Object Connectivity in a Concurrent Calculus of Classes

— Extended Abstract —

September 26, 2003

Erika Ábrahám[1,2], Marcello M. Bonsangue[3], Frank S. de Boer[4], and Martin Steffen[1]

[1] Christian-Albrechts-University Kiel, Germany
[2] University Freiburg, Germany
[3] University Leiden, The Netherlands
[4] CWI Amsterdam, The Netherlands

Abstract. The concurrent ν-calculus has been investigated as a core calculus for imperative, object-oriented languages with multithreading and heap-allocated objects. From an abstract point of view, the combination of this form of concurrency with objects corresponds to features known from the popular language *Java*. One distinctive feature, however, of the concurrent object calculus is that it is *object-based*, whereas the mainstream of object-oriented languages is *class-based*.
This work extends the concurrent ν-calculus by introducing classes and exploring some of the semantical consequences. The external behavior of a component, given as a set of threads and objects, can be described in a linear-time setting as the set of traces of interactions with the environment. Considering classes as part of the component makes instantiation a possible interaction between component and environment. A striking consequence of this new interaction is that we need to take into account *connectivity* information, i.e., the way objects may have knowledge of each other, in order to get a precise characterization of possible traces. We formulate an operational semantics that incoporates the connectivity information into the scoping mechanism of the ν-calculus. Since instantiation itself is to be considered as non-observable, we formulate a semantics where objects are instantiated only when they are accessed for the first time ("*lazy instantiation*").

1 Introduction

The concurrent ν-calculus has been proposed as core calculus for imperative, object-oriented languages with multithreading and heap-allocated objects. In the context of concurrent, *object-based* programs and starting from may-testing as a very simple notion of observation, Jeffrey and Rathke [4] provide a fully abstract trace semantics for the language. Their result roughly says that, given a *component* as a set of objects and threads, the fully abstract semantics consists

of the set of traces at the boundary of the component, where the traces record incoming and outgoing calls and returns. At this level, the result is as one would expect, since intuitively in the chosen setting, the only possible way to observe something about a set of objects and threads is by exchanging messages. It should be equally clear, however, that for the language featuring multithreading, object references with aliasing, and creation of new objects and threads, the details of defining the semantics and proving the full abstraction result are far from trivial.

The result in [4] is developed within the concurrent ν-calculus [3], an extension of the sequential ν-calculus [6] which stands in the tradition of various object calculi [1] and also of the π-calculus [5, 7]. One distinctive feature of the ν-calculus is that it is *object-based,* which in particular means that there are no *classes* as templates for new objects. This is in contrast to the mainstream of object-oriented languages where the code is organized in classes. This report addresses therefore the following question:

> What changes when switching from an object-based to a class-based setting?

Considering the observable behavior of a component, we have to take into account that in addition to objects, which are the passive entities containing the instance state and the methods, and threads, which are the active entities, *classes* come into play. Classes serve as a blueprint for their instances and can be conceptually understood as particular objects supporting just a method which allows to generate instances.

Important in our context is that now the division between the program fragment under observation and its environment also separates *classes:* There are classes internal to the component and those belonging to the environment. As a consequence, not only calls and returns are exchanged at the interface between component and environment, but instantiation requests, as well. This possibility of *cross-border instantiation* is absent in the object-based setting: Objects are created by directly providing the code of their implementation, not referring to the name of a class, which means that the component creates only component-objects and dually the environment only environment objects. To understand the bearing of this change on the semantics, we must realize that the interesting part of the problem is not so much to just cover the possible behavior at the interface —there is little doubt that sequences of calls, returns, and instantiations with enough information at the labels would do— but to characterize it *exactly,* i.e., to exclude impossible environment interaction. As an obvious example, a trace with two consecutive calls from the same thread without outgoing communication in between cannot be part of the component behavior.

Let' s concentrate on the issue of instantiation across the demarcation line between component and its environment, and imagine that the component creates an instance of an environment class. The first question is: does this yield a component object or an environment object? As the code of the object is provided by the external class which is in the hand of the observer, the interaction between the component and the newly created object can lead to observable ef-

fects and must thus be traced. In other words, instances of environment classes belong to the environment, and those of internal classes to the component.

Whereas in the above situation, the object is instantiated to be part of the environment, the *reference* to it is kept at the creator, for the time being. So in case, an object of the program, say o_1 instantiates two objects o_2 and o_3 of the environment, the situation informally looks as shown in Figure 1, where the dotted bubbles indicate the scope of o_2 respectively o_3.

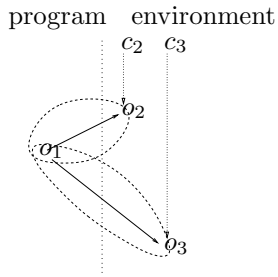

Fig. 1. Instances of external classes

In this situation it is impossible, that there be an incoming call from the environment carrying both names o_2 and o_3, as the only entity aware of both references is o_1. Unless the component gives away the reference to the environment, o_2 and o_3 are completely separated.

Thus, in order to exclude impossible combinations of object reference in the communication labels, the component has to keep track which objects of the environment are connected. The component has, of course, by no means full information about the complete system; after all it can at most trace what happens at the interface, and the objects of the environment can exchange information "behind the component's back". Therefore, the component must conservatively over-approximate the potential knowledge of objects in the environment, i.e., it must make *worst-case assumptions* concerning the proliferation of knowledge, which means it must assume that

1. once a name is out, it is never forgotten, and
2. if there is a possibility that a name is leaked from one environment object to another, this will happen.

Sets of environment objects which can possibly be in contact with each other form therefore equivalence classes of names —we call them *cliques*— and the formulation of the semantics must include a representation of them. New cliques can be created, as new objects can be instantiated without contact to others, and furthermore cliques can merge, if the component leaks the identity of a member of one clique to a member of another.

This extended abstract investigates a class-based variant of the concurrent ν-calculus, formalizing the ideas sketched above about cliques of objects. Instantiation itself, even across the environment-program boundary, is unobservable, since the calculus does not have constructor methods. In the semantics, an externally instantiated object is created only at the point, when it is actually accessed the for the first time, which we call "*lazy instantiation*". For want of space, we concentrate here on the intuition and, in the formal part, the operational semantics describing the external behavior of a component. For deeper coverage we refer to the technical report [2]; the derivation rules for typing and for the operational semantics are included also in the appendix for reference.

2 A concurrent class calculus

In this section, we present the calculus used in our development. As we concentrate on the semantical issues of connectivity of objects and the interface behavior of a component, we gloss over the exact syntax, ignore typing issues and also omit structural equivalence rules, as they are rather standard. As mentioned, the reader will find details in the appendix and the accompanying technical report.

The calculus is a syntactic extension of the concurrent object calculus or concurrent ν-calculus from [3, 4]. The basic change is the introduction of *classes,* where a class is a named collection of methods. In contrast to object references, class names are literals introduced when defining the class; they may be hidden using the ν-binder but unlike object names, the scopes for class names are *static.* Object names, on the other hand, are first-order citizens of the calculus in that they can be stored in variables, passed to other objects as method parameters, making the scoping *dynamic,* and especially they can be created freshly by instantiating a class.

A program is given by a collection of classes. A class $c[\![O]\!]$ carries a name c and defines the implementation of its methods, and analogously for objects. A method $\varsigma(n{:}T).\lambda(x_1{:}T_1, \ldots, x_n{:}T_k).t$ provides the definition of the method body abstracted over the formal parameters of the method and the ς-bound "self" parameter [1]. Besides named objects and classes, the dynamic configuration of a program can contain as active entities *named threads* $n\langle t\rangle$, which, like objects, can be dynamically created. Unlike objects, threads are not instantiated by some statically named entity (a "thread class"), but directly created by providing the code. A thread, whose exact syntax is of no particular interest here, basically is either a value (especially a reference to another named entity) or a sequence of expressions, notably method calls (written $o.l(\vec{v})$) and creation of new objects and new threads ($new\, c$ and $new\langle t\rangle$ where c is a class name and t a thread). We will generally use n and its syntactic variants as name for threads (or just in general for names), o for objects, and c for classes. Furthermore we will use f specifically for instance variables or fields, we use $f = v$ for field variable declaration, field access is written as $x.f$, and field update as $x := v$.[5]

Concerning the *operational semantics* of the calculus, the basic steps are given in two levels: *internal* steps whose effect is completely confined within a configuration, and those with external effect. Interested mainly in the external behavior we elide the definition of the internal steps (but see Appendix A.3).

The *external* behavior of a component is given in terms of labeled transitions describing the communication at the interface of an *open* program. For the completeness of the semantics, it is crucial ultimately to consider only communication traces realizable by an actual program context which, together with the component, yields a well-typed closed program.

[5] In the class-based setting we don't use general method update as supported by the object-based ν-calculus.

The concentration on actually realizable traces has various aspects, e.g., the transmitted values needs to adhere to the static typing assumptions, only publicly known objects can be called from the outside, and the like. Being concerned in the dynamic relationship among objects, we omit also these aspects here. Besides that, this part is rather standard and also quite similar to the one in [4].

2.1 Connectivity contexts and cliques

The informal discussion in the introduction argued that in the presence of internal and external classes and cross-border instantiation, the component must *keep track* of which identities it gives away to which objects in order to exclude impossible as described for instance in connection with Figure 1. The external semantics is formalized as labeled transitions between judgments of the from

$$\Delta; E_\Delta \vdash C : \Theta; E_\Theta \ , \tag{1}$$

where $\Delta; E_\Delta$ are the *assumptions* about the environment of the component C and $\Theta; E_\Theta$ the *commitments.* The assumptions consists of a part Δ concerning the existence (plus static typing information) of *named entities* in the environment. For the book-keeping of which objects of the environment have been told which identities, a well-typed component must take into account the *relation* of object names from the assumption context Δ amongst each other, and the knowledge of objects from Δ about those exported by the component, i.e., those from Θ. [6] In analogy to the name contexts Δ and Θ, E_Δ expresses assumptions about the environment, and E_Θ commitments of the component.

$$E_\Delta \subseteq \Delta \times (\Delta + \Theta) \ . \tag{2}$$

and dually $E_\Theta \subseteq \Theta \times (\Theta + \Delta)$. We will write $o_1 \hookrightarrow o_2$ ("o_1 may know o_2") for pairs from these relations. The component has by no means full information about the complete system; after all it can at most trace what happens at the interface, and the objects of the environment can exchange information "behind the component's back". Thus it must conservatively overapproximate the potential knowledge of objects in the environment, i.e., it must make *worst-case assumptions* concerning the proliferation of knowledge by assuming:

1. once a name is out, it is never forgotten, and
2. if there is a possibility that a name is leaked from one environment object to another, this will happen.

More technically, the worst case assumptions about the actual situation are represented as the *reflexive, transitive,* and *symmetric* closure of the $\hookrightarrow$-pairs of

[6] Besides the relationships amongst objects, we need to keep one piece of information concerning the "connectivity" of *threads.* To exclude situations where a known thread leaves the component into one clique of objects but later returns to the component coming from a different clique without connection to the first, we remember for each thread that has left the component the object from Δ it has left into.

objects *from* Δ the component maintains. Given Δ, Θ, and E_Δ, we write $\leftrightharpoons$ for this closure, i.e.,

$$\leftrightharpoons \triangleq (\hookrightarrow\!\!\downarrow_\Delta \cup \hookleftarrow\!\!\downarrow_\Delta)^* \subseteq \Delta \times \Delta \ . \tag{3}$$

Note that we close the part of $\hookrightarrow$ concerning only environment objects from Δ, but not wrt. objects at the *interface,* i.e., the part of $\hookrightarrow \subseteq \Delta \times \Theta$. We will also need the union of $\leftrightharpoons \cup \leftrightharpoons; \hookrightarrow \subseteq \Delta \times (\Delta + \Theta)$, for which we will also write $\leftrightharpoons\!\hookrightarrow$. As judgment, we use $\Delta; E_\Delta \vdash v_1 \leftrightharpoons v_2 : \Theta$ respectively $\Delta; E_\Delta \vdash v_1 \leftrightharpoons\!\hookrightarrow v_2 : \Theta$. For Θ, E_Θ, and Δ, the definitions are applied dually.

The relation $\leftrightharpoons$ is an equivalence relation on the objects from Δ and partitions them in equivalence classes. As a manner of speaking, we call a set of object names from Δ (or dually from Θ) such as for all objects o_1 and o_2 from that set, $\Delta; E_\Delta \vdash o_1 \leftrightharpoons o_2 : \Theta$, a *clique*, and if we speak of *the* clique of an object we mean the whole equivalence class.

2.2 External steps

As mentioned before, the external semantics is given by transitions between $\Delta; E_\Delta \vdash C : \Theta; E_\Theta$ judgments. Table 1 for the exchange of free names and in Table 2 dealing with bound names. Beside internal steps a component exchanges information with the environment by transitions; the core labels γ are of the form $n\langle[o]\mathit{call}\ o.l(\vec{v})\rangle$ and $n\langle \mathit{return}(v)\rangle$; names may occur bound in a label $\nu(n{:}T).\gamma$, and sending and receiving labels are written as $\gamma?$ and $\gamma!$.

The component exchanges information with the environment via *calls* and *returns.* Using a lazy instantiation scheme for cross-border object creation, there are no separate external labels for *new*-steps. In the extended abstract, we omit the static typing premises in the operational rules as they are straightforward and we concentrate in the discussion on the novel aspects, namely the connectivity information. The external steps are given in Tables 1 and 2.

As for the relationship of communicated values, incoming and outgoing communication play dual roles: E_Θ overapproximates the actual connectivity of the component, while the assumption context E_Δ is consulted to exclude impossible combinations of incoming values. For incoming communication (cf. rule $\textsc{CalleI}_2$ and $\textsc{RetI}$)[7] we require that the sender be acquainted with the transmitted arguments. In case of a call, the caller o_1 must additionally be acquainted with the callee o_2 and furthermore the calling thread must originate from a clique of objects in connection with the one to which the thread had left the component the last time: $\Delta; E_\Delta \vdash n \leftrightharpoons [o_1] : \Theta$.[8] To assure these connectivity conditions, the identity of the callee has been remembered as part of the block-syntax when the call was issued. It is worth mentioning that in rule $\textsc{RetI}$ the proviso that the callee o_2 knows indirectly the caller o_1, i.e., $\Delta; E_\Delta \vdash o_2 \leftrightharpoons\!\hookrightarrow o_1 : \Theta$ is not needed. Neither is it necessary to require in analogy to the situation for the incoming call

[7] We omit rule $\textsc{CalleI}_1$ which deals with a thread that entering for the first time. The rule is identical to $\textsc{CalleI}_2$ as far as the treatment of E_Δ and E_Θ is concerned.

[8] Since the caller o_1 is in the domain of Δ, we can write $n \leftrightharpoons [o_1]$ instead of $n \leftrightharpoons\!\hookrightarrow [o_1]$.

$$\frac{\begin{array}{c} E_\Delta \vdash [o_1] \leftrightharpoons\hookrightarrow \vec{v} : \Theta \qquad E_\Delta \vdash [o_1] \leftrightharpoons\hookrightarrow o_2 : \Theta \qquad E_\Delta \vdash n \leftrightharpoons [o_1] : \Theta \\ \acute{E}_\Delta = E_\Delta \setminus n \qquad \acute{E}_\Theta = E_\Theta + (o_2 \hookrightarrow \vec{v}, n \hookrightarrow o_2) \end{array}}{\begin{array}{l} \Delta; E_\Delta \vdash C \parallel n\langle let\, x{:}T' = [o_2]\ blocks\ for\ o_2'\ in\, t\rangle : \Theta; E_\Theta \xrightarrow{n\langle [o_1] call\ o_2.l(\vec{v})\rangle ?} \\ \Delta; \acute{E}_\Delta \vdash C \parallel n\langle let\, x{:}T = o_2.l(\vec{v})\ in\ return[o_1]\ x; let\, x'{:}T' = [o_2]\ blocks\ for\ o_2'\ in\, t\rangle : \Theta; \acute{E}_\Theta \end{array}} \text{CALLI}_2$$

$$\frac{\acute{E}_\Delta = E_\Delta + ([o_1] \hookrightarrow v, n \hookrightarrow [o_1]) \qquad \acute{E}_\Theta = E_\Theta \setminus n}{\Delta; E_\Delta \vdash C \parallel n\langle let\, x{:}T = return[o_1]\ v\ in\, t\rangle : \Theta; E_\Theta \xrightarrow{n\langle return(v)\rangle !} \Delta; \acute{E}_\Delta \vdash C \parallel n\langle t\rangle : \Theta; \acute{E}_\Theta} \text{RETO}$$

$$\frac{o_2 \in \Delta \qquad \acute{E}_\Delta = E_\Delta + (o_2 \hookrightarrow \vec{v}, n \hookrightarrow o_2) \qquad \acute{E}_\Theta = E_\Theta \setminus n}{\begin{array}{l} \Delta; E_\Delta \vdash C \parallel n\langle let\, x{:}T = [o_1]\ o_2.l(\vec{v})\ in\, t\rangle : \Theta; E_\Theta \xrightarrow{n\langle [o_1] call\ o_2.l(\vec{v})\rangle !} \\ \Delta; \acute{E}_\Delta; \vdash C \parallel n\langle let\, x{:}T = [o_1]\ blocks\ for\ o_2\ in\, t\rangle : \Theta; E_\Theta \end{array}} \text{CALLO}$$

$$\frac{\begin{array}{c} ; \Delta, \Theta \vdash v : T \qquad \Delta; E_\Delta \vdash o_2 \leftrightharpoons\hookrightarrow v : \Theta \qquad \Delta; E_\Delta \vdash n \hookrightarrow o_2 : \Theta \\ \acute{E}_\Delta = E_\Delta \setminus n \qquad \acute{E}_\Theta = E_\Theta + (o_1 \hookrightarrow v, n \hookrightarrow [o_1]) \end{array}}{\begin{array}{l} \Delta; E_\Delta \vdash C \parallel n\langle let\, x{:}T = [o_1]\ blocks\ for\ o_2\ in\, t\rangle : \Theta; E_\Theta \xrightarrow{n\langle return(v)\rangle ?} \\ \Delta; \acute{E}_\Delta \vdash C \parallel n\langle t[v/x]\rangle : \Theta; \acute{E}_\Theta \end{array}} \text{RETI}$$

Table 1. External steps (free core labels)

that the thread is acquainted with the callee. If fact, both requirements will be automatically assured for traces where calls and return occur in correct manner. A commonality for incoming communication from a thread n is that the (only) pair $n \hookrightarrow o$ for some object reference o is removed from E_Δ, for which we write $E_\Delta \setminus n$. While E_Δ imposes restrictions for incoming communication, the commitment context E_Θ is *updated* when receiving new information. For instance in CALLI_2, the commitment $\acute{E}_\Theta$ after reception marks that now the callee o_2 is acquainted with the received arguments and furthermore that the thread n is visiting (for the time being) the callee o_2. For *outgoing* communication, the E_Δ and E_Θ play dual roles. Note further that E_Θ is not mentioned (except for the connectivity of the thread identity n).

2.3 Scoping, lazy instantiation, and guessing of connectivity

Next we discuss the exchange of new names by means of scope extrusion and intrusion and related to that instantiation across the component boundary (cf. Table 2). Besides bound names in the form $\nu(n{:}T)$, the labels contain also *connectivity* information which is exchanged. This means the labels are of the form $\nu(n{:}T, E_\Theta).\gamma!$ of outgoing and $\nu(n{:}T, E_\Delta).\gamma?$ for incoming communication.

As expected, the ν-binder influences only names occurring freely in the label (cf. rule COMM). In case of a *bound input* with label $\nu(n{:}T, \tilde{E}_\Delta).\gamma?$ in rule BIN, the name's scope is extruded into the component. The treatment of the knowledge base E_Δ merits a closer look. As specified in $\tilde{E}_\Delta \subseteq \Delta' + (\Delta' \times \Theta)$, a new

$$\frac{n \notin fn(a) \qquad \Delta; E_\Delta \vdash C : \Theta, n{:}T \xrightarrow{a} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}, n{:}T; \acute{E}_\Theta}{\Delta; E_\Delta \vdash \nu(n{:}T).C : \Theta \xrightarrow{a} \Delta'; E'_\Delta \vdash \nu(n{:}T).C' : \acute{\Theta}; \acute{E}_\Theta \setminus n}\ \textsc{Comm}$$

$$\frac{\begin{array}{c} n \in fn(\gamma) \qquad \Delta' = \Delta, n{:}T \quad E'_\Delta = E_\Delta + \tilde{E}_\Delta \quad \Delta; E_\Delta \vdash \Delta'; E'_\Delta \downarrow_{\Delta\times(\Delta+\Theta)} : \Theta \\ \Delta'; E'_\Delta \vdash C : \Theta \xrightarrow{\gamma?} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta \end{array}}{\Delta; E_\Delta \vdash C : \Theta \xrightarrow{\nu(n:T;\tilde{E}_\Delta).\gamma?} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta}\ \textsc{In}$$

$$\frac{\begin{array}{c} n \in fn(\gamma) \qquad \Delta; E_\Delta \not\vdash T : [\![\ldots]\!] \\ \Delta; E_\Delta \vdash (C \setminus n) \ : \Theta, n{:}T; E_\Theta + E_\Theta(C, n) \xrightarrow{\gamma!} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta \end{array}}{\Delta; E_\Delta \vdash \nu(n{:}T).C : \Theta; E_\Theta \xrightarrow{\nu(n:T;E_{\overline{\Theta}}^{\leftrightharpoons\hookrightarrow}(C,n)).\gamma!} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta}\ \textsc{BOut}$$

$$\frac{\begin{array}{c} o \in fn(\gamma) \qquad \Theta \vdash c : [\![\ldots]\!] \qquad C(c) = [\![O]\!] \\ \Theta' = \Theta, o{:}c \qquad E'_\Delta = E_\Delta + \tilde{E}_\Delta \qquad \Delta; E_\Delta \vdash \Delta; E'_\Delta \downarrow_{\Delta\times(\Delta+\Theta)} : \Theta \\ \Delta; E'_\Delta \vdash C \parallel o[O] : \Theta'; E_\Theta \xrightarrow{\gamma?} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta \end{array}}{\Delta; E_\Delta \vdash C : \Theta \xrightarrow{\nu(o:c;\tilde{E}_\Delta).\gamma?} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta}\ \textsc{In}_{new}$$

$$\frac{\begin{array}{c} o \in fn(\gamma) \qquad \Delta \vdash c : [\![\ldots]\!] \\ \Delta, o{:}c; E_\Delta \vdash C : \Theta; E_\Theta + E_{\overline{\Theta}}^{\leftrightharpoons\hookrightarrow}(C, n) \xrightarrow{\gamma!} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta \end{array}}{\Delta; E_\Delta \vdash \nu(o : c).C : \Theta; E_\Theta \xrightarrow{\nu(o:c;E_{\overline{\Theta}}^{\leftrightharpoons\hookrightarrow}(C,n)).\gamma!} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta}\ \textsc{BOut}_{new}$$

Table 2. External steps (scoping)

name of an object is related to *some* other objects. Now the gist is to understand that while the rule guesses which acquaintances the new object has, it is not completely free to do so! Since E_Δ is maintained as a worst-case assumption about the connectivity of the known external objects, learning about of a fresh object must not contradict this assumption. Intuitively, by creating new objects, which are initially unknown to the component, the environment cannot contact objects it could not contact otherwise. This restriction is captured in the proviso

$$\Delta; E_\Delta \vdash \acute{\Delta}; \acute{E}_\Delta \downarrow_{\Delta\times(\Delta+\Theta)} : \Theta \ , \tag{4}$$

where $\Delta' = \Delta, n{:}T$ in the rule, which requires that the addition of connectivity of the new identity n added to Δ may not lead to new derivable equations for the objects previously known. The requirement, $\Delta; E_\Delta \vdash E'_\Delta \downarrow_{\Delta\times(\Delta+\Theta)} : \Theta$ thus stands for the implication: If $\Delta'; E'_\Delta \vdash o_1 \leftrightharpoons\hookrightarrow o_2 : \Theta$, then $\Delta; E_\Delta \vdash o_1 \leftrightharpoons\hookrightarrow o_2 : \Theta$, for all o_1 from Δ and o_2 from Δ, Θ. In other words, E'_Δ is a *conservative* extension of E_Δ wrt. the old objects. More colloquially, incoming communication brings "no news about old objects" as we assume the worst already.

The rule BOut for bound output is similar, except that here we can consult the component C and especially the "$\hookrightarrow$"-parts of it to determine the connectivity part of the label: We add the *all* names from $\Theta' \times (\Theta' + \Delta)$ where $\Theta' = \Theta, n{:}T$ which according to C are acquainted with n. The set of acquainted objects are

those in the reflexive, transitive, and symmetric closure in the sense of Equation 3 from. We write $E^{\leftrightharpoons\hookrightarrow}(C, n)$ for that set of names. With the scope opened, we remove from C all $\hookrightarrow$-pairs mentioning n.

Object creation across component boundary is not immediately visible. Instead, new objects are actually created only when first communicated to or used by the other side. We call the mechanism *lazy instantiation.* If the component creates an instance of an external class $c \in \Delta$, a new reference o_3 is generated locally. Additionally it is remembered in a separate "parallel component" $o_1 \hookrightarrow o_3$ that the creator o_1 may now know o_3. The $\rightsquigarrow$-step itself is not externally visible:

$$\frac{c \in \Delta}{\begin{array}{l}\Delta; E_\Delta \vdash n\langle \mathit{let}\, x{:}c = [o_1]\ \ \mathit{new}\, c\, \mathit{in}\, t\rangle : \Theta; E_\Theta \rightsquigarrow \\ \Delta; E_\Delta \vdash \nu(o_3{:}c).o_1 \hookrightarrow o_3 \parallel n\langle \mathit{let}\, x{:}c = o_3\, \mathit{in}\, t\rangle : \Theta; E_\Theta\end{array}}\ \textsc{NewO}_{lazy}$$

If now the accesses the provisionally created object, it is an external step where the new object is mentioned in the label (cf. rule $\textsc{BOut}_{new}$). In contrast to ordinary scope extrusion in BOut, the one for lazy instantiation extends the *assumption* context Δ. Since furthermore the object is *unknown* in the environment, the assumption context E_Δ is used unchanged in the premise of the rule before the output step. The rule $\textsc{BIn}_{new}$ works symmetric, where the component incorporates the component class instance upon request from the environment.

3 Conclusion

In this report we presented, inspired by the work of [4], an operational semantics of a class-based, object-oriented calculus with multithreading. The seemingly innocent step from an *object-based* setting as in [4] to a framework with classes requires quite some extension in the operational semantics to characterize the possible behavior of a component. In particular it is necessary to keep track of the potential *connectivity* of objects of the environment to exclude impossible communication labels.

It is therefore instructive, to review the differences in this conclusion, especially to try to understand how the calculus of [4] can be understood as a special case of the framework explored here.

The fundamental dichotomy underlying the observational definition of equivalence is the one between the inside and the outside: program or component vs. environment or observer. This leads to the crucial difference between object-based languages, instantiating from objects, and class-based language, instantiating from classes: In the class-based setting, instantiation may *cross the demarcation line between component and environment,* while in the object-based setting, this is not possible: the program only instantiates program objects, and the environment only objects belonging to the environment. All other complications, expounded here at some great length, follow from this difference. The

most visible complication is that it is necessary to represent the dynamic object structure into the semantics, or rather an approximation of the connectivity of the environment objects. Another way to see it is, that in the setting of [4], there is only *one clique* in the environment, i.e., in the worst case, which is the relevant one, all environment objects are connected with each other. Since the component cannot create environment objects (or vice versa), never new isolated cliques are created. The object-based case can therefore be understood by invariantly (and trivially) taking $E_\Delta = \Delta \times (\Delta + \Theta)$, while in our setting, E_Δ may be more specific.

We see this study of the semantics as a step towards a full-abstraction result for the class-based calculus. Other future work is to extend the language and the semantics in a number of ways. One inherent feature of the calculus is that objects are *input enabled.* This disallows to model directly *synchronized methods* as in *Java*. Another generalization is to consider *cloning* of objects, i.e., to create a replica of an object In a certain way, *instantiation* of a class is just like cloning with the restriction that only objects in their initial state can be obtained by instantiation, while cloning can be applied to an object in mid-life. The ability to create an object in a state different from the initial one makes new observations possible, most notably the branching structured gets exposed. One therefore has to generalize the *linear-time* framework of traces to a *branching-time* view. More challenging is to take seriously the notion of classes in that they are not only considered as generator of new objects by instantiation, but also as template for new classes, i.e., to consider inheritance and subtyping. This makes new "observations" on classes possible, namely by subclassing.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects.* Monographs in Computer Science. Springer, 1996.
2. E. Ábrahám, M. M. Bonsangue, F. S. de Boer, and M. Steffen. A structural operational semantics for a concurrent class calculus. Technical Report 0307, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, Aug. 2003.
3. A. D. Gordon and P. D. Hankin. A concurrent object calculus: Reduction and typing. In U. Nestmann and B. C. Pierce, editors, *Proceedings of HLCL '98*, volume 16.3 of *Electronic Notes in Theoretical Computer Science.* Elsevier Science Publishers, 1998.
4. A. Jeffrey and J. Rathke. A fully abstract may testing semantics for concurrent objects. In *Proceedings of LICS '02.* IEEE, Computer Society Press, July 2002.
5. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part I/II. *Information and Computation*, 100:1–77, Sept. 1992.
6. A. M. Pitts and D. B. Stark. Observable properties of higher-order functions that dynamically create local names, or: What's new. In A. M. Borzyszkowski and S. Sokołowski, editors, *Proceedings of MFCS '93*, volume 711 of *Lecture Notes in Computer Science*, pages 122–141. Springer-Verlag, Sept. 1993.
7. D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes.* Cambridge University Press, 2001.

A Appendix

The appendix contains further material, especially the formalization of the type system and the operational semantics for internal component behavior. Further material and discussions can be found in the technical report [2].

A.1 Syntax

Table 3 and 4 give the definitions of the types and the abstract syntax. For the types $[\![l_1{:}U_1, \ldots, l_k{:}U_k]\!]$ is the type of a class, whose instances support methods labeled l_1 to l_k of the indicated types, and $[l_1{:}U_1, \ldots, U_k]$ the type of corresponding instances.

$$
\begin{aligned}
T &::= B \mid \mathit{none} \mid \mathit{thread} \mid [l{:}U, \ldots, l{:}U] \mid n \mid [\![l{:}U, \ldots, l{:}U]\!] \\
U &::= T \times \ldots \times T \to T
\end{aligned}
$$

Table 3. Types

C	$::= \mathbf{0} \mid C \parallel C \mid \nu(n{:}T).C \mid n[\![O]\!] \mid n[O] \mid n\langle t\rangle$	program
O	$::= l = m, \ldots, l = m$	object
m	$::= \varsigma(n{:}T).\lambda(x{:}T, \ldots, x{:}T).t$	method
t	$::= v \mid \mathit{stop} \mid \mathit{let}\, x{:}T = e\, \mathit{in}\, t$	thread
e	$::= t \mid \mathsf{if}\, v = v\, \mathsf{then}\, e\, \mathsf{else}\, e$	expr.
	$\mid\ v.l(v, \ldots, v) \mid n.l \Leftarrow m \mid \mathit{currentthread}$	
	$\mid\ \mathit{new}\, n \mid \mathit{new}\langle t\rangle$	
v	$::= x \mid n$	values

Table 4. Abstract syntax

A.2 Type system

The type system or static semantics presented next characterizes the well-typed programs. The derivation rules are given in Table 5 and 6.

A.3 Internal steps

For the component-internal steps from Table 7 we distinguish, as in [4], confluent $\rightsquigarrow$-step and competing τ-steps, i.e., steps that may lead to race conditions. Both kind of steps are invisible to the outside.

$$\frac{}{\Delta \vdash \mathbf{0} : ()}\ \text{T-Empty} \qquad \frac{\Delta, \Theta_2 \vdash C_1 : \Theta_1 \qquad \Delta, \Theta_1 \vdash C_2 : \Theta_2}{\Delta \vdash C_1 \parallel C_2 : \Theta_1, \Theta_2}\ \text{T-Par}$$

$$\frac{\Delta \vdash C : \Theta, n{:}T}{\Delta \vdash \nu(n{:}T).C : \Theta}\ \text{T-Nu}$$

$$\frac{;\Delta, c{:}T \vdash [\![O]\!] : T}{\Delta \vdash c[\![O]\!] : (c{:}T)}\ \text{T-NClass} \qquad \frac{;\Delta, o{:}c \vdash [O] : [T] \qquad ;\Delta \vdash c : [\![T]\!]}{\Delta \vdash o[O] : (o{:}c)}\ \text{T-NObj}$$

$$\frac{;\Delta, n{:}\,\mathit{thread} \vdash t : \mathit{none}}{\Delta \vdash n\langle t\rangle : (n{:}\,\mathit{thread})}\ \text{T-NThread}$$

Table 5. Static semantics (components)

A.4 External steps

In contrast to the abridged presentation in Section 2.2 we include here the full rules for external steps. In particular, the static typing premises are listed. The full rules are shown in Table 9.

$$\frac{\Gamma;\Delta \vdash m_1{:}T_1 \quad \ldots \quad \Gamma;\Delta \vdash m_k{:}T_k \qquad T = [\!(l_1{:}T_1, \ldots, l_k{:}T_k)\!]}{\Gamma;\Delta \vdash [\!(l_1 = m_1, \ldots, l_k = m_k)\!] : T}\ \text{T-Class}$$

$$\frac{\Gamma;\Delta \vdash m_1{:}T_1 \quad \ldots \quad \Gamma;\Delta \vdash m_k{:}T_k \qquad T = [l_1{:}T_1, \ldots, l_k{:}T_k]}{\Gamma;\Delta \vdash [l_1 = m_1, \ldots, l_k = m_k] : T}\ \text{T-Obj}$$

$$\frac{\Gamma, x_1{:}T_1, \ldots, x_k{:}T_k; \Delta, n{:}c \vdash t : T' \quad \Gamma;\Delta \vdash c : T \quad T = [\!(\ldots, l{:}T_1 \times \ldots \times T_k \to T', \ldots)\!]}{\Gamma;\Delta \vdash \varsigma(n{:}c).\lambda(x_1{:}T_1, \ldots, x_k{:}T_k).t : T.l}\ \text{T-Meth}$$

$$\frac{\Gamma;\Delta \vdash v : c \quad \Gamma;\Delta \vdash c : [\!(\ldots, l{:}T_1 \times \ldots \times T_k \to T, \ldots)\!] \quad \Gamma;\Delta \vdash v_1 : T_1 \ \ldots \ \Gamma;\Delta \vdash v_k : T_k}{\Gamma;\Delta \vdash v.l(v_1, \ldots, v_k) : T}\ \text{T-Call}$$

$$\frac{\Gamma;\Delta \vdash v : c \qquad \Gamma;\Delta \vdash c : T \qquad \Gamma;\Delta \vdash v' : T.f}{\Gamma;\Delta \vdash v.f := v' : c}\ \text{T-FUpdate}$$

$$\frac{\Gamma;\Delta \vdash c : [\!(T)\!]}{\Gamma;\Delta \vdash \mathit{new}\ c : c}\ \text{T-NewC} \qquad \frac{\Gamma;\Delta \vdash t : T}{\Gamma;\Delta \vdash \mathit{new}\langle t\rangle : \mathit{thread}}\ \text{T-NewT}$$

$$\frac{}{\Gamma;\Delta \vdash \mathit{currentthread} : \mathit{thread}}\ \text{T-CurrT}$$

$$\frac{\Gamma;\Delta \vdash e : T_1 \qquad \Gamma, x{:}T_1;\Delta \vdash t : T_2}{\Gamma;\Delta \vdash \mathit{let}\ x{:}T_1 = e\ \mathit{in}\ t : T_2}\ \text{T-Let}$$

$$\frac{\Gamma;\Delta \vdash v_1 : T_1 \qquad \Gamma;\Delta \vdash v_2 : T_1 \qquad \Gamma;\Delta \vdash e_1 : T_2 \qquad \Gamma;\Delta \vdash e_2 : T_2}{\Gamma;\Delta \vdash \mathsf{if}\ v_1 = v_2\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : T_2}\ \text{T-Cond}$$

$$\frac{}{\Gamma;\Delta \vdash \mathit{stop} : T}\ \text{T-Stop}$$

$$\frac{\Gamma(x) = T}{\Gamma;\Delta \vdash x : T}\ \text{T-Var} \qquad \frac{\Delta(n) = T}{\Gamma;\Delta \vdash n : T}\ \text{T-Name}$$

$$\frac{}{\Gamma;\Delta \vdash \mathit{block} : T}\ \text{T-Block} \qquad \frac{\Gamma;\Delta \vdash v : T}{\Gamma;\Delta \vdash \mathit{return}[o_1]\ v : T'}\ \text{T-Return}$$

Table 6. Static semantics (2)

$n\langle \mathit{let}\, x{:}T = v\ \mathit{in}\, t\rangle \rightsquigarrow n\langle t[v/x]\rangle$ RED

$n\langle \mathit{let}\, x_2{:}T_2 = (\mathit{let}\, x_1{:}T_1 = e_1\ \mathit{in}\, e)\ \mathit{in}\, t\rangle \rightsquigarrow n\langle \mathit{let}\, x_1{:}T_1 = e_1\ \mathit{in}\ (\mathit{let}\, x_2{:}T_2 = e\ \mathit{in}\, t)\rangle$ LET

$n\langle \mathit{let}\, x{:}T = (\mathsf{if}\ v = v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)\ \mathit{in}\, t\rangle \rightsquigarrow n\langle \mathit{let}\, x{:}T = e_1\ \mathit{in}\, t\rangle$ COND$_1$

$n\langle \mathit{let}\, x{:}T = (\mathsf{if}\ v_1 = v_2\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)\ \mathit{in}\, t\rangle \rightsquigarrow n\langle \mathit{let}\, x{:}T = e_2\ \mathit{in}\, t\rangle$ COND$_2$

$n\langle \mathit{let}\, x{:}T = \mathit{currentthread}\ \mathit{in}\, t\rangle \rightsquigarrow n\langle \mathit{let}\, x{:}T = n\ \mathit{in}\, t\rangle$ CURRENTTHREAD

$c[\![O]\!] \parallel n\langle \mathit{let}\, x{:}c = [o_1]\ \ \mathit{new}\, c\ \mathit{in}\, t\rangle \rightsquigarrow$
$c[\![O]\!] \parallel \nu(o{:}c).(o_1 \hookrightarrow o \parallel o[O] \parallel n\langle \mathit{let}\, x{:}c = o\ \mathit{in}\, t\rangle)$ NEWO$_i$

$n\langle \mathit{let}\, x{:}\, \mathit{thread} = \mathit{new}\langle t\rangle\ \mathit{in}\, t_1\rangle \rightsquigarrow \nu(n_2{:}T).(n\langle \mathit{let}\, x{:}T = n_2\ \mathit{in}\, t_1\rangle \parallel n_2\langle t\rangle)$ NEWT

$n\langle \mathit{let}\, x{:}T = \mathit{stop}\ \mathit{in}\, t\rangle \rightsquigarrow n\langle \mathit{stop}\rangle$ STOP

$o[O] \parallel n\langle \mathit{let}\, x{:}T = o.l(\vec{v})\ \mathit{in}\, t\rangle \xrightarrow{\tau} o[O] \parallel n\langle \mathit{let}\, x{:}T = O.l(o)(\vec{v})\ \mathit{in}\, t\rangle$ CALL$_i$

$o[O] \parallel n\langle \mathit{let}\, x{:}T = o.f := v\ \mathit{in}\, t\rangle \xrightarrow{\tau} o[O.f := v] \parallel n\langle \mathit{let}\, x{:}T = o\ \mathit{in}\, t\rangle$ FUPDATE

Table 7. Internal steps

$\gamma ::= n\langle [o]\mathit{call}\ o.l(\vec{v})\rangle \mid n\langle \mathit{return}(v)\rangle \mid \nu(n{:}T).\gamma$	basic labels
$a ::= \gamma? \mid \gamma!$	receive and send labels

Table 8. Labels

$$\frac{\begin{array}{c} ;\Delta \vdash o_1 : [\ldots] \qquad ;\Delta,\Theta \vdash o_2.l(\vec{v}) : T \qquad o_2 \in \Theta \qquad \Delta \vdash n : \mathit{thread} \qquad n \notin \mathit{dom}(\Theta) \\ \Delta; E_\Delta \vdash [o_1] \leftrightharpoons\hookrightarrow \vec{v} : \Theta \qquad \Delta; E_\Delta \vdash [o_1] \leftrightharpoons\hookrightarrow o_2 : \Theta \qquad \Delta; E_\Delta \vdash n \leftrightharpoons [o_1] : \Theta \\ \acute{E}_\Delta = E_\Delta \setminus n \qquad \acute{\Theta} = \Theta, n\colon \mathit{thread} \qquad \acute{E}_\Theta = E_\Theta + (o_2 \hookrightarrow \vec{v}, n \hookrightarrow o_2) \end{array}}{\begin{array}{c} \Delta; E_\Delta \vdash C : \Theta; E_\Theta \xrightarrow{n\langle [o_1]\mathit{call}\ o_2.l(\vec{v})\rangle ?} \\ \Delta; \acute{E}_\Delta \vdash C \parallel n\langle \mathit{let}\, x{:}T = o_2.l(\vec{v})\ \mathit{in}\ \mathit{return}[o_1]\ x\rangle : \acute{\Theta}; \acute{E}_\Theta \end{array}} \ \textsc{CalliI}_1$$

$$\frac{\begin{array}{c} ;\Delta \vdash o_1 : [\ldots] \qquad ;\Delta,\Theta \vdash o_2.l(\vec{v}) : T \qquad o_2 \in \Theta \qquad \Delta \vdash n : \mathit{thread} \\ E_\Delta \vdash [o_1] \leftrightharpoons\hookrightarrow \vec{v} : \Theta \qquad E_\Delta \vdash [o_1] \leftrightharpoons\hookrightarrow o_2 : \Theta \qquad E_\Delta \vdash n \leftrightharpoons [o_1] : \Theta \\ \acute{E}_\Delta = E_\Delta \setminus n \qquad \acute{E}_\Theta = E_\Theta + (o_2 \hookrightarrow \vec{v}, n \hookrightarrow o_2) \end{array}}{\begin{array}{c} \Delta; E_\Delta \vdash C \parallel n\langle \mathit{let}\, x'{:}T' = [o_2]\ \mathit{blocks\ for}\ o_2'\ \mathit{in}\ t\rangle : \Theta; E_\Theta \xrightarrow{n\langle [o_1]\mathit{call}\ o_2.l(\vec{v})\rangle ?} \\ \Delta; \acute{E}_\Delta \vdash C \parallel n\langle \mathit{let}\, x{:}T = o_2.l(\vec{v})\ \mathit{in}\ \mathit{return}[o_1]\ x; \mathit{let}\, x'{:}T' = [o_2]\ \mathit{blocks\ for}\ o_2'\ \mathit{in}\ t\rangle : \Theta; \acute{E}_\Theta \end{array}} \ \textsc{CalliI}_2$$

$$\frac{\acute{E}_\Delta = E_\Delta + ([o_1] \hookrightarrow v, n \hookrightarrow [o_1]) \qquad \acute{E}_\Theta = E_\Theta \setminus n}{\Delta; E_\Delta \vdash C \parallel n\langle \mathit{let}\, x{:}T = \mathit{return}[o_1]\ v\ \mathit{in}\ t\rangle : \Theta; E_\Theta \xrightarrow{n\langle \mathit{return}(v)\rangle !} \Delta; \acute{E}_\Delta \vdash C \parallel n\langle t\rangle : \Theta; \acute{E}_\Theta} \ \textsc{RetO}$$

$$\frac{o_2 \in \Delta \qquad \acute{E}_\Delta = E_\Delta + (o_2 \hookrightarrow \vec{v}, n \hookrightarrow o_2) \qquad \acute{E}_\Theta = E_\Theta \setminus n}{\begin{array}{c} \Delta; E_\Delta \vdash C \parallel n\langle \mathit{let}\, x{:}T = [o_1]\ o_2.l(\vec{v})\ \mathit{in}\ t\rangle : \Theta; E_\Theta \xrightarrow{n\langle [o_1]\mathit{call}\ o_2.l(\vec{v})\rangle !} \\ \Delta; \acute{E}_\Delta; \vdash C \parallel n\langle \mathit{let}\, x{:}T = [o_1]\ \mathit{blocks\ for}\ o_2\ \mathit{in}\ t\rangle : \Theta; E_\Theta \end{array}} \ \textsc{CallO}$$

$$\frac{\begin{array}{c} ;\Delta,\Theta \vdash v : T \qquad \Delta; E_\Delta \vdash o_2 \leftrightharpoons\hookrightarrow v : \Theta \qquad \Delta; E_\Delta \vdash n \hookrightarrow o_2 : \Theta \\ \acute{E}_\Delta = E_\Delta \setminus n \qquad \acute{E}_\Theta = E_\Theta + (o_1 \hookrightarrow v, n \hookrightarrow [o_1]) \end{array}}{\begin{array}{c} \Delta; E_\Delta \vdash C \parallel n\langle \mathit{let}\, x{:}T = [o_1]\ \mathit{blocks\ for}\ o_2\ \mathit{in}\ t\rangle : \Theta; E_\Theta \xrightarrow{n\langle \mathit{return}(v)\rangle ?} \\ \Delta; \acute{E}_\Delta \vdash C \parallel n\langle t[v/x]\rangle : \Theta; \acute{E}_\Theta \end{array}} \ \textsc{RetI}$$

Table 9. External steps

$$\frac{n \notin fn(a) \qquad \Delta; E_\Delta \vdash C : \Theta, n{:}T \xrightarrow{a} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}, n{:}T; \acute{E}_\Theta}{\Delta; E_\Delta \vdash \nu(n{:}T).C : \Theta \xrightarrow{a} \Delta'; E'_\Delta \vdash \nu(n{:}T).C' : \acute{\Theta}; \acute{E}_\Theta \setminus n}\ \textsc{Comm}$$

$$\frac{\begin{array}{c} n \in fn(\gamma) \qquad \Delta' = \Delta, n{:}T \quad E'_\Delta = E_\Delta + \tilde{E}_\Delta \quad \Delta; E_\Delta \vdash \Delta'; E'_\Delta \downarrow_{\Delta\times(\Delta+\Theta)} : \Theta \\ \Delta'; E'_\Delta \vdash C : \Theta \xrightarrow{\gamma?} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta \end{array}}{\Delta; E_\Delta \vdash C : \Theta \xrightarrow{\nu(n{:}T;\tilde{E}_\Delta).\gamma?} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta}\ \textsc{BIn}$$

$$\frac{\begin{array}{c} n \in fn(\gamma) \qquad \Delta; E_\Delta \not\vdash T : [\![\ldots]\!] \\ \Delta; E_\Delta \vdash (C \setminus n) \ : \Theta, n{:}T; E_\Theta + E_\Theta(C, n) \xrightarrow{\gamma!} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta \end{array}}{\Delta; E_\Delta \vdash \nu(n{:}T).C : \Theta; E_\Theta \xrightarrow{\nu(n{:}T;E_\Theta^{\leftrightarrows\hookrightarrow}(C,n)).\gamma!} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta}\ \textsc{BOut}$$

$$\frac{\begin{array}{c} o \in fn(\gamma) \qquad \Theta \vdash c : [\![\ldots]\!] \qquad C(c) = [\![O]\!] \\ \Theta' = \Theta, o{:}c \qquad E'_\Delta = E_\Delta + \tilde{E}_\Delta \qquad \Delta; E_\Delta \vdash \Delta; E'_\Delta \downarrow_{\Delta\times(\Delta+\Theta)} : \Theta \\ \Delta; E'_\Delta \vdash C \parallel o[O] : \Theta'; E_\Theta \xrightarrow{\gamma?} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta \end{array}}{\Delta; E_\Delta \vdash C : \Theta \xrightarrow{\nu(o{:}c;\tilde{E}_\Delta).\gamma?} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta}\ \textsc{BIn}_{new}$$

$$\frac{\begin{array}{c} o \in fn(\gamma) \qquad \Delta \vdash c : [\![\ldots]\!] \\ \Delta, o{:}c; E_\Delta \vdash C : \Theta; E_\Theta + E_\Theta^{\leftrightarrows\hookrightarrow}(C, n) \xrightarrow{\gamma!} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta \end{array}}{\Delta; E_\Delta \vdash \nu(o : c).C : \Theta; E_\Theta \xrightarrow{\nu(o{:}c;E_\Theta^{\leftrightarrows\hookrightarrow}(C,n)).\gamma!} \acute{\Delta}; \acute{E}_\Delta \vdash \acute{C} : \acute{\Theta}; \acute{E}_\Theta}\ \textsc{BOut}_{new}$$

Table 10. External steps (scoping)