevents recursion. Based on the notion of *thread*, so-called *reentrant* mutexes, as in Java, can be used to allow an operation to re-enter an object under the assumption that this operation belongs to the same thread as the operation that is currently active in the object. In Obliq, however, the more cautious idea of *self-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 the kind of inter-object mutual recursion, where a method in an object a calls a method in another object b, which then tries to 'call back' another method in a.

Based on self-infliction, objects are *protected* against external modifications in a natural way: updates, cloning, and aliasing, are only allowed if these operations are self-inflicted. In Øjeblik, for simplicity, all objects are both protected *and* serialized.

Executive Summary It is instructive to classify the operations on Øjeblik objects, as we do in the following table, not only according to the *protection conditions* (subject to self-infliction, or not), but also with respect to the *node of action* denoting the node where the operation is finally carried out (locally at the initially called node, or only at the endpoint of a chain starting at the called node). All of these properties are unambiguously stated in Obliq's informal semantics:

operation	*protection condition?*	*node of action*
cloning, aliasing	Y: self-inflicted	local
update	Y: self-inflicted	endpoint
invocation, surrogation, ping	N: unconstrained	endpoint

According to this table, we sometimes use the term "local request" to denote a request for cloning or aliasing and "endpoint request" for the others. Similarly, we sometimes use the term "unconstrained request" for those operations that are not subject to protection. Note the hybrid role of update requests, which are endpoint requests as well as subject to apparently local protection.

2.2 Types and Typing, but no Subtyping

In Table 2, we present the rules for static typing. Type environments Γ are finite lists of pairs of variables and types; they can also be seen as functions, where $\Gamma(x)$ searches in Γ from right to left for the first occurrence of x and returns the associated type. Type judgments are of the form $\Gamma \vdash a{:}A$ and express that term a has type A under the assumptions Γ on the free variables of a. The typing rules themselves are not surprising. The operations clone, alias, $\mathsf{surrogate}$, ping, and update, all yield a result of the same type as the object that they address. While fork packs a type into a thread type, join unpacks it accordingly. The rules for variables, let, objects, and invocations are standard.

The only potential surprise is that we cannot assume the usual subtyping rules of the Imperative Object Calculus [AC96]. These rules state that $\vdash [\mathrm{l}_j{:}B]_{j\in J} <: [\mathrm{l}_j{:}B]_{j\in I}$ if $I\subseteq J$ (T-SubObj) and that $\Gamma \vdash a{:}B$ if $\Gamma \vdash a{:}A$ and $\Gamma \vdash A{<:}B$ (T-Subsum). In

$$\mathsf{let}\; x{:}[\mathrm{l}_1{:}B_1, \mathrm{l}_2{:}B_2] = a \;\mathsf{in}\; \mathsf{let}\; y{:}[\mathrm{l}_1{:}B_1] = b \;\mathsf{in}\; x.\mathsf{alias}\langle y\rangle; x.\,\mathrm{l}_2$$

with arbitrary types B_1 and B_2 and terms a and b of the required types, the alias operation in the term is well-typed, as we can give a the type $[\mathrm{l}_1{:}B_1]$ by (T-Subsum). However, the program will fail at run-time when, after the aliasing, we activate the method l_2 on x, as l_2 is not available in y. So, we refrain from giving an account of subtyping (cf. § 6).

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

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

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

$$\frac{\Gamma \vdash a : [\mathrm{l}_j{:}\tilde{B}_j{\to}C_j]_{j\in J} \qquad \Gamma \vdash \tilde{b}_k{:}\tilde{B}_k \qquad k{\in}J}{\Gamma \vdash a.\mathrm{l}_k\langle\, \tilde{b}_k\,\rangle : C_k}\ (\textsc{T-Inv})$$

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

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

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

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

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

Table 2: Typing Rules

3 Intermezzo: Towards Formal Semantics

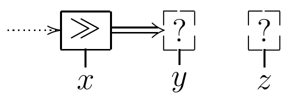
Although the informal semantics of Obliq is reasonably clear at first sight, its formalization enforces one to reason about even the slightest detail. In doing so, we discovered several fundamental facts on alias chains, implied by a few basic assumptions on Øjeblik's operations, that are worth being spelled out explicitly (§ 3.1). Based on them and the classification of operations in Section 2.1, a whole range of design choices for the forwarding of requests come into play preparing the ground to properly introduce the various aliasing models discussed in this paper (§ 3.2).

3.1 Facts: On the Stability of Alias Chains

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

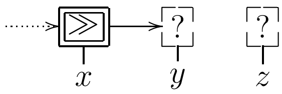
Below, we introduce pictures where we use single/double boxes to denote inactive/active nodes, and single/double arrows to denote unstable/stable aliases. Furthermore, dashed boxes and dotted arrows denote unspecified respective entities.

Inactive Nodes: No Tasks By definition, the only way to receive a self-inflicted request is to have already at least one local task running. In other words, if there is no local task, then each incoming request is doomed to be external. Now, consider the example term $\mathsf{let}\, z = [\,\mathrm{l}= \textit{"bar"}\,]\ \mathsf{in}\ \mathsf{let}\, y = [\,\mathrm{l}= \textit{"foo"}\,]\ \mathsf{in}\ \mathsf{let}\, x = [\,\mathrm{l}{=}\varsigma(s, w)s.\mathsf{alias}\langle w\rangle\,]\ \mathsf{in}\ x.\mathrm{l}\langle\, y\,\rangle; x.\mathrm{l}\langle\, z\,\rangle$ after it carried out the call $x.\mathrm{l}\langle\, y\,\rangle$, which means that the object referred to by x has turned itself into an alias for y and then terminated its activity. We depict the situation as follows

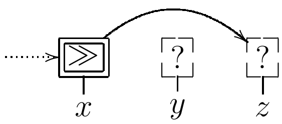


where, in general, the node x may itself be referred to by other aliases, while y and z may be either an alias or an object record. In fact, the alias $x{\gg}y$ is *stable* in the very sense: no re-aliasing operation on x to another node will ever possibly take place since it could only be carried out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

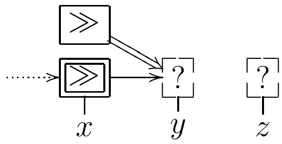
Active Nodes: At Least One Task As an example, let us first consider the term $\mathsf{let}\, z = [\,\mathrm{l}{=}\,\textit{“bar”}\,]\,\mathsf{in}\,\mathsf{let}\, y = [\,\mathrm{l}{=}\,\textit{“foo”}\,]\,\mathsf{in}\,\mathsf{let}\, x = [\,\mathrm{l}{=}\varsigma(s,w)s.\mathsf{alias}\langle w\rangle;\, \underline{\textit{“bla”}}\,]\,\mathsf{in}\, x.\mathrm{l}\langle\, y\,\rangle$ just after object x has accepted the request for method l and turned itself into an alias for y. Since x continues to operate on itself, according to *“bla”* in method l, x is an *active* alias node.



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



or



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

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

3.2 Design Choices: Four Aliasing Models

Apart from the above-mentioned facts about the stability of alias nodes, Obliq's informal semantics is rather imprecise on the behavior of alias nodes. In particular, there is quite some freedom on how to precisely model serialization and protection in aliased objects.

There are many possible variants, some of which we list below. First, we provide an abstract characterization. Then, we refine the description of the individual models by means of a comparison on how requests are forwarded, respectively.

How much Protection and Serialization? In this paper, we introduce and study

- a *conservative model* ($\mathcal{C}$) that keeps protection and serialization unchanged,
- a *relaxed model* ($\mathcal{R}$) that relaxes protection to some extent, but ignores serialization,
- a *forwarder model* ($\mathcal{F}$) that relaxes protection even more, and
- a *serialized model* ($\mathcal{S}$) that reintroduces some serialization to the forwarder model.

Our motivation for investigating and presenting all of the above four models is that $\mathcal{C}$ corresponds to Cardelli's implementation [Car94], while $\mathcal{R}$ corresponds to Talcott's operational semantics [Tal96]. As we will see in Section 5, neither of these two models provides sufficiently nice properties regarding the safety of surrogation. Therefore, we developed ourselves the model $\mathcal{F}$ as a consequent generalization of $\mathcal{R}$ that does well for surrogation, and $\mathcal{S}$ as a version of $\mathcal{F}$ with better programming and implementation properties.

Request Forwarding: Which? What? When? Each request arriving at an alias node must carry with it the knowledge about its "current self", i.e., an identification of the caller; this is required since otherwise self-infliction could not be checked locally by the requested node. The essential design choices to be made are the following:

1. Which requests (cf. table on page 7) are forwarded? (protection)
2. What is the current self of forwarded requests? (protection/serialization)
3. When does the forwarding take place? (serialization)

We use these questions to compare the individual above-mentioned aliasing models.

Which? The most intuitive model with respect to the first question, at least at first sight, is probably model $\mathcal{R}$: only endpoint requests are forwarded, local requests are handled locally. In contrast, model $\mathcal{C}$ is more restrictive for the case of updates. Recall that updates have a hybrid status in that they are protected endpoint requests: they can only perform meaningfully on object records, i.e., on endpoints of alias chains, and they can only be carried out when self-inflicted. Since in model $\mathcal{C}$, as in Cardelli's implementation, object aliasing is just as a macro for universal field aliasing, alias nodes are just the original objects with field redirection, so protection and serialization are still fully intact. Therefore, only unconstrained requests are forwarded, while the others—including updates—are either performed or blocked. If updates are self-inflicted in an intermediate node, then in model $\mathcal{R}$ they will be rejected only at the endpoint, while in model $\mathcal{C}$ already in the successor. The models $\mathcal{F}/\mathcal{S}$ are more liberal than model $\mathcal{R}$ and forward *all* requests except for self-inflicted local ones, because they are the only ones that can be carried out successfully in the requested node. This behavior will be of vital importance in Section 5.

What? This question has just two possible answers: either forwarding changes the current self of the request, or it does not. If it changes it, which is the case only in model $\mathcal{C}$, then the new current self is the one of requested node. In the models $\mathcal{R}/\mathcal{F}/\mathcal{S}$, however, forwarding does not change the current self. Although the difference is apparently minor, its impact is crucial for both protection and serialization: for example, consider a forwarded update request that keeps the same current self as it started out with—it may finally reach the endpoint of the alias chain and be self-inflicted *there*! The impact in the models $\mathcal{F}/\mathcal{S}$ is even bigger, because, as mentioned above, even cloning and aliasing requests may be forwarded.

When? The third question just concerns how much serialization is present in alias nodes. Here it is crucial to recall the fact that alias nodes are unstable as long as they are active (cf.§ 3.1). Should those external requests that are to be forwarded wait until the alias becomes stable, or should they be forwarded immediately, i.e., independent of stability? Here, the four models that we discuss behave again differently: model $\mathcal{C}$, as explained before, keeps serialization in aliases, so all external requests have to wait. Furthermore, the alias is afterwards locked until the forwarded request has successfully signaled termination from its point of action, so we may call this forwarding game *lock-and-go*. The models $\mathcal{R}$ and $\mathcal{F}$ completely ignore serialization in alias nodes, so forwarding of all intended requests is immediate. It is only now that we introduce our model $\mathcal{S}$ as a variant of model $\mathcal{F}$: here, all external requests to an alias node are blocked until the current tasks have finished. Then, the pending requests are forwarded without waiting for the previously forwarded request to signal termination; we call this forwarding game *touch-and-go* since external requests in alias nodes have to wait for the appearance of the node's mutex, but once it becomes available—in fact, just touchable—they will not grab it, but will simply go ahead. As we will argue later on (§ 4.4), in comparison with model $\mathcal{F}$, model $\mathcal{S}$ is better suited for programming since it enhances the predictability of program behaviors. It can also be used for *path-compression*, i.e., the optimization of alias chains to turns aliases into aliases for the farthest stable successor.

We summarize this rather complex discussion on design choices for aliasing models in Øjeblik by the following table. The important aspects are that the lower a model is listed in the table the more requests are forwarded, which prevents a client of an alias node to observe that the node really is an alias. This, however, only holds if the current self of forwarded requests is not changed by the alias node they pass.

	Which requests?	*What current self?*	*When?*
$\mathcal{C}$	all unconstrained	intermediate	lock-and-go
$\mathcal{R}$	all endpoint	unchanged	immediate
$\mathcal{F}$	all endpoint + external local	unchanged	immediate
$\mathcal{S}$	all endpoint + external local	unchanged	touch-and-go

There would, of course be more variants and combinations of different design decisions, but we chose the above due to their existing counterparts in Obliq ($\mathcal{C}/\mathcal{R}$) on the one hand, and due to their better surrogation properties ($\mathcal{F}/\mathcal{S}$) on the other hand.

Design Consequences The notion of current self is essential for the meaning of self-infliction. In fact, in the models $\mathcal{R}$, $\mathcal{F}$, and $\mathcal{S}$, which have in common that they do not change the current self of forwarded requests, self-infliction mutates naturally into *pre-infliction*: a thread may reenter an object or alias node from another node if it has only visited *pre*decessors of the current node since its last visit of the current node. In terms of an implementation using object mutexes, this amounts to a generalization of self-inflicted mutexes towards reentrant mutexes, but in a controlled way, so we may coin the term *preentrant mutexes* for this purpose. At Aalborg University, we have developed a prototype implementation, which allows us to experiment with the feasibility of such mutexes in practise [NOI+99].

4 Four Operational Semantics for Øjeblik

Our various transition semantics for Øjeblik terms follow loosely 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. The section is organized by first presenting the common basic concepts (§ 4.1) of the formal semantics for the various aliasing models (§ 4.2). We then offer behavioral semantics (§ 4.3), examples (§ 4.4), and an investigation of various properties of our semantics that prove the meaningfulness of our semantics (§ 4.5–4.6).

4.1 Common Basic Concepts

The semantics performs local changes on global run-time configurations, which are mappings from references $v \in \mathbf{R}$ to run-time entities. More precisely, a configuration $\mathfrak{C}$ maps task references $t \in \mathbf{R}_{\mathcal{T}}$ to *tasks* $\mathcal{T}$, and object references $o \in \mathbf{R}_{\mathcal{O}}$ to *objects* $\mathcal{O}$ (see below). We use $\mathrm{dom}_X(\mathfrak{C})$ to denote $\mathrm{dom}(\mathfrak{C}) \cap \mathbf{R}_X$ for $X \in \{\mathcal{T}, \mathcal{O}\}$, and $\uparrow$ for undefined references.

$$\boxed{a, b ::= \dots \mid v \mid \mathsf{wait}}$$

Table 3: Syntax of Øjeblik run-time expressions

Run-Time Entities A run-time expression a is generated from the extended Øjeblik grammar in Table 3, where we introduce references v as *values*, 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 $\mathcal{L}_{\mathbf{R}}$. A run-time object $O \in \mathcal{O}$ is either an object record $\mathbb{O}$ (ranging over $[\mathrm{l}_j{=}m_j]_{j \in J}$) or a pointer $\gg o$ to an object reference $o \in \mathbf{R}_{\mathcal{O}}$. A run-time task T is a triple $\langle\, p, s, a \,\rangle \in \mathbf{R}_{\mathcal{T}} \times \mathbf{R}_{\mathcal{O}} \times \mathcal{L}_{\mathbf{R}}$ that refers to a *parent* p, a current *self* s, and a run-time expression a that remains to be evaluated. By the partial functions $\mathrm{s}_{\mathfrak{C}}(t)$ and $\mathrm{p}_{\mathfrak{C}}(t)$, we refer to the—possibly undefined—current self and parent of the task associated with reference t in $\mathfrak{C}$. We reserve the task references $t_{\mathrm{m}}, t_{\mathrm{g}} \in \mathbf{R}_{\mathcal{T}}$ for special purposes.

Alias chains The partial function $\mathrm{ali}_{\mathfrak{C}} : \mathbf{R}_{\mathcal{O}} \rightharpoonup \mathbf{R}_{\mathcal{O}}^{*} \cup (\mathbf{R}_{\mathcal{O}}^{*} \cdot \{\uparrow\})$ with

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

computes the *alias chain*, starting at reference o, where $\cdot$ denotes concatenation of (sets of) strings of references, in general possibly ending with $\uparrow$. This computation obviously only terminates, if there are no cycles in the chain. The endpoint of an alias chain is denoted by $\mathrm{end}(\mathrm{ali}_{\mathfrak{C}}(o))$; if it exists, then the semantics will guarantee that it is associated with an object record $\mathbb{O}$ (cf. Lemma 4.5.1). We write $o' \in \mathrm{ali}_{\mathfrak{C}}(o)$ if o' occurs in the string representing the alias chain starting at o and we sometimes abbreviate finite alias chains $\{o := \gg o_1, \dots, o_i := \gg o_{i+1}, \dots, o_n := \mathbb{O}\}$ with $\{o \gg o_1 \cdots o_i \gg o_{i+1} \cdots o_n := \mathbb{O}\}$).

As a specialization of the above function, we define

$$\mathrm{pre}_{\mathfrak{C}}(o, s) \stackrel{\text{def}}{=} \begin{cases} \uparrow & \text{if } \mathfrak{C}(o) = \uparrow \\ o & \text{if } \mathfrak{C}(o) = \mathbb{O} \text{ or } o = s \\ o \cdot \mathrm{pre}_{\mathfrak{C}}(o', s) & \text{if } \mathfrak{C}(o) = \gg o' \text{ and } o \neq s \end{cases}$$

which yields the prefix of the alias chain starting in o that ends with the first occurrence of s, if it exists. If $s \notin \mathrm{ali}_{\mathfrak{C}}(o)$, then $\mathrm{pre}_{\mathfrak{C}}(o, s) = \mathrm{ali}_{\mathfrak{C}}(o)$.

We sometimes refer to object references as *nodes*, reflecting the fact that may be nodes in an alias chain. A node $o \in \mathrm{dom}_{\mathcal{O}}(\mathfrak{C})$ is *active* if there is $t \in \mathrm{dom}_{\mathcal{T}}(\mathfrak{C})$ with $\mathrm{s}_{\mathfrak{C}}(t) = o$, otherwise it is called *idle*. An alias node is a node $o \in \mathrm{dom}_{\mathcal{O}}(\mathfrak{C})$ with $\mathfrak{C}(o) = \gg o'$ for $o' \in \mathrm{dom}_{\mathcal{O}}(\mathfrak{C})$. An alias node is *stable* (c.f. § 3.1), if it is idle.

Threads Since tasks, in general, refer explicitly to their parent, we can build up task chains, which start in tasks that have no parent assigned. Such chains precisely correspond to Øjeblik *threads*, as derived from a task as the task's ancestors, similar to call-stacks in an implementation. Let $\mathrm{his}_{\mathfrak{C}} : \mathbf{R}_{\mathcal{T}} \to {\mathbf{R}_{\mathcal{T}}}^*$ with

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

be the *history* of a task t, where ϵ denotes the empty string.

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

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

be the *trace* of a task, representing the string of object references that occur as the current self of itself and its ancestors. For example, if $\mathfrak{C} = \{t_0 := \langle \uparrow, \uparrow, a_0 \rangle, t_1 := \langle t_0, s_1, a_1 \rangle, t_2 := \langle \uparrow, \uparrow, a_2 \rangle\}$, then $\mathrm{his}_{\mathfrak{C}}(t_1) = t_1 t_0$ and $\mathrm{trc}_{\mathfrak{C}}(t_1) = s_1$. A task with reference $t \neq t_{\mathrm{g}}$ is *current* in $\mathfrak{C}$, if (1) it is defined in $\mathfrak{C}$, (2) it is not the parent of any other task in $\mathfrak{C}$, and (3) it does not have parent t_{g}. The threads of $\mathfrak{C}$ are the histories of the current tasks in $\mathfrak{C}$:

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

In the above example, there are just the two threads $t_1 t_0$ and t_2.

Evaluation Table 4 also contains grammars for generating *redexes* r and *evaluation contexts* $e[\cdot]$, which we use to control the evaluation [FF86] of (the expression part of) run-time tasks. The contexts are designed such that evaluation proceeds leftmost-innermost. Note that $\mathsf{fork}\langle a \rangle$ is a redex independent of a being a value; similarly, an expression $\mathsf{let}\, x = a \,\mathsf{in}\, b$ becomes a redex as soon as the bound expression a has reduced to a value. According to this definition, a simple algorithm computes for every closed run-time expression $a \notin \mathbf{R}$ a *unique* pair of a redex r and a context $e[\cdot]$ such that $a = e[r]$.

Behaviors The semantics of a closed term $a \in \mathcal{L}_0$ is given by assigning to it the initial configuration $[\![\, a \,]\!] := \{t_{\mathrm{m}} {:=} \langle \uparrow, \uparrow, a \rangle, t_{\mathrm{g}} {:=} \langle \uparrow, \uparrow, t_{\mathrm{m}} \rangle\}$. The task referred to by t_{m} represents the start of the so-called *main* thread; the task reference t_{g} is used as the parent of all *garbage* task references, i.e., references to threads that are accomplished during reduction, and that have already been join'ed. Both, t_{m} and t_{g} are always defined in configurations.

The behavior of configurations is generated from (subsets of) the rules in Tables 5–12. 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, $\mathfrak{C}\{t{:=}T, o{:=}O\}$ 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 Table 5 describe the local activity in a single task t in a straightforward manner; recall that let is not recursive. Furthermore, we assume that the value v is either a task or an object reference whose actual run-time entity is accessible through $\mathfrak{C}$.

The rules in Table 6 exhibit the interplay of fork and join: in rule (Fork), a new task t' is spawned off, which runs the expression a without current self. In rule (Join), the parent referring to its child t' is returned a value v. Note that fork'ed tasks do not know their parent, so they indeed represent initial tasks of new threads. As soon as a thread t is join'ed, it is marked as garbage by means of the special reference t_{g} as its parent; no further attempt to join t will succeed, and t can not be reused after the first join.

The rules in Table 7 exemplify the synchronous method invocation protocol. In the rule schema ($\mathcal{X}$-Inv), a call to an object results in the creation of a new (callee-) task *within* the target object, while the caller-task is delayed, which is syntactically represented by the term wait inserted into its evaluation context. In rule (Ret), this caller-callee pair can communicate the result as soon as the callee-expression has reduced to a value; the task then disappears. A reference to a callee-task may even be reused after a successful (Ret), because the semantics guarantees that there is exactly one task, namely the caller-task, that is wait'ing for the callee-task to finish (cf. Lemma 4.5.2); in particular, references to callee-tasks never occur in run-time expressions—only references to fork'ed tasks can. Like the other rules of the operational semantics in Tables 8–12, especially the rules (Ali)/(Cln)/(Upd) for protected operations on objects, the rule ($\mathcal{X}$-Inv) for invocation crucially depends on how aliased objects should behave. Therefore, we parameterized the above rule schema, and start splitting up our presentation into different variants corresponding to different aliasing models $\mathcal{X}$, as introduced in the next subsection.

Self-Infliction In order to formalize the test for the either self-inflicted or pre-inflicted character of operations on objects, we introduce some suitable notation based on the information of alias chains in configurations. Let s be the current self of the caller-task, and o be the requested object reference. Then the following definitions are obvious.

$$
\begin{array}{rcl@{\quad}l}
r & ::= & \mathbb{O} \;\mid\; \mathsf{wait} \;\mid\; o.\mathrm{l}\Leftarrow m & \\
 & \mid & o.\mathrm{l}\langle\, \tilde{v} \,\rangle & \\
 & \mid & o.\mathsf{clone} & \\
 & \mid & o.\mathsf{alias}\langle o' \rangle & \\
 & \mid & o.\mathsf{surrogate} & \mid\; o.\mathsf{ping} \\
 & \mid & \mathsf{let}\; x = v \;\mathsf{in}\; b & \\
 & \mid & \mathsf{fork}\langle a \rangle & \mid\; \mathsf{join}\langle t \rangle
\end{array}
\qquad
\begin{array}{rcl@{\quad}l}
e[\cdot] & ::= & [\cdot] \;\mid\; e[\cdot].\mathrm{l}\Leftarrow m & \\
 & \mid & e[\cdot].\mathrm{l}\langle\, \tilde{a} \,\rangle & \mid\; o.\mathrm{l}\langle\, \tilde{v}, e[\cdot], \tilde{a} \,\rangle \\
 & \mid & e[\cdot].\mathsf{clone} & \\
 & \mid & e[\cdot].\mathsf{alias}\langle b \rangle & \mid\; o.\mathsf{alias}\langle e[\cdot] \rangle \\
 & \mid & e[\cdot].\mathsf{surrogate} & \mid\; e[\cdot].\mathsf{ping} \\
 & \mid & \mathsf{let}\; x = e[\cdot] \;\mathsf{in}\; b & \\
 & \mid & \mathsf{join}\langle e[\cdot] \rangle &
\end{array}
$$

Table 4: Evaluation of Øjeblik run-time expressions

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

$$
\frac{\mathfrak{C}(t) = \langle\, p, s, e[\mathbb{O}] \,\rangle \qquad \mathfrak{C}(o) = \uparrow}{\mathfrak{C} \;\rightarrow \mathfrak{C}\{t := \langle\, p, s, e[o] \,\rangle, o := \mathbb{O}\}} \quad (\textsc{New})
$$

Table 5: Local transitions

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

$$
\frac{\mathfrak{C}(t) = \langle\, p, s, e[\mathsf{join}\langle t' \rangle] \,\rangle \qquad \mathfrak{C}(t') = \langle\, \uparrow, \uparrow, v \,\rangle}{\mathfrak{C} \;\rightarrow \mathfrak{C}\{t := \langle\, p, s, e[v] \,\rangle, t' := \langle\, t_{\mathrm{g}}, \uparrow, v \,\rangle\}} \quad (\textsc{Join})
$$

Table 6: Concurrency transitions

$$
\frac{\begin{array}{l} \mathfrak{C}(t) = \langle\, p, s, e[o.\mathrm{l}_k\langle\, \tilde{v} \,\rangle] \,\rangle \qquad k \in J \\ \mathfrak{C}(o) = [\mathrm{l}_j{=}\varsigma(s_j, \tilde{x}_j)b_j]_{j\in J} \quad \ldots \quad \mathfrak{C}(t') = \uparrow \end{array}}{\mathfrak{C} \;\rightarrow \mathfrak{C}\{\, t := \langle\, p, s, e[\mathsf{wait}] \,\rangle, t' := \langle\, t, o, b_k\{{}^{o\tilde{v}}/_{s_k\tilde{x}_k}\} \,\rangle\}} \quad (\mathcal{X}\text{-}\textsc{Inv})
$$

$$
\frac{\mathfrak{C}(t) = \langle\, p, s, e[\mathsf{wait}] \,\rangle \qquad \mathfrak{C}(t') = \langle\, t, s', v \,\rangle}{\mathfrak{C} \;\rightarrow \mathfrak{C}\{t := \langle\, p, s, e[v] \,\rangle, t' := \uparrow\}} \quad (\textsc{Ret})
$$

Table 7: Synchronous Method Invocation

$$\begin{array}{lcl}
\text{self-infliction } (\mathcal{C}) & : & s = o \\
\text{pre-infliction for endpoint ops } (\mathcal{R}/\mathcal{F}/\mathcal{S}) & : & s = \mathrm{end}(\mathrm{ali}_{\mathfrak{C}}(o)) \\
\text{pre-infliction for local ops } (\mathcal{F}/\mathcal{S}) & : & s \in \mathrm{ali}_{\mathfrak{C}}(o)
\end{array}$$

In addition, we have to be able to check formally that an intended node o is currently available to deal with requests from task t:

$$\mathrm{Avail}_{\mathfrak{C}}(o,t) \stackrel{\mathrm{def}}{=} \underbrace{\bigwedge\nolimits_{t' \in \mathrm{dom}_{\mathcal{T}}(\mathfrak{C})} (o \neq \mathrm{s}_{\mathfrak{C}}(t'))}_{o \text{ is idle}} \quad \vee \quad \underbrace{(o = \mathrm{s}_{\mathfrak{C}}(t))}_{\text{self-inflicted}}$$

An object o is *available* for task t in $\mathfrak{C}$, if o is idle, or it it is the same as the current self of t, such that operations from t on o would be self-inflicted.

4.2 Aliasing Models

According to the four models informally proposed in Section 3.2, we now provide coherent formal semantics for each of them, where the inclusion of a rule to the semantics of a particular aliasing model is indicated by prefixing its name with $\mathcal{C}$, $\mathcal{R}$, $\mathcal{F}$, or $\mathcal{S}$.

4.2.1 The Conservative Model

Each alias node is protected. Thus, the rules in Table 8, all of which address protected operations on object o—which can only happen if they are self-inflicted—require the premise $o = s$. Note that for simplicity we do not generate run-time errors (as in Obliq) for invalid access to protected operations; the calls to such operations just block forever.

As exemplified in rule ($\mathcal{X}$-Inv), the rules ($\mathcal{C}$-Inv$_1$)/($\mathcal{C}$-Inv$_2$) in Table 9 formalize synchronous method invocation by means of new task t' in the target object o, while forcing the caller t to wait. The difference between these two rules is, whether the target o is an object record or an alias. In the former case, the available method body is instantiated immediately; in the latter case, instead, a new call is placed in the target object o, but now directed to the new target o', according to the aliasing information in $\mathfrak{C}$.

Each alias node is serialized. To this aim, the use of $\mathrm{Avail}_{\mathfrak{C}}(o,t)$ checks for availability of the current call from task t with respect to the target o. Note that a method call on an aliased object o in rule ($\mathcal{C}$-Inv$_2$) creates a forwarding sub-task t' with current self o. Essentially, this means that o's mutex is now locked by this task and that the current self of the forwarded call has been changed, according to the discussion in § 3.2.

4.2.2 The Relaxed Model

While protection is kept in alias nodes, serialization is ignored. Talcott [Tal96] proposes a scheme for method calls in Obliq, which is optimized in the sense that it directly addresses the endpoint of a chain, if it exists, and instantly creates a sub-task there. In rule ($\mathcal{R}/\mathcal{F}$-Inv) of Table 10, this scheme is formalized in terms of the functions end and $\mathrm{ali}_{\mathfrak{C}}$. In particular, no tasks are created in intermediate nodes between o and $\hat{o}$, which also means that none of the intermediate nodes is locked, and the availability of the endpoint is checked with respect to the current self s of the calling task t.

Although aliased objects are protected, updates on aliased objects are modeled as in rule ($\mathcal{R}/\mathcal{F}$-Upd): if an update on o is originating from the current endpoint of its alias chain (as checked by the condition $\mathrm{end}(\mathrm{ali}_{\mathfrak{C}}(o)) = s$), then it is forwarded there to take effect, there. Otherwise, the caller is blocked. It is this peculiar behavior that inspired us to formally introduce the notion of pre-infliction (cf. § 4.1).

While it is a convenient abstract optimization to immediately forward requests to the endpoint of a chain, as it is it does not give rise to a proper implementation technique, because unstable nodes are transparently traversed although they may give rise to different alias chain later on (c.f. § 3.1).

4.2.3 The Forwarder Model

We 'learn' from the peculiar modeling of update transitions in the relaxed model, and generalize it to also apply to cloning and aliasing: the rules in Table 11 replace the former tests $o = s$ for immediate self-infliction with a test for self-infliction on successors in the alias chain starting from the entry in the chain. In effect, this behavior prescribes an implementation of an aliased object as a pure *forwarder* for external and a partial forwarder for internal requests: partial since requests for invocation and update are forwarded, while requests for cloning and aliasing are accepted, if self-inflicted.

4.2.4 The Serialized Model

The step from the forwarder model to the serialized model is simple. To every rule for calling an operation at an object, we add the requirement that all alias nodes that have to be traversed in the alias chain starting from the entry node up to the point of action have to be available for the calling task. In the rules for *endpoint* operations ($\mathcal{S}$-Inv) and ($\mathcal{S}$-Upd), we add $\forall \dot{o} \in \mathrm{ali}_{\mathfrak{C}}(o) : \mathrm{Avail}_{\mathfrak{C}}(\dot{o}, t)$ to check the chain until the endpoint. In the rules for *local* operations ($\mathcal{S}$-Cln) and ($\mathcal{S}$-Ali), we instead add $\forall \dot{o} \in \mathrm{pre}_{\mathfrak{C}}(o, s) : \mathrm{Avail}_{\mathfrak{C}}(\dot{o}, t)$ to check the chain until we reach self-infliction.

4.2.5 Roundup

Whenever we refer to a particular aliasing model, we use the notation $\mathcal{X}[\![\, a \,]\!]$ to denote the semantics of a according to model $\mathcal{X} \in \{\mathcal{C}, \mathcal{R}, \mathcal{F}, \mathcal{S}\}$; otherwise, we simply write $[\![\, a \,]\!]$. In general, let $\rightarrow^*$ denote the reflexive-transitive closure of the transition relation on Øjeblik-configurations.

$$\frac{\mathfrak{C}(t) = \langle\, p, s, e[o.\mathsf{alias}\langle o'\rangle]\,\rangle \qquad o = s}{\mathfrak{C} \;\rightarrow \mathfrak{C}\{t := \langle\, p, s, e[o']\,\rangle, s := \gg o'\}} \quad (\mathcal{C}/\mathcal{R}\text{-}\textsc{Ali})$$

$$\frac{\mathfrak{C}(t) = \langle\, p, s, e[o.\mathsf{clone}]\,\rangle \qquad o = s \qquad \mathfrak{C}(o') = \uparrow}{\mathfrak{C} \;\rightarrow \mathfrak{C}\{t := \langle\, p, s, e[o']\,\rangle, o' := \mathfrak{C}(s)\}} \quad (\mathcal{C}/\mathcal{R}\text{-}\textsc{Cln})$$

$$\frac{\begin{array}{ll}\mathfrak{C}(t) = \langle\, p, s, e[o.\mathrm{l}_k \Leftarrow m]\,\rangle & o = s \\ \mathfrak{C}(s) = [\mathrm{l}_j{=}m_j]_{j\in J} & k\in J\end{array}}{\mathfrak{C} \;\rightarrow \mathfrak{C}\{\, t := \langle\, p, s, e[s]\,\rangle, s := [\mathrm{l}_k{=}m, \mathrm{l}_{j\neq k}{=}m_j]_{j\in J}\}} \quad (\mathcal{C}\text{-}\textsc{Upd})$$

Table 8: Protected transitions in the conservative (and relaxed) models

$$\frac{\begin{array}{lll}\mathfrak{C}(t) = \langle\, p, s, e[o.\mathrm{l}_k\langle\, \tilde{v}\,\rangle]\,\rangle & \mathrm{Avail}_{\mathfrak{C}}(o,t) & \mathfrak{C}(t') = \uparrow \\ \mathfrak{C}(o) = [\mathrm{l}_j{=}\varsigma(s_j, \tilde{x}_j)b_j]_{j\in J} & k\in J & \end{array}}{\mathfrak{C} \;\rightarrow \mathfrak{C}\{\, t := \langle\, p, s, e[\mathsf{wait}]\,\rangle, t' := \langle\, t, o, b_k\{^{o\tilde{v}}/_{s_k\tilde{x}_k}\}\,\rangle\}} \quad (\mathcal{C}\text{-}\textsc{Inv}_1)$$

$$\frac{\begin{array}{lll}\mathfrak{C}(t) = \langle\, p, s, e[o.\mathrm{l}\langle\, \tilde{v}\,\rangle]\,\rangle & \mathrm{Avail}_{\mathfrak{C}}(o,t) & \mathfrak{C}(t') = \uparrow \\ \mathfrak{C}(o) = \gg o' & & \end{array}}{\mathfrak{C} \;\rightarrow \mathfrak{C}\{\, t := \langle\, p, s, e[\mathsf{wait}]\,\rangle, t' := \langle\, t, o, o'.\mathrm{l}\langle\, \tilde{v}\,\rangle\,\rangle\}} \quad (\mathcal{C}\text{-}\textsc{Inv}_2)$$

Table 9: Method transitions in the conservative model

$$\frac{\begin{array}{lll}\mathfrak{C}(t) = \langle\, p, s, e[o.\mathrm{l}_k\langle\, \tilde{v}\,\rangle]\,\rangle & \mathrm{end}(\mathrm{ali}_{\mathfrak{C}}(o)) = \hat{o} & \mathrm{Avail}_{\mathfrak{C}}(\hat{o},t) \\ \mathfrak{C}(\hat{o}) = [\mathrm{l}_j{=}\varsigma(s_j, \tilde{x}_j)b_j]_{j\in J} & k\in J & \mathfrak{C}(t') = \uparrow\end{array}}{\mathfrak{C} \;\rightarrow \mathfrak{C}\{\, t := \langle\, p, s, e[\mathsf{wait}]\,\rangle, t' := \langle\, t, \hat{o}, b_k\{^{\hat{o}\tilde{v}}/_{s_k\tilde{x}_k}\}\,\rangle\}} \quad (\mathcal{R}/\mathcal{F}\text{-}\textsc{Inv})$$

$$\frac{\begin{array}{ll}\mathfrak{C}(t) = \langle\, p, s, e[o.\mathrm{l}_k \Leftarrow m]\,\rangle & \mathrm{end}(\mathrm{ali}_{\mathfrak{C}}(o)) = s \\ \mathfrak{C}(s) = [\mathrm{l}_j{=}m_j]_{j\in J} & k\in J\end{array}}{\mathfrak{C} \;\rightarrow \mathfrak{C}\{\, t := \langle\, p, s, e[s]\,\rangle, s := [\mathrm{l}_k{=}m, \mathrm{l}_{j\neq k}{=}m_j]_{j\in J}\}} \quad (\mathcal{R}/\mathcal{F}\text{-}\textsc{Upd})$$

Table 10: Method invocation and update in the relaxed and forwarder models

$$\frac{\mathfrak{C}(t) = \langle\, p, s, e[o.\mathsf{clone}]\,\rangle \qquad s \in \mathrm{ali}_{\mathfrak{C}}(o) \qquad \mathfrak{C}(o') = \uparrow}{\mathfrak{C} \;\rightarrow \mathfrak{C}\{t := \langle\, p, s, e[o']\,\rangle, o' := \mathfrak{C}(s)\}} \quad (\mathcal{F}\text{-}\textsc{Cln})$$

$$\frac{\mathfrak{C}(t) = \langle\, p, s, e[o.\mathsf{alias}\langle o'\rangle]\,\rangle \qquad s \in \mathrm{ali}_{\mathfrak{C}}(o)}{\mathfrak{C} \;\rightarrow \mathfrak{C}\{t := \langle\, p, s, e[o']\,\rangle, s := \gg o'\}} \quad (\mathcal{F}\text{-}\textsc{Ali})$$

Table 11: Protected transitions in the forwarder model

$$\frac{\begin{array}{l}\mathfrak{C}(t) = \langle\, p, s, e[o.\mathrm{l}_k\langle\, \tilde{v}\,\rangle]\,\rangle \qquad \mathrm{end}(\mathrm{ali}_{\mathfrak{C}}(o)) = \hat{o} \qquad \forall \dot{o} \in \mathrm{ali}_{\mathfrak{C}}(o) : \mathrm{Avail}_{\mathfrak{C}}(\dot{o}, t) \\ \mathfrak{C}(\hat{o}) = [\mathrm{l}_j{=}\varsigma(s_j, \tilde{x}_j)b_j]_{j\in J} \qquad k{\in}J \qquad \mathfrak{C}(t') = \uparrow\end{array}}{\mathfrak{C} \;\rightarrow \mathfrak{C}\{\, t := \langle\, p, s, e[\mathsf{wait}]\,\rangle, t' := \langle\, t, \hat{o}, b_k\{^{\hat{o}\tilde{v}}/_{s_k\tilde{x}_k}\}\,\rangle\}} \quad (\mathcal{S}\text{-}\textsc{Inv})$$

$$\frac{\begin{array}{l}\mathfrak{C}(t) = \langle\, p, s, e[o.\mathrm{l}_k{\Leftarrow}m]\,\rangle \qquad \mathrm{end}(\mathrm{ali}_{\mathfrak{C}}(o)) = s \qquad \forall \dot{o} \in \mathrm{ali}_{\mathfrak{C}}(o) : \mathrm{Avail}_{\mathfrak{C}}(\dot{o}, t) \\ \mathfrak{C}(s) = [\mathrm{l}_j{=}m_j]_{j\in J} \qquad k{\in}J\end{array}}{\mathfrak{C} \;\rightarrow \mathfrak{C}\{\, t := \langle\, p, s, e[s]\,\rangle, s := [\mathrm{l}_k{=}m, \mathrm{l}_{j\neq k}{=}m_j]_{j\in J}\}} \quad (\mathcal{S}\text{-}\textsc{Upd})$$

$$\frac{\begin{array}{l}\mathfrak{C}(t) = \langle\, p, s, e[o.\mathsf{clone}]\,\rangle \qquad s \in \mathrm{ali}_{\mathfrak{C}}(o) \qquad \forall \dot{o} \in \mathrm{pre}_{\mathfrak{C}}(o, s) : \mathrm{Avail}_{\mathfrak{C}}(\dot{o}, t) \\ \mathfrak{C}(o') = \uparrow\end{array}}{\mathfrak{C} \;\rightarrow \mathfrak{C}\{t := \langle\, p, s, e[o']\,\rangle, o' := \mathfrak{C}(s)\}} \quad (\mathcal{S}\text{-}\textsc{Cln})$$

$$\frac{\mathfrak{C}(t) = \langle\, p, s, e[o.\mathsf{alias}\langle o'\rangle]\,\rangle \qquad s \in \mathrm{ali}_{\mathfrak{C}}(o) \qquad \forall \dot{o} \in \mathrm{pre}_{\mathfrak{C}}(o, s) : \mathrm{Avail}_{\mathfrak{C}}(\dot{o}, t)}{\mathfrak{C} \;\rightarrow \mathfrak{C}\{t := \langle\, p, s, e[o']\,\rangle, s := \gg o'\}} \quad (\mathcal{S}\text{-}\textsc{Ali})$$

Table 12: Operations in the serialized model

Uniform Methods Both surrogate and ping are represented in Øjeblik as uniform methods that are treated like standard methods, except that they shall not be updatable. Thus, the treatment of requests for surrogate and ping in the various aliasing models is analogue the (Inv)-rules for standard methods, except that there is no requirement $k{\in}J$ to match one of the defined labels since surrogate and ping are uniformly and implicitly present.

Program Contexts In order to define a contextual notion of program equivalence in the next subsection, we need a general notion of program context that differs from the notion of evaluation context given in Table 4. More specifically, according to Table 13, a *context* $C[\cdot]$ has a single hole $[\cdot]$ that may be filled with an Øjeblik term.

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

Table 13: Øjeblik contexts

4.3 Behavioral Semantics

Based on a may-variant of convergence [Mor68] of terms, we define a contextual notion of equivalence [GHL97] uniformly for the four semantics of Øjeblik. In the context of a concurrent language with fork, threads may nondeterministically affect the outcome and convergence of evaluation. So, with respect to our goal of reasoning about surrogation, we regard *must*-variants of convergence as too strong (see also the testing semantics of the Join Calculus [Lan96]; stronger versions of equivalence based on *barbed bisimulation* can be found in [BFL98]).

Definition 4.3.1 (Convergence). *A closed term $a \in \mathcal{L}_0$* converges*, written $a{\Downarrow}$, if there is a configuration $\mathfrak{C}$ with $[\![\, a \,]\!] \ \rightarrow^* \mathfrak{C}$ and $\mathfrak{C}(t_{\mathrm{m}}) = \langle\, \uparrow, \uparrow, v\,\rangle$ for some v.*

This notion of convergence does not mean that the whole computation of term a terminates, but rather that the main task t_{m} does so: the evaluation of $[\![\, a \,]\!]$ *may* converge to a value v and *can* be reached in finite time within t_{m}. Note that there might be fork'ed computations around that have not yet been join'ed, and which may possibly run forever.

Definition 4.3.2 (Equivalence). *Two terms $a, b \in \mathcal{L}$ are equivalent, written $a \cong b$, if for all closing contexts $C[\cdot]$ (s.t. $C[a]$ and $C[b]$ are closed): $C[a]\Downarrow$ iff $C[b]\Downarrow$.*

In a typed language, it is natural to sometimes only consider well-typed terms. Although the notion of convergence itself is defined on all terms, it is natural to consider only well-typed contexts for equivalence.

Definition 4.3.3 (Typed Equivalence). *Two terms $a, b \in \mathcal{L}$ with $\Gamma \vdash a, b : A$ for some type A are* typed equivalent*, written $a =^{\vdash} b$, if for all well-typed closing contexts $C[\cdot]$, i.e., with $\emptyset \vdash C[a], C[b] : B$ for some type B, we have $C[a]\Downarrow$ iff $C[b]\Downarrow$.*

Note that the definition of typed equivalence does not use the type information to equate more terms—it just reduces the number of contexts that are taken into account for equivalence. We have the following close relation between the typed and untyped equivalence.

Lemma 4.3.4. *Let $a, b \in \mathcal{L}$. If $a \cong b$ and $\Gamma \vdash a, b : A$, then $a =^{\vdash} b$.*

A proof of this lemma is trivial since we check strictly fewer context for establishing $a =^{\vdash} b$.

Conjecture 4.3.5. *Let $a, b \in \mathcal{L}$. If $a =^{\vdash} b$, then $a \cong b$.*

We have not proved this conjecture, but we strongly believe that it holds since the only way an ill-typed context can invalidate $a \cong b$ is basically by using method labels inappropriately. Then, we conjecture that any such misbehaving context could be simulated by a well-typed one that simply diverges in the case that the ill-typed context produces a message-not-understood. We also expect the conjecture to hold when we take ground values into account. However, note that it would not hold in the context of subtyping: where the typing system would allow us to compare two objects $a := [\,\mathrm{l}_1{=}c_1,\ \mathrm{l}_2{=}c_2\,]$ and $b := [\,\mathrm{l}_1{=}c_1\,]$ at the common type $[\,\mathrm{l}_1 : C_1\,]$, thus rendering them equal, although the ill-typed context (c.f. Table 13) $C[\cdot] := [\cdot].\mathrm{l}_2$ would obviously be able to distinguish them.

We postpone the study of properties of our four semantics until after the next subsection, where we expose several sometimes surprising example behaviors that can be formulated in terms of Øjeblik.

4.4 Peculiarities of Infliction, Aliasing, and Fork

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

Example 4.4.1 (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:*

$$\mathsf{let}\; x = [\,\mathrm{l}{=}\varsigma(s,z)z.\mathsf{clone}\,]\;\mathsf{in}\; x.\mathrm{l}\langle x\rangle \tag{1}$$

The behavior of this term is unproblematic and independent of $\mathcal{X}$: the cloning operation may take place since z is replaced with x such that the call $z.\mathsf{clone}$ is indeed self-inflicted.

Example 4.4.2 (Self-Aliasing and Cycles). *Øjeblik does not prevent the programmer from (either consciously or accidentally) introducing self-aliases or alias chains with cycles, e.g. as in Equation (1) via substitution:*

$$\mathsf{let}\; x = [\,\mathrm{k}{=}\mathrm{id}\,,\; \mathrm{l}{=}\varsigma(s,z)s.\mathsf{alias}\langle z\rangle\,]\;\mathsf{in}\; x.\mathrm{l}\langle x\rangle ; x.\mathrm{k} \tag{2}$$

In the semantics, after carrying out the aliasing operation, by means of the call to $x.\mathrm{l}\langle x\rangle$, yielding configuration $\mathfrak{C}$, the call to $x.\mathrm{k}$ results in the function $\mathrm{ali}_{\mathfrak{C}}$ not terminating. Thus, every operation on an object in an alias chain with cycles via the object's successors will block (unless one of the objects breaks the cycle by means of internal re-aliasing; cf. § 3.1).

Example 4.4.3 (Re-aliasing vs Serialization). *Øjeblik allows the programmer to perform aliasing twice, one after the other, e.g. as in the following term:*

$$\begin{aligned}
&\mathsf{let}\; x = [\,\mathrm{k}{=}\mathrm{id}\,,\mathrm{l}{=}\mathrm{id}\,]\;\mathsf{in}\\
&\mathsf{let}\; y = [\,\mathrm{k}{=}\Omega\,,\mathrm{l}{=}\mathrm{id}\,]\;\mathsf{in}\\
&\mathsf{let}\; z = [\,\mathrm{k}{=}\Omega\,,\mathrm{l}{=}\varsigma(s)s.\mathsf{alias}\langle x\rangle; s.\mathsf{alias}\langle y\rangle\,]\;\mathsf{in}\\
&\mathsf{fork}\langle z.\mathrm{l}\rangle; z.\mathrm{k}
\end{aligned} \tag{3}$$

Interestingly, the outcome of the whole term depends on the chosen aliasing model. First, computation starts by setting up the three objects. Then, two competing method calls are launched on z. If the double-aliasing method on label l gets into z first, then we may reach the state, where this method has just performed the first aliasing request, ready to perform the second aliasing request, but having just opened a "convergence window" for $z.\mathrm{k}$.

Now, it matters whether or not the competing method call on label k is granted to pass through the locked object. If yes, as in the forwarder model $\mathcal{F}$ (and also model $\mathcal{R}$), then the main thread may reach a terminating state; otherwise, as in the serialized forwarder model $\mathcal{S}$ (and also model $\mathcal{C}$), the competing request may not pass through until the current method has terminated itself. However, then the second alias request must have been performed, and the pending request on label k will again be doomed to fail, i.e., it diverges.

Example 4.4.4 (Reflexive Threads). *Øjeblik allows the programmer, independent of the underlying aliasing model $\mathcal{X}$, to fork off a concurrent thread that gets to know its own thread identifier by means of an intermediate storage cell and uses it afterwards:*

$$\begin{aligned}
\mathsf{let}\ x = [\ & \mathrm{use}{=}\varsigma(s,t)\mathsf{join}\langle t\rangle\,, \\
& \mathrm{get}{=}\varsigma(s)\mathsf{fork}\langle[\,]\rangle\,, \\
& \mathrm{put}{=}\varsigma(s,t)s.\mathrm{get}{\Leftarrow}\varsigma(s)t;t\,]\ \mathsf{in} \\
x.\mathrm{put}\langle\ & \mathsf{fork}\langle x.\mathrm{use}\langle\, x.\mathrm{get}\,\rangle\rangle\,\rangle
\end{aligned} \tag{4}$$

We are not aware of a good application of this behavior, but at least we found it interesting to observe that it can be formulated within Øjeblik.

4.5 Invariant Run-Time Properties

In this subsection, we collect facts and lemmas about the well-definedness of our operational semantics. Although we concentrate our explanations and proofs here on the serialized model $\mathcal{S}$, which is our reference semantics, the results also hold for the other models.

Essentially, we collect properties of configurations that are invariant under reduction, under the assumption that we start out with an initial configuration that arises from a *static term*, as we may call terms in the original Øjeblik syntax, following the terminology of [GHL97]. Subject reduction, i.e., the invariance of typing under reduction, is then developed and proved in the next subsection. We start with a simple fact.

Fact 4.5.1 (Alias chains). *Let $a \in \mathcal{L}_0$ and $\mathcal{S}[\![\, a\,]\!] \;\rightarrow^*\; \mathfrak{C}$. For all $o \in \mathrm{dom}_{\mathcal{O}}(\mathfrak{C})$: either* $\mathrm{ali}_{\mathfrak{C}}(o)$ *does not terminate, or* $\mathfrak{C}(\mathrm{end}(\mathrm{ali}_{\mathfrak{C}}(o)))$ *is an object record* $[\mathrm{l}_j{=}m_j]_{j\in J}$.

A configuration $\mathfrak{C}$ is *well-behaved* if all of its objects are inhabited by at most one current task, if all run-time terms do not contain free variables, if all wait'ing tasks have exactly one deliverer for results, and if all non-current tasks are actually wait'ing. All configurations reachable in our semantics are well-behaved.

Lemma 4.5.2 (Well-behavior). *Let $a \in \mathcal{L}_0$ and $\mathcal{S}[\![\, a\,]\!] \;\rightarrow^*\; \mathfrak{C}$.*

1. *For all $o \in \mathrm{dom}_{\mathcal{O}}(\mathfrak{C})$:*
   there is at most one task $t \in \mathrm{dom}_{\mathcal{T}}(\mathfrak{C})$ with $o = \mathrm{s}_{\mathfrak{C}}(t)$ and t current in $\mathfrak{C}$.
2. *For all $t \in \mathbf{R}_{\mathcal{T}}, o \in \mathbf{R}_{\mathcal{O}}$:*
   if $\mathfrak{C}(t) = \langle\, p, s, b\,\rangle$, then b is a closed run-time term.
   if $\mathfrak{C}(o) = \mathbb{O}$, then $\mathbb{O}$ is a closed run-time term.
3. *For all $t \in \mathbf{R}_{\mathcal{T}}$:*

(a) *if* $\mathfrak{C}(t) = \langle\, p, s, e[\mathsf{wait}]\,\rangle$, *then there is exactly one* $t' \in \mathbf{R}_{\mathcal{T}}$ *with* $\mathfrak{C}(t') = \langle\, t, s', c\,\rangle$ *for some* s' *and* c.

(b) *if* $\mathfrak{C}(t) = \langle\, p, s, e[b]\,\rangle$ *and there is* $t' \in \mathbf{R}_{\mathcal{T}}$ *with* $\mathfrak{C}(t') = \langle\, t, s', c\,\rangle$ *for some* s' *and* c, *then* $b = \mathsf{wait}$.

Proof. By induction on the length of $\mathcal{S}[\![\, a \,]\!] \;\rightarrow^* \mathfrak{C}$. □

A configuration $\mathfrak{C}$ is *closed* if all object and task references occurring in run-time terms of task or in run-time object records are also captured within $\mathfrak{C}$, and if all task references occurring in run-time terms are mapped to thread-initial tasks, which also may have become garbage. All configurations reachable in our semantics are closed in that sense.

Lemma 4.5.3 (Closedness). *Let* $a \in \mathcal{L}_0$ *and* $\mathcal{S}[\![\, a \,]\!] \;\rightarrow^* \mathfrak{C}$.

1. *Let* $t \in \mathbf{R}_{\mathcal{T}}$ *and* $o, o' \in \mathbf{R}_{\mathcal{O}}$: *if* $\mathfrak{C}(t) = \langle\, p, s, C[o']\,\rangle$ *or* $\mathfrak{C}(o) = C[o']$ *for some* p, s, *and* $C[\cdot]$, *then* $o' \in \mathrm{dom}_{\mathcal{O}}(\mathfrak{C})$.
2. *Let* $t, t' \in \mathbf{R}_{\mathcal{T}}$ *and* $o \in \mathbf{R}_{\mathcal{O}}$: *if* $\mathfrak{C}(t) = \langle\, p, s, C[t']\,\rangle$ *or* $\mathfrak{C}(o) = C[t']$ *for some* p, s, *and* $C[\cdot]$, *then* $t' \in \mathrm{dom}_{\mathcal{T}}(\mathfrak{C})$. *Moreover* $\mathfrak{C}(t') = \langle\, p', s', b\,\rangle$ *for some* s', b, *and* $p' = \uparrow$ *or* $p' = t_{\mathrm{g}}$.

Proof. By induction on the length of $\mathcal{S}[\![\, a \,]\!] \;\rightarrow^* \mathfrak{C}$. □

Serialization in Øjeblik, and also in general, 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.

Definition 4.5.4 (Self-Serialization). *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 sequences* $t_n \dots t_0 \in \mathrm{thr}(\mathfrak{C})$*:* *if* $\mathrm{s}_{\mathfrak{C}}(t_i) = \mathrm{s}_{\mathfrak{C}}(t_j)$ *for* $0 \leq i < j \leq n$, *then* $\mathrm{s}_{\mathfrak{C}}(t_i) = \mathrm{s}_{\mathfrak{C}}(t_{i+1}) = \ldots = \mathrm{s}_{\mathfrak{C}}(t_{j-1})$.

Note that this definition is compatible with pre-infliction. In the models $\mathcal{R}$, $\mathcal{F}$, and $\mathcal{S}$, there is no book-keeping on traversed predecessor nodes, when requests are forwarded. Thus, when checking for the intersection of traces, we only see the "nodes of action".

Proposition 4.5.5. *Let* $a \in \mathcal{L}_0$. *If* $[\![\, a \,]\!] \;\rightarrow^* \mathfrak{C}$, *then* $\mathfrak{C}$ *is self-serialized.*

Proof (Sketch). Again, by induction on the length of $\mathcal{S}[\![\, a \,]\!] \;\rightarrow^* \mathfrak{C}$, exploiting in each case the premises of the transition rules that address self-infliction and availability of objects. 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), 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, except for the case of (Join), which is the only rule that changes the parent of a task. Yet, also this case is harmless since it does not interfere with the condition of serialization due to initial and garbage tasks not having a current self and, thus, no trace. ☐

4.6 Reduction vs Typing

We show that reduction preserves typing and, consequently, that every configuration reachable from a well-typed term is also well-typed. Therefore, we have to extend our type system to the domain of run-time configurations $\mathfrak{C}$, which involves all run-time extensions: references v, tasks T, proxies $\gg o$, and run-time terms $a \in \mathcal{L}_{\mathbf{R}}$. For convenience in the typing of run-time terms, especially to keep track of the types and occurrences of wait, we introduce the task-annotated construct wait_t, which is uniquely determined on its creation within the (Inv)-rules. Accordingly, we may strengthen Lemma 4.5.2-3.

Lemma 4.6.1. *Let $a \in \mathcal{L}_0$ and $[\![\, a \,]\!] \;\rightarrow^* \mathfrak{C}$. Let $t, p \in \mathbf{R}_{\mathcal{T}}$ and $s \in \mathbf{R}_{\mathcal{O}}$.*

1. *If $\mathfrak{C}(t) = \langle\, p, s, e[\mathsf{wait}_{t'}] \,\rangle$ for some t', then $\mathfrak{C}(t') = \langle\, t, s', c \,\rangle$ for some s' and c.*
2. *If $\mathfrak{C}(t) = \langle\, p, s, e[b] \,\rangle$ and $\mathfrak{C}(t') = \langle\, t, s', c \,\rangle$ for some b, t', s', and c, then $b = \mathsf{wait}_{t'}$. Furthermore, if $\mathfrak{C}(t'') = \langle\, p'', s'', e''[\mathsf{wait}_{t'}] \,\rangle$, then $t'' = t$, $s'' = s$, and $e''[\cdot] = e[\cdot]$.*

Note that the second property of this lemma now tells us that there are never two different parents waiting for the same task to finish. In Lemma 4.5.2-3, no such confusion could arise since parents did not know their children—with annotated wait, parents learn about their children, so we must explicitly prove that parents do not share their children.

Proof. By induction on the length of $[\![\, a \,]\!] \;\rightarrow^* \mathfrak{C}$. ☐

Since we annotate all binding occurrences of variables with types, together with the dynamic annotation of all occurrences of wait, and also the fact that we record types for all references ever created, the proof of subject reduction is smooth, even in the critical cases of method invocation and return, as well as the interplay between fork and join. Note that it is also because of subject reduction that we model garbage threads explicitly; as long as a thread may be referred to, we need some typing assumption to preserve typability.

$$\frac{\Gamma \vdash a{:}A \qquad p \notin \{\uparrow, t_{\mathrm{g}}\}}{\Gamma \vdash \langle\, p, s, a\,\rangle : A}\ (\textsc{T-Task}) \qquad \frac{\Gamma(v) = A}{\Gamma \vdash v{:}A}\ (\textsc{T-Ref})$$

$$\frac{\Gamma \vdash a{:}A \qquad p \in \{\uparrow, t_{\mathrm{g}}\}}{\Gamma \vdash \langle\, p, \uparrow, a\,\rangle : \mathrm{Thr}(A)}\ (\textsc{T-Thrd}) \qquad \frac{\Gamma(t) = A}{\Gamma \vdash \mathsf{wait}_t : A}\ (\textsc{T-Wait})$$

$$\frac{\Gamma \vdash o{:}A}{\Gamma \vdash \gg o : A}\ (\textsc{T-Proxy})$$

Table 14: Typing rules for run-time entities and run-time terms

As new forms of typing judgments we need $\Gamma \vdash v{:}A$, $\Gamma \vdash T{:}A$, $\Gamma \vdash \gg o{:}A$, and $\Gamma \vdash a{:}A$, where the type environments Γ now may also include pairs $v{:}A$ as assumptions on the types of references. The extended type system consists of the former rules in Table 2, now extended to admit run-time terms, and the additional rules in Table 14. The rules (T-Task) and (T-Thrd) derive the type of a task directly from the type of its enclosed run-time expression, where thread types only appear for orphan or garbage tasks. Similarly, the rule (T-Proxy) derives the type of proxies from the type of the target object. Recall from Lemma 4.5.2 that run-time expressions within tasks are always closed terms, so the type environment Γ in these cases will just have to contain typings for references, not for variables. Moreover, there are trivial rules to extract the types of references from the assumptions (T-Ref) and to extract the type of wait from its annotated task (T-Wait).

The typability of a configuration $\mathfrak{C}$ follows the standard idea of a sanity check that all of the run-time entities mentioned in the configuration, via references v, have to be typable under the assumptions that Γ makes of the types of those references (c.f. [Rep92]).

Definition 4.6.2. *A closed, well-behaved configuration $\mathfrak{C}$ is called* well-typed *with respect to an environment Γ, written $\Gamma \vdash \mathfrak{C}$, if for all $v \in \mathrm{dom}(\mathfrak{C})$, indeed $\Gamma \vdash \mathfrak{C}(v) : \Gamma(v)$.*

For subject reduction, we further need a series of standard lemmas, which we present without proofs, that allow us to manipulate type environments. In the following, q ranges over variables and references, and $\mathrm{refs}(a)$ extracts the references that occur within term a. The following typing results hold for all aliasing variants of our operational semantics.

The first two lemmas allow us to add and remove type assumptions about variables or references not used in the term being typed.

Lemma 4.6.3 (Weakening). *If $\Gamma \vdash a{:}A$ and $q \notin \mathrm{dom}(\Gamma)$, then $\Gamma, q{:}B \vdash a{:}A$.*

Lemma 4.6.4 (Contraction). *If* $\Gamma, q{:}B \vdash a{:}A$ *and* $q \notin \mathrm{fv}(a) \cup \mathrm{refs}(a)$*, then* $\Gamma \vdash a{:}A$.

The next standard lemma allows us to substitute a term for a variable if the variable is assumed to have the same type as the term.

Lemma 4.6.5 (Substitution). *If* $\Gamma, x{:}B \vdash a{:}A$ *and* $\Gamma \vdash b{:}B$*, then* $\Gamma \vdash a\{{}^{b}\!/_{x}\} : A$.

We often need to move terms in and out of evaluation contexts. Lemma 4.6.7 states when we may do so in a "type-safe" manner. The first part states the obvious fact that if we can type the composition of an evaluation context $e[\cdot]$ and a term a, then the bare term a is also typable. The second part states that we can exchange terms of the same type within typable evaluation contexts. First, however, we need a way to relate the type environments involved in the lemma.

Definition 4.6.6. *Two type environments* Γ *and* Γ' *are* compatible*, written* $\Gamma \asymp \Gamma'$*, if* $\Gamma(q) = \Gamma'(q)$ *for all* $q \in \mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma')$*. A type environment* Γ' *is a* compatible extension *of* Γ*, written* $\Gamma' \succeq \Gamma$*, if* $\mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(\Gamma')$ *and* $\Gamma \asymp \Gamma'$.

Lemma 4.6.7. *Let* $\Gamma \vdash e[a] : B$ *and* $\mathrm{fv}(e[a]) = \emptyset$*. Then:*

1. *There is a* $\Delta \succeq \Gamma$ *and* A *such that* $\Delta \vdash a{:}A$.
2. *If* $\Delta \vdash a, b : A$ *and* $(\mathrm{fv}(b) \cup \mathrm{refs}(b)) \subseteq (\mathrm{fv}(a) \cup \mathrm{refs}(a))$*, then* $\Gamma \vdash e[b] : B$.

We now proceed to the essential part of this subsection. The main result, subject reduction, states that typability of reachable configurations is preserved under reduction. Type safety, i.e., the property that every reachable configuration is in fact typable, is then proved by a simple induction using the main result.

Theorem 4.6.8 (Subject Reduction). *Let* $a \in \mathcal{L}_0$ *with* $\emptyset \vdash a{:}A$*. Let* $[\![\, a \,]\!] \rightarrow^{*} \mathfrak{C}$*. If* $\Gamma \vdash \mathfrak{C}$ *and* $\mathfrak{C} \rightarrow \mathfrak{C}'$*, then there is* Γ' *with* $\Gamma \asymp \Gamma'$ *such that* $\Gamma' \vdash \mathfrak{C}'$.

Proof. By transition induction. All cases in the proof follow the same scheme. Transitions $\mathfrak{C} \rightarrow \mathfrak{C}'$ involve one or two tasks. We need to construct a new type environment $\Gamma' \asymp \Gamma$ such that $\Gamma' \vdash \mathfrak{C}'$. Therefore, we extend Γ to Γ' by application of the rules (New), (Cln), (Inv), and (Fork), while (Ret) shrinks Γ to Γ' and in the other cases we have $\Gamma' = \Gamma$.

If $\Gamma \vdash \mathfrak{C}$, then for a task $T := \mathfrak{C}(t) = \langle\, p, s, a \,\rangle$, we must have $\Gamma \vdash T{:}C$ and $\Gamma \vdash a{:}B$; if $p \in \{\uparrow, t_{\mathrm{g}}\}$, then $C = \mathrm{Thr}(B)$, otherwise $C = B$ (c.f. Table 16). Now consider that task $T = \langle\, p, s, a \,\rangle$ takes part in a transition. Then, term a is decomposed into a context $e[\cdot]$ and a redex r, and a transition causes the redex to change into a run-time term b. What we need to show is that $e[b]$ has the same type as $a = e[r]$. This we do by first using Lemma 4.6.7-1 to derive a type for the redex r, then we argue that b has the same type as r, then we use Lemma 4.6.7-2 to derive that $e[b]$ has the same type as a. Finally, we can derive that the task $\langle\, p, s, e[b] \,\rangle$ has the same type as T by observing that p does not change (except for the rule (JOIN), but there it changes from $\uparrow$ to t_{g}, so the type is preserved also in that case). Having done so for the tasks involved in a transition we get that $\Gamma' \vdash \mathfrak{C}'$ (tasks not involved in a transition need not be considered, since we do not change assumptions about references when constructing Γ', we only add and remove assumptions).

(Let) By assumption $\Gamma \vdash e[\mathsf{let}\, x{:}A = v \,\mathsf{in}\, b] : B$ for some type B. By Lemma 4.6.7-1, there exists a $\Delta \succeq \Gamma$ such that for some type B', $\Delta \vdash \mathsf{let}\, x{:}A = v \,\mathsf{in}\, b : B'$. The only typing rule that applies here is (T-LET), implying that $\Delta \vdash v{:}A$ and $\Delta, x{:}A \vdash b{:}B'$. By Lemma 4.6.5, $\Delta \vdash b\{{}^{v}\!/_{x}\} : B'$, and by Lemma 4.6.7-2, $\Gamma \vdash e[b\{{}^{v}\!/_{x}\}] : B$.

($\mathcal{X}$-Inv) By assumption, $\Gamma \vdash e[o.\mathrm{l}_k\langle\, \tilde{v} \,\rangle]{:}B, \mathfrak{C}(o){:}\Gamma(o)$ and $\mathfrak{C}(o) = [\mathrm{l}_j{=}\varsigma(s_j{:}A, \tilde{x}_j{:}\tilde{B}_j)b_j]_{j\in J}$. The only rule allowing us to type $[\mathrm{l}_j{=}\varsigma(s_j{:}A, \tilde{x}_j{:}\tilde{B}_j)b_j]_{j\in J}$ is (T-OBJ) and therefore $\Gamma(o)$ must be a type $[\mathrm{l}_j{:}\tilde{B}_j{\to}C_j]_{j\in J}$. By Lemma 4.6.7-1, there is $\Delta \succeq \Gamma$ such that $\Delta \vdash o.\mathrm{l}_k\langle\, \tilde{v} \,\rangle : C$, which could only occur with an application of (T-INV). Therefore $C = C_k$ and $\Delta \vdash \tilde{v} : \tilde{B}_k$. By Lemma 4.6.4, we derive that $\Gamma \vdash \tilde{v} : \tilde{B}_k$. By Lemma 4.6.5, we derive that $\Gamma \vdash b_k\{{}^{o\tilde{v}}\!/_{s_k\tilde{x}_k}\} : C_k$, because $\Gamma, s_k{:}[\mathrm{l}_j{:}\tilde{B}_j{\to}C_j]_{j\in J}, \tilde{x}_k{:}\tilde{B}_k \vdash b_k{:}C_k$. Let $\langle\, p', o, b_k\{{}^{o\tilde{v}}\!/_{s_k\tilde{x}_k}\} \,\rangle$ be the new task asscociated with $t' \notin \mathrm{dom}(\Gamma)$, created by (INV). Let $\Gamma' = \Gamma, t'{:}C_k$. Then, we get $\Gamma' \vdash b_k\{{}^{o,\tilde{v}}\!/_{s_k,\tilde{x}_k}\} : \Gamma'(t')$. Finally, we get $\Gamma' \vdash e[\mathsf{wait}_{t'}] : B$ by rule (T-WAIT) and Lemma 4.6.7-2.

(Ret) We have two tasks $\mathfrak{C}(t) = \langle\, p, s, e[\mathsf{wait}_{t'}] \,\rangle$ and $\mathfrak{C}(t') = \langle\, t, s', v \,\rangle$, where $\Gamma \vdash e[\mathsf{wait}_{t'}] : B$ and $\Gamma \vdash v{:}C$. By Lemma 4.6.7-1, we have a $\Delta \succeq \Gamma$ such that $\Delta \vdash \mathsf{wait}_{t'} : C$, which can only be using (T-WAIT) with $\Delta(t') = \Delta(v) = A = C$. By Lemma 4.6.7-2, we immediately get $\Gamma \vdash e[v]{:}B$. Let $\Gamma' = \Gamma\backslash\{t'\}$. Finally, by Lemma 4.6.4, also $\Gamma' \vdash e[v]{:}B$, which reflects the fact that the task reference t' is no longer used.

(Fork) Assume $\Gamma \vdash e[\mathsf{fork}\langle a\rangle] : B$. By Lemma 4.6.7-1, we have $\Delta \vdash \mathsf{fork}\langle a\rangle : C$ for some $\Delta \succeq \Gamma$ and C. The only way to type $\mathsf{fork}\langle a\rangle$ is using (T-FORK) implying that $C = \mathrm{Thr}(A)$ and $\Delta \vdash a{:}A$. By Lemma 4.5.2, $\mathrm{fv}(a) = \emptyset$, and by Lemma 4.6.4, $\Gamma \vdash a{:}A$. Let $\Gamma' = \Gamma, t'{:}\,\mathrm{Thr}(A)$. Then, by Lemmas 4.6.3 and 4.6.7-2, $\Gamma' \vdash e[t']{:}B, a{:}A$.

(Join) Assume $\mathfrak{C}(t') = \langle\, \uparrow, \uparrow, v \,\rangle$ and $\Gamma \vdash e[\mathsf{join}\langle t'\rangle]{:}B, \langle\, \uparrow, \uparrow, v \,\rangle{:}C$. The only way to type $\langle\, \uparrow, \uparrow, v \,\rangle{:}C$ is using (T-THRD) implying that $C = \mathrm{Thr}(C')$ and $\Gamma \vdash v{:}C'$. Since $\Gamma \vdash \mathfrak{C}$, $\Gamma(t') = \mathrm{Thr}(C')$. By Lemma 4.6.7-1 for some $\Delta \succeq \Gamma$ and A, $\Delta \vdash \mathsf{join}\langle t'\rangle : A$. This typing must use (T-JOIN) as the last rule, this (together with (T-REF)) implies that $A = C'$. By Lemma 4.6.3, $\Delta \vdash v{:}C'$, and by Lemma 4.6.7-2, $\Gamma \vdash e[v] : A$. The join'ed task becomes a garbage task $\langle\, t_{\mathrm{g}}, \uparrow, v \,\rangle$, but this does not change its type since tasks with an undefined parent or the garbage parent both are typed with (T-THRD).

($\mathcal{X}$-Ali) As usual we have a $\Delta \succeq \Gamma$ and A such that $\Delta \vdash o.\mathsf{alias}\langle o'\rangle : A$. By (T-ALI) and (T-REF), this implies that $\Delta(o) = \Delta(o') = A$. By the observation that evaluation contexts do not bind references, $\Gamma(o) = \Gamma(o') = A$. By Lemma 4.6.7-2, $\Gamma \vdash e[o'] : A$. Consider an alias run-time entity $\mathfrak{C}(s^*) = \gg o^*$ with $\Gamma \vdash \mathfrak{C}$. We claim that s^* and o^* must have the same type. This is because, if $\Gamma \vdash \mathfrak{C}$, then for all references v in $\mathfrak{C}$, $\Gamma \vdash \mathfrak{C}(v) : \Gamma(v)$, then for a part of an alias chain $\mathfrak{C}(s^*) = \gg o^*$, $\Gamma \vdash o^* : \Gamma(s)$. Therefore, if $s \in \mathrm{ali}_{\mathfrak{C}}(o)$, then $\Gamma \vdash s{:}A$ and $\Gamma \vdash \gg o^* : \Gamma(s)$ as required.

($\mathcal{X}$-Cln) Assume $\Gamma \vdash e[o.\mathsf{clone}]$. By Lemma 4.6.7-1, we have a $\Delta \succeq \Gamma$ and A such that $\Delta \vdash o.\mathsf{clone} : A$. By Lemma 4.6.4, we get $\Gamma \vdash o.\mathsf{clone} : A$, and by (T-CLO) and (T-REF), we get $A = \Gamma(o)$. If $s \in \mathrm{ali}_{\mathfrak{C}}(o)$, like in case ($\mathcal{X}$-ALI), we have $\Gamma(s) = \Gamma(o)$. Now let $\Gamma' = \Gamma, o'{:}\Gamma(s)$. By Lemma 4.6.7-2, $\Gamma' \vdash e[o'] : B$ and $\Gamma' \vdash \mathfrak{C}'(o') : \Gamma'(o')$.

□

Theorem 4.6.9 (Type Safety). *Let $a \in \mathcal{L}_0$ with $\emptyset \vdash a{:}A$. If $[\![\, a \,]\!] \rightarrow^* \mathfrak{C}$, then there is Γ such that $\Gamma \vdash \mathfrak{C}$.*

Proof. By induction on the length of the reduction sequence.

(base case) Initial configurations of well-typed terms are well-typed: $\{t_{\mathrm{m}}{:}\,\mathrm{Thr}(A), t_{\mathrm{g}}{:}\,\mathrm{Thr}(\mathrm{Thr}(A))\} \vdash [\![\, a \,]\!]$.

(induction step) Let $[\![\, a \,]\!] \rightarrow^{n-1} \mathfrak{C}_{n-1} \rightarrow \mathfrak{C}_n = \mathfrak{C}$. Assume, by induction, that there is Γ_{n-1} such that $\Gamma_{n-1} \vdash \mathfrak{C}_{n-1}$. Then, Theorem 4.6.8 concludes the proof.

□

A final remark is due on the notion of type preservation. Indeed, we allow in our semantics that task references that are allocated for invocations are released after a successful termination in rule (RET). This is witnessed by the fact that, only in this case, the type environment may shrink. If, at some later moment in evaluation, the reference would be reused, this may in general be with a different type. Yet, this does not harm for subject reduction since one needs at least two steps to "change" the type of a task reference, while Theorem 4.6.8 is addresses single steps only. On the other hand, we may easily provide a semantics where task references are not reusable such that type environments never shrink.

5 On the Safety of Surrogation

The goal of this section is manifold. We motivate how to formulate the safety of surrogation in Øjeblik as an equation (§ 5.1). We study the two different kinds of external (§ 5.2) and self-inflicted (§ 5.3) surrogation and discuss how the various aliasing models behave in each of the cases. Then, we show how to formalize the fact that a certain surrogation is uniformly external (§ 5.4) and hint at a formal proof for the case of external surrogation (§ 5.5) that we report on in a companion paper. Finally, we provide a simple extension of Øjeblik's type system that guarantees uniformly external surrogation (§ 5.6).

5.1 Safety as an Equation

We formalize the idea that *an object before and after surrogation should behave the same in all possible contexts* as an equation using the notion of typed equivalence of Definition 4.3.3. Let $a \in \mathcal{L}$ be an Øjeblik term. Our first attempt was to require:

$$a =^{\vdash} a.\mathsf{surrogate} \tag{5}$$

It turns out that this conjecture is rather naive, and indeed wrong with all four aliasing models. In the remainder of this section we adapt and narrow down the above equation such that it becomes true, and provably so. The following discussion also generalizes to *migration* in a distributed lexically-scoped setting, like Obliq.

The simplest case of Equation 5 is where a is an Øjeblik object $\mathbb{O}$. In this case the surrogation is surely safe in all four semantics, because (1) the process of surrogation is carried out correctly since, due to serialization, 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 identically by $\mathbb{O}.\mathsf{surrogate}$, which suffices since after surrogation nobody has access to the previous $\mathbb{O}$.

In the general case, however, neither of the two above arguments holds. The reason is mainly because of possible copying of a reference to the former object such that, after surrogation, requests can still be directed to that reference. If these requests are subject to protection and serialization, then the semantics of aliasing models comes into the play. Observing that $a =^{\vdash} \mathsf{let}\, x = a \,\mathsf{in}\, x$ (in all contexts, the let just adds one unconditional step after reducing a) and that the notion of equivalence takes all well-typed Øjeblik contexts into account, Equation 5 can be reduced to the problem of surrogation on variables:

$$x =^{\vdash} x.\mathsf{surrogate} \tag{6}$$

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

$$C[\cdot] := \mathsf{let}\ x = [\ \mathrm{l}{=}\varsigma(s)s.\mathsf{alias}\langle s\rangle\ ]\ \mathsf{in}\ x.\mathrm{l}; [\cdot] \tag{7}$$

Independent of the underlying aliasing model, $C[x]\Downarrow$ since x returns immediately, while $C[x.\mathsf{surrogate}]\not\Downarrow$, because $x.\mathsf{surrogate}$ sends the request into a loop along the trivially cyclic alias chain. Due to the above problem, we refine the safety equation to include some trivial request to check whether the *object before surrogation* is actually reachable:

$$x.\mathsf{ping} =^{\vdash} x.\mathsf{surrogate} \tag{8}$$

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

A closer look at Øjeblik examples, as we will have in the following two subsections, shows that for the safety of surrogation it is crucial to distinguish, whether or not the call $x.\mathsf{surrogate}$ within a given context $C[\cdot]$ is "external for" x, or whether it is self-inflicted. Note that, unfortunately, this problem is undecidable, as already observed for Obliq [Car95]—only at run-time may we observe which case applies. Intuitively, this means that we must therefore execute a term $C[x.\mathsf{ping}]$ until that particular access to $x.\mathsf{ping}$ appears at the top-level for evaluation.

5.2 Aliasing Models for External Surrogation

We provide two single-threaded, distinguishing examples that exhibit unsafe surrogations in the conservative and relaxed models, while the forwarder models behave safely. The two examples have a very similar structure, only differing in the object $\mathbb{O}$ that, for the sake of well-typedness, has to serve single-parameter methods on label l.

$$C[\cdot] := \mathsf{let}\ x = \mathbb{O}\ \mathsf{in}\ [\cdot].\mathrm{l}\langle x\rangle$$

This context resembles the term of Equation (1) in that we intend to apply the context $C[\cdot]$ to two different requests: on the one hand $x.\mathsf{ping}$, and on the other hand its surrogating counterpart $x.\mathsf{surrogate}$. Applying our semantics we get that $C[x.\mathsf{ping}]$ and $C[x.\mathsf{surrogate}]$ roughly leads to $x.\mathrm{l}\langle x\rangle$ and $x'.\mathrm{l}\langle x\rangle$, respectively, where x' is bound to the surrogation result, which is definitely different from x itself.

In Table 15, we show the initial steps according to our operational semantics: we indicate the applied transition rules, as well as the respective uniquely defined redex for each following step. (We deliberately omit the garbage reference task associated with t_{g} since it is not used in the examples.) Here, we are not specific about the aliasing model on which the semantics we follow is based, because all models, so far, coincide. Note the two additional steps (Cln) and (Ali) needed to perform the surrogation in between the applications of (Inv) and (Ret). Note further the different resulting states X and S, where either o or o' is called, respectively, from within the main task t_{m} with o as parameter.

$$
\begin{aligned}
&[\![\, C[x.\mathsf{ping}] \,]\!]\\
&\quad =\{t_{\mathrm{m}} := \langle\,\uparrow,\uparrow, \mathsf{let}\, x = \underline{\mathbb{O}}\, \mathsf{in}\, x.\mathsf{ping}.\mathrm{l}\langle x\rangle\,\rangle\}\\
&\xrightarrow{(\textsc{New})}\{t_{\mathrm{m}} := \langle\,\uparrow,\uparrow, \underline{\mathsf{let}\, x = o\, \mathsf{in}\, x.\mathsf{ping}.\mathrm{l}\langle x\rangle}\,\rangle, o := \mathbb{O}\}\\
&\xrightarrow{(\textsc{Let})}\{t_{\mathrm{m}} := \langle\,\uparrow,\uparrow, \underline{o.\mathsf{ping}}.\mathrm{l}\langle o\rangle\,\rangle, o := \mathbb{O}\}\\
&\xrightarrow{(\textsc{Inv})}\{t_{\mathrm{m}} := \langle\,\uparrow,\uparrow, \underline{\mathsf{wait}}.\mathrm{l}\langle o\rangle\,\rangle, o := \mathbb{O}, t_1 := \langle\, t_{\mathrm{m}}, o, \underline{o}\,\rangle\}\\
&\xrightarrow{(\textsc{Ret})}\{t_{\mathrm{m}} := \langle\,\uparrow,\uparrow, \underline{o.\mathrm{l}\langle o\rangle}\,\rangle, o := \mathbb{O}\} \stackrel{\mathrm{def}}{=} X\\
&\\
&[\![\, C[x.\mathsf{surrogate}] \,]\!]\\
&\quad =\{t_{\mathrm{m}} := \langle\,\uparrow,\uparrow, \mathsf{let}\, x = \underline{\mathbb{O}}\, \mathsf{in}\, x.\mathsf{surrogate}.\mathrm{l}\langle x\rangle\,\rangle\}\\
&\xrightarrow{(\textsc{New})}\{t_{\mathrm{m}} := \langle\,\uparrow,\uparrow, \underline{\mathsf{let}\, x = o\, \mathsf{in}\, x.\mathsf{surrogate}.\mathrm{l}\langle x\rangle}\,\rangle, o := \mathbb{O}\}\\
&\xrightarrow{(\textsc{Let})}\{t_{\mathrm{m}} := \langle\,\uparrow,\uparrow, \underline{o.\mathsf{surrogate}}.\mathrm{l}\langle o\rangle\,\rangle, o := \mathbb{O}\}\\
&\xrightarrow{(\textsc{Inv})}\{t_{\mathrm{m}} := \langle\,\uparrow,\uparrow, \mathsf{wait}.\mathrm{l}\langle o\rangle\,\rangle, t_1 := \langle\, t_{\mathrm{m}}, o, o.\mathsf{alias}\langle \underline{o.\mathsf{clone}}\rangle\,\rangle, o := \mathbb{O}\}\\
&\xrightarrow{(\textsc{Cln})}\{t_{\mathrm{m}} := \langle\,\uparrow,\uparrow, \mathsf{wait}.\mathrm{l}\langle o\rangle\,\rangle, t_1 := \langle\, t_{\mathrm{m}}, o, \underline{o.\mathsf{alias}\langle o'\rangle}\,\rangle, o := \mathbb{O}, o' := \mathbb{O}\}\\
&\xrightarrow{(\textsc{Ali})}\{t_{\mathrm{m}} := \langle\,\uparrow,\uparrow, \underline{\mathsf{wait}}.\mathrm{l}\langle o\rangle\,\rangle, t_1 := \langle\, t_{\mathrm{m}}, o, \underline{o'}\,\rangle, o \gg o' := \mathbb{O}\}\\
&\xrightarrow{(\textsc{Ret})}\{t_{\mathrm{m}} := \langle\,\uparrow,\uparrow, \underline{o'.\mathrm{l}\langle o\rangle}\,\rangle, o \gg o' := \mathbb{O}\} \stackrel{\mathrm{def}}{=} S
\end{aligned}
$$

Table 15: Initial Executions of the Counterexample

Countering to the Conservative Model The following context distinguishes a request $x.\mathsf{ping}$ and its surrogating counterpart $x.\mathsf{surrogate}$ in the model $\mathcal{C}$:

$$
C_1[\cdot] := C[\cdot] \text{ with } \mathbb{O} \text{ instantiated as } \mathbb{O}_1 := [\,\mathrm{k}{=}\mathrm{id}\,, \mathrm{l}{=}\varsigma(s,z)z.\mathrm{k}\,] \tag{9}
$$

Let X_1 and S_1 denote the respective derivatives of $C_1[x.\mathsf{ping}]$ and $C_1[x.\mathsf{surrogate}]$ in Table 15. Then, we can trace some more reductions now specific to $\mathbb{O}_1$:

$$
\begin{aligned}
X_1 &\xrightarrow{(\textsc{Inv})} \{t_{\mathrm{m}} := \langle\,\uparrow,\uparrow,\mathsf{wait}\,\rangle, t' := \langle\, t_{\mathrm{m}}, o, \underline{o.\mathrm{k}}\,\rangle, o := \mathbb{O}_1\}\\
&\xrightarrow{(\textsc{Inv})} \{t_{\mathrm{m}} := \langle\,\uparrow,\uparrow,\mathsf{wait}\,\rangle, t' := \langle\, t_{\mathrm{m}}, o, \underline{\mathsf{wait}}\,\rangle, t'' := \langle\, t', o, \underline{o}\,\rangle, o := \mathbb{O}_1\}\\
&\xrightarrow{(\textsc{Ret})} \{t_{\mathrm{m}} := \langle\,\uparrow,\uparrow,\underline{\mathsf{wait}}\,\rangle, t' := \langle\, t_{\mathrm{m}}, o, \underline{o}\,\rangle, o := \mathbb{O}_1\}\\
&\xrightarrow{(\textsc{Ret})} \{t_{\mathrm{m}} := \langle\,\uparrow,\uparrow, o\,\rangle, o := \mathbb{O}_1\} \stackrel{\mathrm{def}}{=} X_1' \not\rightarrow
\end{aligned}
$$

Here we were not specific about the semantics we follow when applying rule (Inv) since the behavior is the same in either case ($\mathcal{C}$-$\textsc{Inv}_1$) or ($\mathcal{R}/\mathcal{F}/\mathcal{S}$-$\textsc{Inv}$) and leads to convergence ($X_1{\Downarrow}$ and thus $C_1[x]{\Downarrow}$).

In contrast, for the execution of S_1, the various semantics exhibit different behaviors (which we indicate by subscripts) after the first common invocation step:

$$S_1 \xrightarrow{(\textsc{Inv})} \{t_\mathrm{m} := \langle\, \uparrow, \uparrow, \mathsf{wait}\,\rangle, t' := \langle\, t_\mathrm{m}, o', \underline{o.\mathrm{k}}\,\rangle, o \gg o' := \mathbb{O}_1\} \stackrel{\mathrm{def}}{=} S_1'$$

While the model $\mathcal{C}$ blocks after

$$S_1' \xrightarrow{(\textsc{Inv}_2)}_{\mathcal{C}} \{t_\mathrm{m} := \ldots, t' := \langle\, t_\mathrm{m}, o', \mathsf{wait}\,\rangle, t_\mathrm{i} := \langle\, t', o, \underline{o'.\mathrm{k}}\,\rangle, o \gg o' := \mathbb{O}_1\}$$

because the addressed object o' is already inhabited by task t', the models $\mathcal{R}/\mathcal{F}/\mathcal{S}$ exhibit further reductions, like X_1, although yielding value o':

$$\begin{aligned}
S_1' &\xrightarrow{(\textsc{Inv})}_{\mathcal{R}/\mathcal{F}/\mathcal{S}} \{t_\mathrm{m} := \ldots, t' := \langle\, t_\mathrm{m}, o', \underline{\mathsf{wait}}\,\rangle, t'' := \langle\, t', o', \underline{o'}\,\rangle, o \gg o' := \mathbb{O}_1\} \\
&\xrightarrow{(\textsc{Ret})}_{\mathcal{R}/\mathcal{F}/\mathcal{S}} \{t_\mathrm{m} := \langle\, \uparrow, \uparrow, \underline{\mathsf{wait}}\,\rangle, t' := \langle\, t_\mathrm{m}, o', \underline{o'}\,\rangle, o \gg o' := \mathbb{O}_1\} \\
&\xrightarrow{(\textsc{Ret})}_{\mathcal{R}/\mathcal{F}/\mathcal{S}} \{t_\mathrm{m} := \langle\, \uparrow, \uparrow, o'\,\rangle, o \gg o' := \mathbb{O}_1\} \stackrel{\mathrm{def}}{=} S_1'' \not\rightarrow
\end{aligned}$$

The reason is simply that in the latter modeling, the intermediate task t_i with parent o is not created at all, but instead the forwarding to o', which represents the endpoint of the alias chain when starting at o, is done immediately, while still coming from parent o', such that it is accepted as self-inflicted.

To summarize, we observe that in all four aliasing models, $C_1[x.\mathsf{ping}]$ converges since the critical call $z.\mathrm{k}$ of Equation (9) is transformed, at run-time, into $o.\mathrm{k}$ with parent o, i.e. into a self-inflicted call. In contrast, for $C_1[x.\mathsf{surrogate}]$, this call arises at run-time with parent o', by then the surrogation target of object o, and comes back to o' as a forwarded request. In the models $\mathcal{R}/\mathcal{F}/\mathcal{S}$, the whole term converges in accordance with $C_1[x.\mathsf{ping}]$, but the model $\mathcal{C}$ blocks this call, as it is not self-inflicted due to the intermediate parent o created for task t_i.

Countering to the Relaxed Model The following context distinguishes between a request $x.\mathsf{ping}$ and its surrogating counterpart $x.\mathsf{surrogate}$ in the model $\mathcal{R}$:

$$C_2[\cdot] := C[\cdot] \text{ with } \mathbb{O} \text{ instantiated as } \mathbb{O}_2 := [\,\mathrm{l}{=}\varsigma(s,z)z.\mathsf{clone}\,] \tag{10}$$

Let X_2 and S_2 denote the respective derivatives of $C_2[x.\mathsf{ping}]$ and $C_2[x.\mathsf{surrogate}]$ in Table 15. As before, we now trace some more reductions of X_2 and S_2:

$$\begin{aligned}
X_2 &\xrightarrow{(\textsc{Inv})} \{t_\mathrm{m} := \langle\, \uparrow, \uparrow, \mathsf{wait}\,\rangle, t' := \langle\, t_\mathrm{m}, o, \underline{o.\mathsf{clone}}\,\rangle, o := \mathbb{O}_2\} \\
&\xrightarrow{(\textsc{Cln})} \{t_\mathrm{m} := \langle\, \uparrow, \uparrow, \underline{\mathsf{wait}}\,\rangle, t' := \langle\, t_\mathrm{m}, o, \underline{o'}\,\rangle, o := \mathbb{O}_2, o' := \mathbb{O}_2\} \\
&\xrightarrow{(\textsc{Ret})} \{t_\mathrm{m} := \langle\, \uparrow, \uparrow, o'\,\rangle, o := \mathbb{O}_2, o' := \mathbb{O}_2\} \stackrel{\mathrm{def}}{=} X_2' \not\rightarrow
\end{aligned}$$

Immediately, we observe the well-behavior of X_2 in all aliasing models since the cloning requested within task t' is obviously self-inflicted and leads to convergence ($X_2\Downarrow$, and thus $C_2[x.\mathsf{ping}]\Downarrow$).

In contrast, the behavior of the corresponding S_2 after the first step

$$S_2 \xrightarrow{(\text{Inv})} \{t_{\text{m}} := \langle\,\uparrow,\uparrow,\mathsf{wait}\,\rangle, t' := \langle\, t_{\text{m}}, o', \underline{o.\mathsf{clone}}\,\rangle, o{\gg}o' := \mathbb{O}_2\} \stackrel{\text{def}}{=} S_2'$$

is quite different for the respective models, due to the external cloning request. In the model $\mathcal{C}$, we immediately get $S_2' \not\to_{\mathcal{C}}$. However, also in the model $\mathcal{R}$, we get $S_2' \not\to_{\mathcal{R}}$; the reason is that forwarding, which helped us in dealing with Example (9), was only proposed for invocations and updates.

This is exactly the motivation for our forwarder models $\mathcal{F}$ and $\mathcal{S}$, which remedy the above situation, because they recognize and accept *pre-infliction* also for cloning and aliasing. Consequently, they lead to convergence ($S_2'\Downarrow$ and thus $C_2[x.\mathsf{surrogate}]\Downarrow$):

$$\begin{aligned} S_2' &\xrightarrow{(\text{Cln})}_{\mathcal{F}/\mathcal{S}} \{t_{\text{m}} := \langle\,\uparrow,\uparrow,\underline{\mathsf{wait}}\,\rangle, t' := \langle\, t_{\text{m}}, o', \underline{o''}\,\rangle, o{\gg}o' := \mathbb{O}_2, o'' := \mathbb{O}_2\} \\ &\xrightarrow{(\text{Ret})}_{\mathcal{F}/\mathcal{S}} \{t_{\text{m}} := \langle\,\uparrow,\uparrow,o''\,\rangle, o{\gg}o' := \mathbb{O}_2, o'' := \mathbb{O}_2\} \stackrel{\text{def}}{=} S_2'' \not\to \end{aligned}$$

To summarize, again in all four aliasing models, $C_2[x.\mathsf{ping}]$ converges since the cloning on z of Equation (10) is self-inflicted at run-time. However, among our four candidate semantics only the two forwarder models handle these requests properly for $C_2[x.\mathsf{surrogate}]$.

5.3 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 cannot be safe in general, not even in the forwarder models. We classify two different sets of examples depending on whether they exhibit problems with access to a self-surrogated source or the target thereof. As for the detailed explanation of the examples in terms of their explicit reductions and reachable configurations, we omit them here and leave them as an exercise; they are similar to the developments in the previous subsection.

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

$$C_3[\cdot] := [\,\mathrm{k}{=}\varsigma(s)\mathsf{let}\, y = [\cdot]\, \mathsf{in}\, y.\mathsf{clone}\,].\mathrm{k} \tag{11}$$

yields $C_3[s.\mathsf{ping}]\Downarrow$ and $C_3[s.\mathsf{surrogate}]\not\Downarrow$, because in $C_3[s.\mathsf{ping}]$ the cloning of y is allowed, while in $C_3[s.\mathsf{surrogate}]$ the corresponding call is blocked due to protection.

Source Problems Another problem arises by externally sending a request to the surrogation target, but now via the surrogation source, e.g., for updates:

$$C_4[\cdot] := [\,\mathrm{k}{=}\varsigma(s)\mathsf{let}\, y = [\cdot]\,\mathsf{in}\, s.\mathrm{k}{\Leftarrow}\,\mathrm{id}\,].\mathrm{k} \tag{12}$$

yields $C_4[s.\mathsf{ping}]\Downarrow$ and $C_4[s.\mathsf{surrogate}]\not\Downarrow$. By sending an update to itself after having surrogated, as in $C_4[s.\mathsf{surrogate}]$, the update of y is blocked, 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:

$$\begin{aligned} C_5[\cdot] := \mathsf{let}\ x = {} & [\,\mathrm{l}{=}\Omega\,,\mathrm{k}{=}\Omega\,]\ \mathsf{in} \\ & [\,\mathrm{l}{=}\mathrm{id}\,,\mathrm{k}{=}\varsigma(s)\mathsf{let}\, y = [\cdot]\,\mathsf{in}\, s.\mathsf{alias}\langle x\rangle; y.\mathrm{l}\,].\mathrm{k} \end{aligned} \tag{13}$$

yields $C_5[s.\mathsf{ping}]\not\Downarrow$ and $C_5[s.\mathsf{surrogate}]\Downarrow$. Whereas $C_5[s.\mathsf{ping}]$ diverges since the alias call to s also affects y in that case, the counterpart $C_5[s.\mathsf{surrogate}]$ converges since the re-aliasing of s does not affect the target y.

All of these examples of observable self-surrogation can be interpreted as programming errors. In each case, the current method has the complete and local knowledge about the object and its surrogated counterpart at hand—because it itself performed the surrogation—but nevertheless operates on either of the two in an obviously "stupid" and avoidable manner. In these cases, it is not the fault of the semantics, but the fault of the programmer.

5.4 On the Absence of Self-Surrogation

The previous subsection exhibited that self-inflicted surrogation is inherently problematic. Thus, we should try to achieve a safety theorem at least for those cases where surrogation is guaranteed to be always external. Although this is an undecidable criterion, it can be formally defined and then used in formal proofs.

It suffices to concentrate the exposition on endpoint operations $\mathsf{op} \in \{\mathsf{ping}, \mathsf{surrogate}\}$. Recall from Subsection 4.1 the characterization of self-infliction for the various aliasing models. We use the respective case for endpoint operations op in the context of the serialized forwarder model, which says that a particular request $o.\mathsf{op}$ with current self s is pre-inflicted in a particular run-time configuration $\mathfrak{C}$ if $s = \mathrm{end}(\mathrm{ali}_{\mathfrak{C}}(o))$.

The above definition only helps us to address the question whether a request is external in a particular configuration. However, since concurrent threads may dynamically change the state of the object or alias that a request is directed to, we need a more general definition to express that a request will be external in *all* reachable configurations.

There are two ways to check for self-surrogation: either (1) consider *all* surrogation requests in a given term, or (2) consider only one particular request under study and trace it during the computation that brings it to top-level, i.e., in redex position. While the former is quite restrictive, the latter faces the problem to ensure during the reduction sequence that we can keep track on what happens to a particular request under study; in fact, the object or variable that it is syntactically addressed to is going to be replaced during evaluation by some object reference, and the request itself might be installed as subterm of some other piece of code on a task different from the initial one, which can even happen several times.

An obvious technique to trace a request is to tag the subterm of interest with some meta-information that does not affect the semantics, e.g. by means of overlining. To this aim, we may extend the syntax with overlined operations for ping, and $\mathsf{surrogate}$ (there would not be a problem doing the same for invocation, update, cloning, and aliasing). For this extended syntax we disallow rules with overlined redexes, but we allow to perform substitutions on them possibly replacing a variable with a reference. If we then start the evaluation of a term with at most one overlined subterm, then the evaluation of the expression inside a task is automatically stopped whenever an overlined subterm appears as a redex at top-level. Note that there can appear several instances of the one overlined redex that we started out with in the case that it was a subterm of a method body, which creates a new copy each time it is called.

Definition 5.4.1 (Uniformly External Contexts). *Let $C[\cdot]$ be a context and $x.\mathsf{op}$ a request with $\emptyset \vdash C[x.\mathsf{op}] : B$. Then, $C[\cdot]$ is called* uniformly external *for $x.\mathsf{op}$, if for all $\mathfrak{C}$ with $[\![\, C[\overline{x.\mathsf{op}}] \,]\!] \rightarrow^* \mathfrak{C}$, if $\mathfrak{C}(t) = \langle\, p, s, e[\overline{o.\mathsf{op}}] \,\rangle$ for $t, o \in \mathrm{dom}(\mathfrak{C})$, then $s \neq \mathrm{end}(\mathrm{ali}_{\mathfrak{C}}(o))$.*

Fact 5.4.2. *Let x be an object variable and $C[\cdot]$ a context with $\emptyset \vdash C[x] : B$. Then $C[\cdot]$ is uniformly external for $x.\mathsf{ping}$ iff $C[\cdot]$ is uniformly external for $x.\mathsf{surrogate}$.*

5.5 Towards a Safety Theorem for Provers

According to the above remarks, we concentrate our efforts on reasoning about the equation $x.\mathsf{ping} =^{\vdash} x.\mathsf{surrogate}$ with respect to only those contexts that are uniformly external, because we conjecture that the forwarder models, especially the serialized version $\mathcal{S}$ will allow us to prove a theorem for these cases. According to Definition 4.3.1, this means:

Conjecture 5.5.1 ($\mathcal{S}$-Safety). *Let x be an object variable and $C[\cdot]$ a context with $\emptyset \vdash C[x] : B$. If $C[\cdot]$ is uniformly external for $x.\mathsf{op}$, then $C[x.\mathsf{ping}]\Downarrow$ iff $C[x.\mathsf{surrogate}]\Downarrow$.*

We restrain ourselves to call this conjecture a theorem, because we did not carry it out for the very notion of convergence as introduced in this paper, as we will explain in the following paragraph. Further comments on this matter can be found in the Conclusions.

In previous work on the Imperative Object Calculus (IOC) [AC96], equivalence between IOC terms was defined in a contextual way [GHL97], similar to Definition 4.3.1. In many cases, it seems simpler to use a semantics by translation into π-calculus [MPW92] to establish the equivalence between terms [KS98]. The main advantage is the large number of equivalences and algebraic laws that can be used to reason about expressions. Since IOC is (almost) a concurrency-free subset of Øjeblik, we chose a similar path for establishing the safety of external surrogation. An early sketch of a proof for a restricted version of Theorem 5.5.1, where only an inductively defined subset of 'external' contexts is taken into account, is found in [HKMN99], but the underlying π-calculus translation had several flaws. In our recent paper [MHKN99], we show the non-trivial proof based on a translation of Øjeblik that carefully follows the serialized model, and with respect to the corresponding notions of convergence and uniformly external contexts based on that π-calculus semantics.

5.6 Towards a Safety Theorem for Programmers

We argued earlier that it is the programmer of an object who is often responsible for potential problems caused by self-inflicted surrogation. In order to help a programmer to avoid these problematic cases, it would be useful to provide some syntactic criteria or programming guidelines, although they necessarily would tend to be rather restrictive. Note that the standard technique of restricting protected operations on the literal self, i.e., the innermost surrounding binding for s, does not work in the context of aliasing—see the above counterexamples. Yet, those examples of self-surrogation could be avoided if one was guaranteed that surrogation was the last operation performed on an object by the currently inhabiting tasks. If no operation is afterwards directed to s, then none of the above observable "mistakes" could be done. However, the fact that not only the current task may continue after the surrogation to operate on s, but also potentially other tasks in the call-stack that wait for the current task to return, complicates the quest for a syntactic criterion for a programmer.

There are two ways out of the dilemma of self-surrogation that we can imagine:

1. An application of data-flow analysis could verify that the self-inflicted surrogation is definitely the last operation called in the current self-inflicted method call-stack.

2. A type system could ensure that we never encounter self-surrogation at all, or at least provide us with warnings that self-surrogations might occur.

Here, we propose a solution based on the latter idea. The observation is that, if a evaluates to the current self, or to a node in an alias chain leading to the current self, then a *must* have the same type as the type of the current self (c.f. rule (T-PROXY) in Table 16).

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

$$\frac{\Gamma \vdash_D a{:}A \qquad A = [\mathrm{l}_j{:}\tilde{B}_j{\to}C_j]_{j\in J} \qquad \Gamma, s{:}A, \tilde{x}{:}\tilde{B}_k \vdash_A b{:}C_k \qquad k{\in}J}{\Gamma \vdash_D a.\mathrm{l}_k{\Leftarrow}\varsigma(s{:}A, \tilde{x}{:}\tilde{B}_k)b : A} \text{ (T-UPD)}$$

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

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

Table 16: Typing rules ensuring external surrogate operations

This implies, that if we ensure that the type of a is *not* the same as for the current self, then $a.\mathsf{op}$ cannot result in op being an internal operation. Such a check can be easily incorporated into the type system of Table 2. In the resulting system, judgments are of the form $\Gamma \vdash_D a{:}A$, where D denotes the type of the self variable for the method enclosing a. In Table 16 we present the modifications of the type system; the rules missing are as the ones in Table 2 with $\vdash$ replaced by $\vdash_D$.

The following theorem witnesses a static guarantee for uniformly external contexts provided by the type system. Here, as well as in the rule (T-FORK), we have to assign some current-self type to the top-level judgment. Since, by definition, threads do not have a current self, the natural choice is to use a thread type $\mathrm{Thr}(A)$ as the current-self type. The reason is that thread types never face the danger of being confused. Note that we could also simply use the type $\mathrm{Thr}([\,])$ in these cases.

Theorem 5.6.1. *Let x be a variable and $C[\cdot]$ be a program context with $\emptyset \vdash C[x] : A$. If $\emptyset \vdash_{\mathrm{Thr}(A)} C[x.\mathsf{surrogate}] : A$, then $C[\cdot]$ is uniformly external for $x.\mathsf{surrogate}$.*

Proof. The proof is by contradiction: Assume $\emptyset \vdash_{\mathrm{Thr}(A)} C[x.\mathsf{surrogate}] : A$ where $C[\cdot]$ is *not* uniformly external for $x.\mathsf{surrogate}$. Then, by Definition 5.4.1, there exists a configuration $\mathfrak{C}$ with $C[\,\overline{x.\mathsf{surrogate}}\,] \to^* \mathfrak{C}$, where $\mathfrak{C}(t) = \langle\, f, s, e[\,\overline{o.\mathsf{surrogate}}\,]\,\rangle$ for $t, o \in \mathrm{dom}(\mathfrak{C})$, but $s = \mathrm{end}(\mathrm{ali}_{\mathfrak{C}}(o))$. Here, the typing assumption for $C[x.\mathsf{surrogate}]$, which tells that o and s have different type, contradicts the fact that alias chains in configurations reachable from well-typed terms are homogeneously typed, which proves the theorem. $\square$

Let $=^{\vdash}_{\text{ext}}$ denote the equivalence of Definition 4.3.3 adapted to the new type system.

Conjecture 5.6.2. *Let x be an object variable. Then $x.\mathsf{ping} =^{\vdash}_{\text{ext}} x.\mathsf{surrogate}$.*

Proof. Immediate from Theorem 5.6.1, Lemma 5.4.2, and Conjecture 5.5.1. □

Note that Conjecture 4.3.5 does obviously not hold for the extended type system, because now the type system can no longer mimic any kind of misbehavior of ill-typed contexts; this fact is of course the whole purpose of the definition of the extended type system.

6 Conclusion

Résumé We have shown, by means of examples and the precise examination of their behavior according to formal semantics, that object surrogation in Øjeblik, and consequently object migration in Obliq, is not transparent. We have verified, using the Obliq interpreter [Car94], that these examples indeed expose problems with surrogation/migration in Obliq. Most of the migration problems were actually discovered, when trying to prove the safety of surrogation using π-calculus translations that implemented, first Obliq's semantics, then Talcott's semantics. Experimenting with these failed attempts led us to our final semantics for Øjeblik. The major improvement suggested by our studies is the use of *forwarder models* for the treatment of aliasing, which seems to be necessary in order to ensure transparency of, at least, external surrogation.

As pointed out in Subsection 4.1, the forwarder models can be understood as a good compromise between the concepts of self-infliction mutexes and of re-entrant mutexes to so-called *pre-entrant mutexes*: any request is allowed to re-enter an object, if it has only traveled through predecessors of the object since its last visit. A repaired version of Obliq should, by adopting the forwarder model, employ *object aliasing* rather as a primitive operation than merely derived from field aliasing. By that, the appropriate amount of serialization and protection for nodes in an alias chain can be implemented.

The major lesson learned from the work presented in this paper, is that *concurrent objects need formal analysis*—not only because one should prove properties for their own sake, but because formal analysis is a good debugging tool.

Language Design Issues Our formal discussion on the various aliasing models allows us to propose even more possibilities on how to model nodes in alias chains.

Since object aliasing had better be a primitive operation rather than derived from field aliasing, one may turn alias nodes into pure forwarders, even for local operations. The idea is that re-aliasing is useful for individual fields, but can be harmful for whole objects, where re-aliasing can anyway only occur as long as the node is active (cf. § 3.1). With field aliasing, re-aliasing could be initiated by activities of local siblings, which makes sense even after the current aliasing method has terminated. Note that in this proposal there is no need for protection as a requirement for surrogation since object aliases would be immediately stable if re-aliasing was prevented. (We do not question the general usefulness of protection, although this may be subject to further discussion.)

If the design of Obliq was not based on mutexes for self-infliction, but rather on reentrant mutexes, then the counterexample of Equation (9) would be resolved properly, but there would still be the other counterexample of Equation (10), where pre-entrant cloning would be needed, and also the counterexamples with respect to self-surrogation would still apply. Thus, if the concept of being reentrant would include protection in addition to serialization, then this design proposal would become as good as forwarder models, and even more liberal. However, the increased liberty of reentrant mutexes for serialization was already an argument for Cardelli to reject them for Obliq, and to prefer self-infliction. In this paper, we give formal and practical arguments for why pre-entrant mutexes and protection are the best choice from the point of view of transparent (external) migration.

Is *Migration = Cloning ; Aliasing* a good idea, after all? Since we completely ignored in our study the question of performance, or the feasibility of a run-time system to deal with ever-growing alias chains, it is hard to compare this style of migration to others that have been proposed, for example for Distributed Oz [VHB$^+$97]. (These matters go well beyond the scope of this paper, but we may note here that the compression of alias chains should be straightforward based on our formal notion of stable alias nodes.) However, we would like to emphasize that the intuitive simplicity of the concept does not quite compensate for the complexity that it imposes on formal reasoning, and also not for the sometimes unpredicted and surprising outcome of computations. It should be very interesting to study the safety of migration of other styles.

Current and Future Work We are about to use our semantics to show further properties about Øjeblik. For instance, we are about to show that $\mathsf{join}(\mathsf{fork}(a)) \cong a$ holds, although this may only be true under certain self-infliction conditions, and we also expect some laws of Moggi's computational λ-calculus [Mog89] to hold naturally. Similar properties have already been proved by Kleist and Sangiorgi [KS98] using a π-calculus semantics for the IOC, of which our translation in [MHKN99] is a rather direct generalization.

As noted in the Introduction, our conjectures on safe surrogation will only hold unless sites may fail, which is obvious by considering that the proxy site (where the object returned from $s.\mathsf{ping}$ resides) may fail, while the target site (where the object returned from $s.\mathsf{surrogate}$ resides) is still alive. Naturally, it would be interesting to study what safety of migration in a faulty setting could mean and in what precise sense it would be satisfied.

It would be challenging to exploit the operational semantics of the forwarder models, or suitable variants of it, directly for carrying out safety proofs. This is work in progress.

We sketch two more strands of ongoing work in the next paragraphs.

Operational Correspondence For the serialized aliasing model, we have given two different kinds of semantics, an operational semantics (OS) presented in this paper, and a semantics by translation into π-calculus (PI). An immediate question arises about the formal correspondence between these two semantics, which is even more critical in our case since, up to now, we have used PI for the main safety proof of the serialized model.

The basic complication to resolve is not a surprising one: the OS is very much based on a centralized global view of configurations, while the PI adopts a local view, where each node can only talk to its neighboring nodes, i.e., those to which it has references. For computations, this leads to situations where the PI engages in fully distributed interleaved step-by-step forwarding of requests along alias chains and possibly runs into local deadlocks, while the OS would recognize potential deadlock situations earlier and block the one-step forwarding of a request along the chain immediately. In a formalization of an operational correspondence between OS and PI, this discrepancy has to be taken into account properly.

The main property that the operational correspondence should yield is that convergence defined in the OS coincides with convergence in the PI, because convergence is what the chosen notion of typed equivalence is built upon. Based on the current state of affairs, we strongly believe to be able to achieve that goal.

Regaining Subtyping As we showed in Subsection 2.2, aliasing is not directly compatible with the subtyping rules of the IOC. The problem is that, in order to ensure that b is able to properly deal with all requests sent to a, when typing an alias operation $a.\mathsf{alias}\langle b\rangle$ we must know explicitly all the methods available in a. In this section, we briefly discuss a proposal to modify the type system, such that subtyping becomes possible again.

Our solution is to add some syntactic restrictions on the source a of an alias operation $a.\mathsf{alias}\langle b\rangle$. Since aliasing is a protected operation, a must be an expression that evaluates to (a node in a chain leading to) the current self. But since we know what a must evaluate to, in order for the aliasing operation to succeed, we may as well add the syntactic restriction that a must *be* the self variable of the enclosing method. By doing this we can use the fact that when typing an object we give the self variable a type containing *all* methods in the type environment. The only potential drawback we see in this proposal is that it prevents from pre-entrant aliasing, but this does not harm very much.

In Table 17, we show the rules for the modified type system. Here, a type judgment $\Gamma \vdash_s a : A$ now contains an annotation on the $\vdash$, which provides the syntactic information

$$\frac{-}{\Gamma, x{:}A \vdash_s x{:}A}\ \text{(S-VAR)} \qquad \frac{\Gamma \vdash_s a{:}A \qquad \Gamma, x{:}A \vdash_s b{:}B \qquad s \neq x}{\Gamma \vdash_s \mathsf{let}\, x{:}A = a \,\mathsf{in}\, b : B}\ \text{(S-LET)}$$

$$\frac{\forall j{\in}J \quad \Gamma, s_j{:}A, \tilde{x}_j{:}\tilde{B}_j \vdash_s b_j{:}C_j \qquad A = [\mathrm{l}_j{:}\tilde{B}_j{\rightarrow}C_j]_{j\in J}}{\Gamma \vdash_{s'} [\mathrm{l}_j : \varsigma(s_j{:}A, \tilde{x}_j{:}\tilde{B}_j)b_j]_{j\in J} : A}\ \text{(S-OBJ)}$$

$$\frac{\Gamma \vdash_{s'} s'{:}A \qquad A = [\mathrm{l}_j{:}\tilde{B}_j{\rightarrow}C_j]_{j\in J} \qquad \Gamma, s{:}A, \tilde{x}{:}\tilde{B}_k \vdash_s b{:}C_k \qquad k{\in}J}{\Gamma \vdash_{s'} s'.\mathrm{l}_k{\Leftarrow}\varsigma(s{:}A, \tilde{x}{:}\tilde{B}_k)b : A}\ \text{(S-UPD)}$$

$$\frac{\Gamma \vdash_s b{:}A \qquad A = [\mathrm{l}_j{:}A_j]_{j\in J} \qquad A <: \Gamma(s)}{\Gamma \vdash_s s.\mathsf{alias}\langle b\rangle : A}\ \text{(S-ALI)}$$

$$\frac{\Gamma \vdash_s a{:}A \qquad A = [\mathrm{l}_j{:}A_j]_{j\in J} \qquad \Gamma(s) \not<: A}{\Gamma \vdash_s a.\mathsf{surrogate} : A}\ \text{(S-SUR)}$$

$$\frac{I \subseteq J}{[\mathrm{l}_j{:}A_j]_{j\in J} <: [\mathrm{l}_i{:}A_i]_{i\in I}}\ \text{(S-SUB-OBJ)}$$

$$\frac{A <: B}{\mathrm{Thr}(A) <: \mathrm{Thr}(B)}\ \text{(S-SUB-THREAD)}$$

Table 17: Typing and Subtyping for Aliasing

about the variable that *is* the current self. Note also our proposal for the rule (S-SUR) to prevent from self-inflicted surrogation: whenever we call $a.\mathsf{surrogate}$, then it cannot be that the request arrives via forwarding at the current self s since the requirement $\Gamma(s) \not<: A$ forbids that s could a successor of the chain starting at a with type A.

Acknowledgments We are grateful for discussions with and remarks from Luca Cardelli, Davide Sangiorgi, Carolyn Talcott, and ν-klubben at BRICS Aalborg. Moreover, we thank the anonymous referees for the detailed reading and their constructive remarks, which have helped improving the paper.

References

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

[BFL98] M. Boreale, C. Fournet and C. Laneve. Bisimulations for the Join-Calculus. In D. Gries and W.-P. de Roever, eds, *Proceedings of PROCOMET '98*. International Federation for Information Processing (IFIP), Chapman & Hall, 1998.

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

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

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

[FF86] M. Felleisen and D. P. Friedman. Control Operators, the SECD-machine, and the λ-calculus. In M. Wirsing, ed, *Formal Description of Programming Concepts III*, pages 193–217. North-Holland, 1986.

[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.

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

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

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

[Lan96] C. Laneve. May and Must Testing in the Join-Calculus. Technical Report UBLCS-96-4, DoCS, University of Bologna, March 1996. Revised: May 1996.

[MHKN99] M. Merro, H. Hüttel, J. Kleist and U. Nestmann. Mobile Objects As Mobile Processes. Submitted. Available from `http://www.cs.auc.dk/research/FS/ojeblik/`, 1999.

[Mog89] E. Moggi. Computational Lambda Calculus and Monads. In *Fourth Annual Symposium on Logic in Computer Science (LICS) (Asilomar, California)*, pages 14–23. IEEE, Computer Society Press, June 1989.

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

[MPW92] R. Milner, J. Parrow and D. Walker. A Calculus of Mobile Processes, Part I/II. *Information and Computation*, 100:1–77, Sept. 1992.

[Nes99] U. Nestmann. Mobile Objects (A Project Overview). In K. Spies and B. Schätz, eds, *Proceedings of FBT '99: Formale Beschreibungstechniken für verteilte Systeme (München, Juni 1999)*, Wissenschaft, pages 155–164. Herbert Utz Verlag, 1999.

[NHKM99] U. Nestmann, H. Hüttel, J. Kleist and M. Merro. Aliasing Models for Object Migration. In *Proceedings of EUROPAR '99*, volume 1685 of *LNCS*. Springer, Sept. 1999.

[NOI+99] C. L. Nielsen, L. Outzen, P. M. Ilsøe, S. H. Pedersen, T. Thomadsen and T. Lange. *Øjeblink.* Term project report, Aalborg University, June 1999. Available from `http://www.cs.auc.dk/research/FS/ojeblik/`.

[Rep92] J. Reppy. *Higher-Order Concurrency.* PhD thesis, Cornell University, June 1992. Technical Report TR 92-1285.

[SY97] T. Sekiguchi and A. Yonezawa. A Calculus with Code Mobility. In *Prodeedings of FMOODS'97, Canterbury, July.* IFIP, Chapman and Hall, 1997.

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

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

[Vit99] J. Vitek. Mobile Object Systems: Homepage. `http://cuiwww.unige.ch/~ecoopws/`, 1999.

Recent BRICS Report Series Publications

RS-99-44 **Uwe Nestmann, Hans Hüttel, Josva Kleist, and Massimo Merro. *Aliasing Models for Mobile Objects*. December 1999. ii+46 pp. To appear in a special FOOL6 issue of *Information and Computation*. An extended abstract of this revision, entitled *Aliasing Models for Object Migration*, appeared as Distinguished Paper in Amestoy, Berger, Daydé, Duff, Frayssé, Giraud and Daniel, editors, *5th International Euro-Par Conference*, EURO-PAR '99 Proceedings, LNCS 1685, 1999, pages 1353–1368, which in turn is a revised part of another paper called *Migration = Cloning ; Aliasing* that appeared in Cardelli, editor, *Foundations of Object-Oriented: 6th International Conference*, FOOL6 Informal Proceedings, 1999 and as such supersedes the corresponding part of the earlier BRICS report RS-98-33.**

RS-99-43 **Uwe Nestmann. *What is a 'Good' Encoding of Guarded Choice?* December 1999. ii+34 pp. To appear in a special EXPRESS '97 issue of *Information and Computation*. This revised report supersedes the earlier BRICS report RS-97-45.**

RS-99-42 **Uwe Nestmann and Benjamin C. Pierce. *Decoding Choice Encodings*. December 1999. ii+62 pp. To appear in *Journal of Information and Computation*. An extended abstract appeared in Montanari and Sassone, editors, *Concurrency Theory: 7th International Conference*, CONCUR '96 Proceedings, LNCS 1119, 1996, pages 179–194.**

RS-99-41 **Nicky O. Bodentien, Jacob Vestergaard, Jakob Friis, Kåre J. Kristoffersen, and Kim G. Larsen. *Verification of State/Event Systems by Quotienting*. December 1999. 17 pp. Presented at *Nordic Workshop in Programming Theory*, Uppsala, Sweden, October 6–8, 1999.**

RS-99-40 **Bernd Grobauer and Zhe Yang. *The Second Futamura Projection for Type-Directed Partial Evaluation*. November 1999. Extended version of an article to appear in Lawall, editor, *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, PEPM '00 Proceedings, 2000.**

RS-99-39 **Romeo Rizzi. *On the Steiner Tree $\frac{3}{2}$-Approximation for Quasi-Bipartite Graphs*. November 1999. 6 pp.**