UNIVERSITY OF OSLO
Department of Informatics

Observable interface behavior and inheritance

April 2011

Research Report No. 409

Erika Ábrahám, Thi Mai Thuong Tran, and Martin Steffen

ISBN 82-7368-368-0
ISSN 806-3036

April 2011

Observable interface behavior and inheritance*

Erika Ábrahám[1] and Thi Mai Thuong Tran[2] and Martin Steffen[2]

[1] RWTH Aachen University, Germany
[2] University of Oslo, Norway

1 Introduction

An *open* system is a part of a larger system, which interacts with its environment, and best considered as a black box where the internals are hidden. Such a separation of internal behavior from externally relevant interface behavior is crucial for compositionality. The most popular programming paradigm nowadays is object orientation, which in particular supports interfaces and encapsulation of objects. Another crucial feature in mainstream object orientation is *inheritance,* which allows code reuse and is intended to support incremental program development by gradually extending and specializing an existing class hierarchy.

Openness of a system in the presence of inheritance and late binding is problematic. One symptom of that is known in software engineering as the fragile base class problem [14,20,19]. A base class in an inheritance hierarchy is a (common) super-class, and fragile means that replacing one base class by another, seemingly satisfying the same interface description, may break the code of the client of the base class, i.e., change the behavior of the "environment" of the base class. Consider the following code fragment.

Listing 1.1. Fragile base class

```
class A  {
  void add () {...}
  void add2 () {...}
  ...
}
```

```
class B extends A {
  void add () {
    size = size + 1;
   super.add(); }
   void add2 () {
     size = size + 2;
     super.add2();}
```

The two methods *add* and add_2 are intended to add one respectively two elements to some container data structure. This completely (albeit informally) describes the intended behavior of A's methods. Class B in addition keeps information about the size of the container. Due to late-binding, this implementation of B is wrong if the add_2-method of the super-class A is implemented via *self*-calls using two times the *add*-method. The problem is that *nothing* in the interface, e.g., in the form of a behavioral specification using pre- and post-conditions of the methods, helps to avoid the problem. The interface specification is too weak to allow to consider the base class as a black box which can be safely substituted based on its interface specification only.

* The work has been partly supported by the EU-project and FP7-231620 HATS (Highly Adaptable and Trustworthy Software using Formal Methods).

This paper formally characterizes the interface behavior for open systems with inheritance. From an observational point of view, the only thing that counts is the interaction with the environment (or observer) and whether this interaction leads to observable reactions in the environment. Thus, a rigorous account of such an interface behavior is the key to formal verification of open programs as well as a formal foundation for black-box testing. If done properly, it ultimately allows compositional reasoning, i.e., to infer properties of a composed system from the interface properties of its sub-constituents without referring to further internal representation details. A representation-independent, abstract account of the behavior is also necessary for compositional optimization of components: only when showing the same external behavior one program can replace another without changing the interaction with any client code.

In a message-passing setting, the component behavior consists of message traces, i.e., sequences of component-environment interactions. Writing $C \stackrel{t}{\Longrightarrow} \acute{C}$, the t denotes the *trace* of interface actions by which C evolves into $\acute{C}$, potentially executing internal steps, as well, not recorded in t. An open program C, however, does not act in isolation, but interacts with *some* environment. I.e., we are interested in traces t where *there exists an environment* E such that $C \parallel E \underset{\bar{t}}{\stackrel{t}{\Longrightarrow}} \acute{C} \parallel \acute{E}$ by which we mean: component C produces the trace t and E produces the dual trace $\bar{t}$, both together "canceling out" to internal steps. In other words, our goal is to formulate the external or open semantics with the environment *existentially abstracted away.* With infinitely many possible environments E, the challenge is to capture what is common to *all* those environments. This will be done in form of *assumptions* about the environment. This means, the operational semantics specifies the behavior of C under certain assumptions Ξ_E about the environment. Following standard notation from logics, we do not write $\Xi_E \parallel C$, but rather $\Xi_E \vdash C$, such that the reductions will look like[3]

$$\Xi_E \vdash C \stackrel{t}{\Longrightarrow} \acute{\Xi}_E \vdash \acute{C} \ . \tag{1}$$

Such a characterization of the abstract interface behavior is relevant and useful for the following reasons. Firstly: the set of traces according to equation (1) is in general more restricted than the one obtained when ignoring the environments altogether. This means, when *reasoning* about the behavior of C based on the traces, e.g., for the purpose of verification, the more precise knowledge of the possible traces allows to carry out stronger arguments about C. Secondly, an application for a trace description is black-box testing, in that one describes the behavior of a component in terms of the interface traces and then synthesize appropriate test drivers from it. Obviously it makes no sense to specify interface behavior which is not possible, at all, since in this case one could not generate a corresponding tester. Finally, and not as the least gain, the formulation gives

[3] To avoid later confusion: The Ξ_E as used in the semantics later does not only formalize assumptions about the environment, but also *commitments* of the component, to make the setting symmetric. Also, the notation Ξ_E will not be used later, it is used only here for explanatory reasons.

insight into the inherent semantical nature of the language, as the assumptions Ξ and the semantics captures the existentially abstracted environment behavior.

This paper formalizes an open semantics for a statically typed object-oriented calculus featuring concurrency, dynamic object creation, mutable heap, and single *inheritance.* Based on the ideas sketched above, the interface behavior is phrased in an assumption-commitment framework. In particular, the assumption and commitment contexts need to capture an abstract over-approximation of the heap structure ("connectivity", cf. Section 2 for an informal explanation). Furthermore, we formalize what constitutes allowed interface behavior in general, i.e., abstracting not only away from the environment, but describing the possible interface behavior for arbitrary programs and environments. We prove the soundness of the abstractions. The results here extend previous work by considering the crucial feature of inheritance. Earlier we considered the problem of open systems for different choices of language features (but without inheritance), for instance for futures and promises [4], and for Java-like monitors [5]. Object-connectivity already played a role as a consequence of cross-border instantiation [3][2] but not inheritance (see also [21]).

The paper is organized as follows. We start in Section 2 by explaining the approach of this paper in more detail, by way of examples. Section 3 presents syntax, type system, and (open) semantics of the calculus. Section 4 formalizes allowed interface behavior in general and provides the soundness of the abstractions. We conclude in Section 5 by discussing related and future work.

2 Interface behavior, inheritance, and object connectivity

With sets of classes as units of composition, we start by discussing informally what can be observed from outside a "component" when considering *inheritance.* Even when we restrict ourselves to run-of-the-mill notion of single-inheritance between classes with subtype polymorphism, late-binding, and method overriding, a number of design issues influence what can be observed from the outside given a set of classes. We discuss some of the issues using some object-oriented pseudo-code. The interface behavior of an open system will be given in terms of traces of interactions exchanged between the component and the "environment". We allow that classes of the component extend those from the environment via inheritance, and vice versa. An interface interaction happens if a step of the component affects the environment, resp. vice versa. Objects encapsulate their states, and thus the interaction takes the form of messages exchanged between component and environment (method calls and returns), where the control changes from executing component code to environment code (outgoing message) or vice versa (incoming message).[4]

Assume two classes, C_C as a component class implementing a method m_C, and C_E in the environment providing a method m_E, and assume that m_C calls m_E on an instance of C_E. Figure 1(a) illustrates the situation where an instance

[4] If the language allowed shared variables, an interface interaction not necessarily mean that the *control* changes the side.

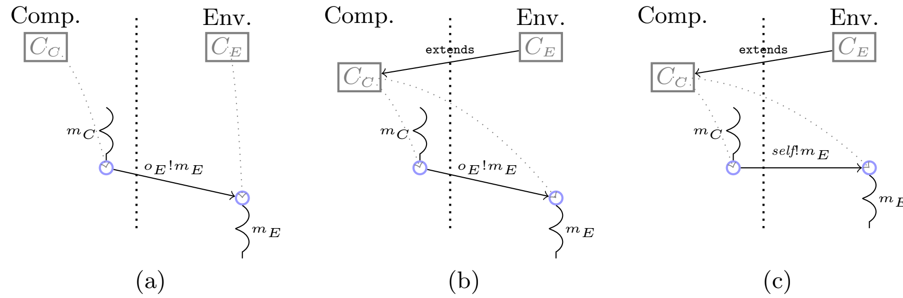

Fig. 1. Calls across the interface

o_C of the component class executes m_C and calls the method m_E on an instance o_E of the environment class, represented by the call $o_E!m_E$ which crosses the interface between the component and the environment.

The picture does not change much when C_C extends C_E, even if both caller and callee are instances of the component class C_C, i.e., if the callee inherits m_E from C_E (cf. Figure 1(b)). See also Listing 1.2. Especially, if the caller and the callee are the same object, i.e., if m_C calls the (inherited) m_E via a self-call, it constitutes an interface interaction, because the code of m_E is specified by the environment (Figure 1(c)).

Listing 1.2. Late binding

```
class C1 {                                  class C2 extends C1 {
  ..                                            ...
  public void m1 () {...} ..                    public void m2 () {...x.m1...}
}                                             }
```

Likewise in the inverse situation in Listing 1.3, which illustrates late-binding and overriding: the self-call in method m_1 is a component-internal call when executed in an instance of C_C, but an interface call when m_1 is an (inherited) method of an instance of C_E. The situation is analogous to the code that illustrated the fragile base class problem.

Listing 1.3. Overriding

```
class CC {                                  class CE extends CC {
  ...                                           ...
  public void m1 () {... self.m2 ...}           public void m2 () {...}
  public void m2 () {...}                       ...
}                                             }
```

Dynamic type and overriding As in Java, we assume that classes, besides being generators of objects, play the role of types as well, and that inheritance implies subtyping. The type system is thus nominal and supports nominal subtyping. A question is, whether in the presence of subtyping, the *dynamic* type of an object is observable. More concretely, assuming two classes c_E and c_C, with c_C

a subclass of c_E, does it make a difference to have an instance of c_E or of c_C? Consider the following two expressions:

$$\mathsf{let}\, x{:}c_E = \mathsf{new}\, \mathbf{c_E}\, \mathsf{in}\, t \quad \text{and} \quad \mathsf{let}\, x{:}c_E = \mathsf{new}\, \mathbf{c_C}\, \mathsf{in}\, t \tag{2}$$

In the first case, the dynamic type of the instance is c_E, in the second case it's the subclass/subtype c_C. Can one distinguish the two situations? If the superclass c_E is a component class and c_C is an observer class, the two situations of equation (2) are distinguishable: by *overriding* a method of c_E in c_C, the behavior of instances of c_E differs from instances of c_C. For instance, as shown more concretely in Java-code in Listing 1.4 (which shows the situation that an instance of the sub-class is created).[5]

Listing 1.4. Dynamic type

```
public class Dynamictypeobs1 {
    public static void main(String[] args){
        C1 c = new C2();
        c.m();
    }
}

class C1 {
    void m () {System.out.print("C1");}
}

class C2 extends C1 {
    void m () {System.out.print("C2");};

}
```

Also in the inverse situation that c_E is an observer class and c_C a class of the environment, the two situations of equation (2) are distinguishable.

Remark 1 (Dynamic type and overriding). An important question when formulating the open semantics is what is observable from the outside, as that determines what should be recorded at the interface. One piece of information included in the interface is the inheritance hierarchy, i.e., which (public) class extends which one. That information is a priori necessary, as the language is typed and supports subtype polymorphism (which is connected to inheritance in that inheritance implies subtyping). Since open programs are required to be well-typed *statically*, the information about inheritance/subtyping needs to be available at compile time, i.e., it is included in the static interface description between environment and component. □

Objects encapsulate their instance states, such that fields of an object cannot be accessed from outside the instance.[6]In particular, each method can access only

[5] Since the observer class `Dynamictypeobs1` literally mentions `new C1()` resp. `new C2()`, one could argue that just by that fact it can see a difference. The point, however, is the change in behavior, and this would also be observable if the observer would not itself create the instance with static type `C1`, but it would be handed over to the environment, for instance as return value of a method call.

[6] This is slightly stronger than the restriction for `private` fields in Java, which allow access among instances of the same class.

the fields of the class that the method is *defined* in. In the presence of inheritance, each object may contain fields defined by the component and fields defined by the environment. Due to the mentioned privacy restriction, component fields are manipulated only by component methods, and dually for environment fields. If the component instantiates a new object, fields from the component class C_C belong to the component part of the heap and fields from C_E to the environment part (cf. Figure 2(a), where the environment part, coming from the abstract environment, is grayed out).

In Figure 2(b), the component creates *two* instances of C_C, say o_1 and o_2. Directly after creation, the fields of o_1 and o_2 are undefined (in absence of constructors) and in particular, o_1 and o_2 are surely unconnected (i.e., their fields do not refer to each other).

The creator of the two objects on the component-side could call a set-method on o_1 with parameter o_2 to set one of the fields of o_1 to point to o_2. If the set method is defined in the *component* class C_C, then it may access only fields defined in C_C. Thus the call is internal and *not visible* at the interface, as indicated in Figure 2(c). However, if the set-method is inherited from C_E, then the call executes a method specified by the *environment* and modifies fields in the environment part of o_1. Therefore, in this case the call is a *visible* interface interaction (Figure 2(d)).

In general, the *environment* part of the objects *created by the component* is *unconnected* unless brought in connection by (outgoing) communication, i.e., method calls and returns, sending some object identities as parameter or return values across the border. These values can be stored in environment fields and can be used to execute calls, thereby exchanging information. In the above example, if the set-method is defined in the environment, then after the creator calls the set-method of o_1 with parameter o_2, the environment part of o_1 has a reference to o_2 (Figure 2(d)). Now o_1 may call a method of o_2 and pass on its own identity as a parameter, such that o_1 and o_2 both "know" about each other, i.e., they are fully *connected*.

The situation concerning component and environment is completely symmetric. The *environment* part of objects *created by the environment* can be connected among each other without being observable at the interface. The environment may connect the *component* part of those objects via (from the component view) incoming communication.

After these observations, let us come back short to the fragile class problem described in the introduction. The following sufficient requirement assures that the discussed unexpected behaviour of classes extending A does not happen: Considering A as a component class, there is *no environment* in which an execution of the method *add2* specified in A leads to a *visible interface* interaction (i.e., only its call and return are visible).

To describe the possible interface behavior, where all possible environments are represented abstractly by assumption contexts, the potential connectivity of the environment is important. E.g., an incoming call of the form $o_1.m(o_2)?$ is impossible if, judging from the earlier interaction history, o_1 and o_2 cannot be

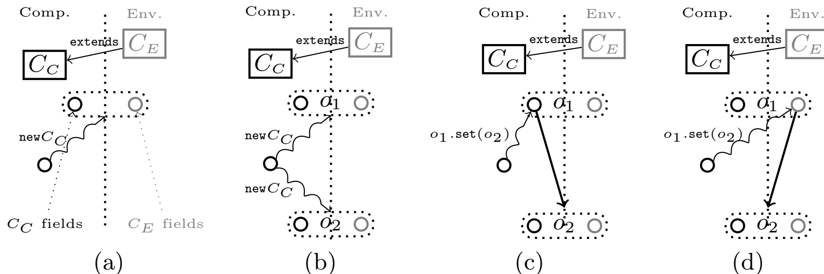

Fig. 2. Heap structure and connectivity

in connection in the environment (i.e., the environment parts of o_1 and o_2 do not have any references to each other). Besides checking that incoming communication is consistent with the connectivity assumptions, the values communicated over the interface *update* those connectivity assumptions, e.g., an outgoing communication $o_1.m(o_2)!$ adds the knowledge to the assumption that after the step, (the environment part of) o_1 may now be in connection with o_2. As via environment-internal communication, o_1 *may* communicate with o_2 and with all other objects it may know, the assumed connectivity is taken as a reflexive, transitive, and symmetric relation, i.e., an equivalence relation. We call the equivalence classes of objects that may be connected with each other *cliques* of objects. The operational semantics in Section 3.3.2 formalizes these intuitions.

3 Calculus

This section presents the calculus, its syntax and operational semantics, capturing the core of the *Creol*-language [8][4].

It is a concurrent variant of an imperative, object-calculus in the style of the calculi from [1]. Unlike in [8][4], we omit the treatment of first-class futures, which can be seen as a generalization of asynchronous method calls, to simplify the presentation. We start with the abstract syntax in Section 3.1. Afterwards we present the type system and the operational semantics in Sections 3.2 and 3.3.

3.1 Syntax

The abstract syntax is given in Table 1. It distinguishes between *user* syntax and *run-time* syntax (the latter underlined). The user syntax contains the phrases in which programs are written; the run-time syntax contains syntactic constituents additionally needed to express the behavior of the executing program in the operational semantics.

The basic syntactic category of names n, which count among the values v, represents references to classes, to objects, and to threads. To facilitate reading,

$C ::= \mathbf{0} \mid C \parallel C \mid \underline{\nu(n{:}T).C} \mid n[\![O]\!] \mid \underline{n[O, L]} \mid \underline{n\langle t\rangle}$	component
$O ::= n, M, F$	object
$M ::= l = m, \ldots, l = m$	method suite
$F ::= l = f, \ldots, l = f$	fields
$m ::= \varsigma(n{:}T).\lambda(x{:}T, \ldots, x{:}T).t$	method
$f ::= v \mid \bot_{n'}$	field
$t ::= v \mid \mathsf{stop} \mid \mathsf{let}\, x{:}T = e \,\mathsf{in}\, t$	thread
$e ::= t \mid \mathsf{if}\ v = v\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \mathsf{if}\ undef(v.l())\ \mathsf{then}\ e\ \mathsf{else}\ e$	expr.
$\quad \mid\ n@l(\vec{v}) \mid v.l() \mid v.l() := v$	
$\quad \mid\ \mathsf{new}\, n \mid \mathsf{claim}@(n, n) \mid \underline{\mathsf{get}@n} \mid \mathsf{suspend}(n) \mid \underline{\mathsf{grab}(n)} \mid \underline{\mathsf{release}(n)}$	
$v ::= x \mid n \mid ()$	values
$L ::= \bot \mid \top$	lock status

Table 1. Abstract syntax

we allow ourselves to write o and its syntactic variants for names referring to objects, c for classes, and p for threads ("processes"). Technically, the disambiguation between the different roles of the names is done by the type system and the abstract syntax of Table 1 uses the non-specific n for names. The unit value is represented by $()$ and x stands for variables, i.e., local variables and formal parameters, but not instance variables.

A *component* C is a collection of classes, objects, and (named) threads, with $\mathbf{0}$ representing the empty component. The sub-entities of a component are composed using the parallel-construct $\parallel$. The entities executing in parallel are the named threads $p\langle t\rangle$, where t is the code being executed and p the name of the thread. A class $c[\![c', M, F]\!]$ carries a name c, it references its immediate superclass c' and defines its methods and fields in M and F. An object $o[c, M, F, L]$ with identity o keeps a reference to the class c it instantiates, contains the embedded methods from its class, stores the current value F of its fields, and maintains a *binary lock* L indicating whether any code is currently active inside the object (in which case the lock is taken indicated by $\top$) or not (in which case the lock is free indicated by $\bot$). From the three kinds of entities at component level — threads $p\langle t\rangle$, classes $c[\![c', M, F]\!]$, and objects $o[c, M, F, L]$— only the threads are *active*, executing entities, being the target of the reduction rules. The objects, in contrast, store the embedded methods implemented by their classes and the state in their fields or instance variables, whereas the classes are constant entities specifying the methods.

The named threads $p\langle t\rangle$ are incarnations of method bodies "in execution". Incarnations insofar, as the formal parameters have been replaced by actual ones, especially the method's self-parameter has been replaced by the identity of the target object of the method call. The term t is basically a sequence of expressions, where the let-construct is used for sequencing and for local declarations.[7] During

[7] $t_1; t_2$ (sequential composition) abbreviates $\mathsf{let}\, x{:}T = t_1 \,\mathsf{in}\, t_2$, where x does not occur free in t_2.

execution, $p\langle t\rangle$ contains in t the currently running code of a method body. When evaluated, the thread is of the form $p\langle v\rangle$ and the value can be accessed via p, the future reference, or future for short.

Each thread belongs to one specific object "inside" which it executes, i.e., whose instance variables it has access to. Object locks are used to rule out unprotected concurrent access to the object states: Though each object may have more than one method body incarnation partially evaluated, at each time point at most one of those bodies (the lock owner) can be active inside the object. In the terminology of Java, all methods are implicitly considered "synchronized". The final construct at the component level is the ν-operator for hiding and dynamic scoping, as known from the π-calculus. In a component $C = \nu(n{:}T).C'$, the scope of the name n (of type T) is restricted to C' and unknown outside C. ν-binders are introduced when dynamically creating new named entities, i.e., when instantiating new objects or new threads. The scope is dynamic, i.e., when the name is communicated by message passing, it is enlarged.

Besides components, the grammar specifies the lower level syntactic constructs, in particular, methods, expressions, and (unnamed) threads, which are basically sequences of expressions. A method $\varsigma(s{:}T).\lambda(\vec{x}{:}\vec{T}).t$ provides the method body t abstracted over the ς-bound "self" parameter s and the formal parameters $\vec{x}$. For fields, they are either a value or yet undefined. In freshly created objects, the lock is free, and all fields carry undefined references $\bot_c$, where class name c is the type of the field.

We use f for instance variables or fields and $l = v$, resp. $l = \bot_c$ for field variable definition. Field access is written as $v.l()$ and field update as $v'.l() := v$. Note that the construct $v.l()$ is used for field access only, but not for method invocation. We will use $v_\bot$ to denote either a value v or a symbol $\bot_c$ for being undefined. Note that the syntax does not allow to set a field back to undefined. Direct access (read or write) to fields across object boundaries is forbidden by convention. In connection with inheritance, there are two further restrictions we assume for the field access: A method defined in a subclass is not allowed to directly access fields that are defined in the super-class, neither by using the keyword `super` (which we omitted anyhow), nor by accessing the variable via self, when the field is inherited. In *Java*, that would correspond to *private* fields, as they cannot be accessed by subclasses. These design choices will have quite some impact on what is observable at the interface. Intuitively, the more liberal the language is wrt. field access, the more details about instances become observable. Instantiation of a new object from class c is denoted by $\mathsf{new}\, c$.

Method calls are written $o@l(\vec{v})$, where the call to l with callee o is sent *asynchronously* and not, as in for instance in *Java*, synchronously. The further expressions claim, get, suspend, grab, and release deal with communication and synchronization. As mentioned, objects come equipped with binary locks, responsible for mutual exclusion. The two basic, complementary operations on a lock are grab and release. The first allows an activity to acquire access in case the lock is free ($\bot$), thereby setting it to $\top$, and $\mathsf{release}(o)$ conversely relinquishes the lock of the object o, giving other threads the chance to be executed in its stead.

The user is not allowed to directly manipulate the object locks. Thus, both expressions belong to the run-time syntax. Instead of using directly grab and release, the lock-handling is done automatically when executing a method body: before starting to execute the method, the lock has to be acquired and upon termination, the lock is released again. Besides that, lock-handling is involved also when futures are claimed, i.e., when a client code executing in an object, say o, intends to read the result of a future. The expression claim@(p, o) is the attempt to obtain the result of a method call from the future p while in possession of the lock of object o. There are two possibilities in that situation: either the value of the future has already been determined, i.e., the method calculating the result has terminated, in which case the client just obtains the value *without* loosing its own lock. In the alternative case, where the value is not yet determined, the client trying to read the value gives up its lock via release and continues executing only after the requested value has been determined (using get to read it) and after it has re-acquired the lock. Unlike claim, the get-operation is not part of the user-syntax. Both expressions are used to read back the value from a future, the difference in behavior is that get unconditionally attempts to get the value, i.e., blocks until the value has arrived, whereas claim gives up the lock temporarily, if the value has not yet arrived, as explained. Finally, executing suspend(o) causes the activity to relinquish and re-grab the lock of the object o. We assume by convention that when appearing in methods of classes, the claim- and the suspend-commands only refer to the self-parameter *self*, i.e., they are written claim@(p, self) and suspend(self).

3.2 Type system

The language is typed and the available types are given in the following grammar:

$$
\begin{array}{ll}
T ::= B \mid \mathsf{Unit} \mid \langle T \rangle \mid [S] \mid [\![S]\!] \mid n & \text{types} \\
S ::= l{:}U, \ldots, (l){:}U, \ldots, l{:}T & \text{signatures} \\
U ::= T \times \ldots \times T \rightarrow T & \text{member types}
\end{array}
$$

Besides base types B (left unspecified; typical examples are booleans, integers, etc.), Unit is the type of the unit value (). Type $\langle T \rangle$ represents a reference to a future which will return a value of type T, in case it eventually terminates. The name of a class serves as the type for its instances. We need as auxiliary type constructions (i.e., not as part of the user syntax, but to formulate the type system) the type or interface of unnamed objects, written $[S]$ and the interface type for classes, written $[\![S]\!]$ where S is the *signature*. The signature contain the labels l of the available members together with the expected types. Furthermore, we distinguish whether a member labelled l is actually implemented by the class (in which case we write $l{:}U$), or whether it is provided, but *inherited* from a super-class (in which case we write $(l){:}U$). Fields, also labelled by labels l, are of types T. We allow ourselves to write $\vec{T}$ for $T_1 \times \ldots \times T_k$ etc. where we assume that the number of arguments match in the rules, and write $\mathsf{Unit} \rightarrow T$ for $T_1 \times \ldots \times T_k \rightarrow T$ when $k = 0$.

We are interested in the behavior of *well-typed* programs, only, and the section presents the type system to characterize those. As the operational rules later, the derivation rules for typing are grouped into two sets: one for typing at the level of components, i.e., global configurations, and secondly one for their syntactic sub-constituents.

Table 2 defines the typing on the level of *global* configurations, i.e., for "sets" of objects, classes, and named threads. On that level, the typing judgments are of the form

$$\Delta \vdash C : \Theta \ , \tag{3}$$

where Δ and Θ are *name contexts*, i.e., finite mappings from names (of classes, objects, and threads) to types. In the judgment, Δ plays the role of the typing assumptions about the *environment,* and Θ of the commitments of the *component,* i.e., the names offered to the environment. Sometimes, the words required and provided interface are used to describe their dual roles. Δ must contain at least all external names referenced by C and dually Θ mentions the names offered by C.

The empty configuration $\mathbf{0}$ is well-typed in any context and exports no names (cf. rule T-Empty). Two configurations in parallel can refer mutually to each other's commitments and together offer the (disjoint) union of their names (cf. rule T-Par). It will be an invariant of the operational semantics that the identities of parallel entities are disjoint wrt. the mentioned names.[8] Therefore, Θ_1 and Θ_2 in the rule for parallel composition are merged disjointly, indicated by writing Θ_1, Θ_2 (analogously for the assumption contexts). In general, C_1 and C_2 can rely on the same assumptions that also $C_1 \parallel C_2$ in the conclusion uses, as it represents the environment *common* to $C_1 \parallel C_2$.

$$\frac{}{\Delta \vdash \mathbf{0} : ()}\ \text{T-Empty} \qquad \frac{\Delta_1, \Theta_2 \vdash C_1 : \Theta_1 \quad \Delta_2, \Theta_1 \vdash C_2 : \Theta_2}{\Delta_1, \Delta_2 \vdash C_1 \parallel C_2 : \Theta_1, \Theta_2}\ \text{T-Par} \qquad \frac{\Delta \vdash C : \Theta, n{:}T}{\Delta \vdash \nu(n{:}T).C : \Theta}\ \text{T-Nu}$$

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

$$\frac{\bullet; \Delta, p{:}\langle T\rangle \vdash t : T}{\Delta \vdash p\langle t\rangle : (p{:}\langle T\rangle)}\ \text{T-NThread} \qquad \frac{\Delta' \leq \Delta \quad \Theta \leq \Theta' \quad \Delta \vdash C : \Theta}{\Delta' \vdash C : \Theta'}\ \text{T-Sub}$$

Table 2. Typing (component level)

The ν-binder hides object names and future/thread names inside the component (cf. rule T-Nu). In the T-Nu-rule, we assume that the bound name n is

[8] In the open semantics later, the Δ and the Θ contexts will *not* be disjoint wrt. object names.

new to Δ and Θ. Object names created by new and thread/future names created by asynchronous method calls are *heap* allocated and thus checked in a "parallel" context (cf. again the assumption-commitment rule T-PAR). The rule for named classes introduce the name of the class and its type into the commitment (cf. T-NCLASS). The code $[\![O]\!]$ of the class $c[\![O]\!]$ is checked in an assumption context where the name of the class is available. Note also that the premise of T-NCLASS (like those of T-NOBJ and T-NTHREAD) is not covered by the rules for type checking at the component level, but by the rules for the lower level entities (in this particular case, by rule T-OBJ from Table 3). The judgments there use as assumption not just a name context, but additionally a stack-organized context Γ in order to handle the let-bound variables. So in general, the assumption context at that level is of the form $\Gamma; \Delta$. The premise of T-NCLASS starts, however, with Γ being empty, i.e., with no assumptions about the type of local variables. This is written in the premise as $\bullet; \Delta, c{:}T \vdash [\![O]\!] : T$; similar for the premises of T-NOBJ and T-NTHREAD. An instantiated object will be available in the exported context Θ by rule T-NOBJ. Threads $p\langle t\rangle$ are treated by rule T-NTHREAD, where the type $\langle T\rangle$ of the future reference p is matched against the result type T of thread t. The last rule is a rule of subsumption, expressing a simple form of subtyping: we allow that an object respectively a class contains *at least* the members which are required by the interface. This corresponds to width subtyping.

Next we formalize the typing for objects and threads and their syntactic sub-constituents. The judgments are of the form

$$\Gamma; \Delta \vdash e : T \tag{4}$$

(and analogously m, $[\![O]\!]$, etc. instead of e). The typing is given in Tables 3 and 4. Besides assumptions about the provided names of the environment kept in Δ, the typing is done relative to assumptions about occurring free variables. They are kept separately in a variable context Γ, a finite mapping from variables to types.

Rule T-CLASS type-checks classes $[\![c_2, \vec{l}_f = \vec{f}, \vec{l} = \vec{m}]\!]$, "called" in the premise of rule T-NCLASS from Table 2 for named classes on the global level, where c_1 in the conclusion of T-CLASS is the class/type of $[\![c_2, \vec{l}_f = \vec{f}, \vec{l} = \vec{m}]\!]$ and c_2 its direct super-class. The name of the class c_1 is used in the first premise to determine its interface type, which lists the types of the class members. For the methods, $\vec{l}{:}\vec{U}$ specifies the type of the method directly implemented by c_1 and $(\vec{l'}){:}\vec{U}'$ those inherited from c_2 (i.e., implemented by c_2 or further inherited by a class higher up in the hierarchy). The premises $\Gamma; \Delta \vdash f_k : T_k$ and $\Gamma; \Delta \vdash m_i : U_i$ check the well-typedness of all implemented members of the class (where we silently assume that f_k ranges over all fields and m_i over all methods implemented by the class and mentioned in $\vec{l}_f$ resp. in $\vec{l}$ of the signature. That also implies that the class does not provide code for methods labeled by labels from $(\vec{l'})$). The inherited methods are dealt with in the last premise $\Gamma; \Delta \vdash c_2.l'_j : U'_j$. The $c_2.l'_j : U'_j$ is a short-hand for looking up the type of l'_j from the interface information of c_2, i.e., for $\Gamma; \Delta \vdash c_2 : [\![S_2]\!]$ where $S_2 = \ldots l'_j{:}U_j \ldots$ or $S_2 = \ldots (l'_j){:}U_j \ldots$. I.e., the

type of l'_j is checked to coincide with the interface information of c_1 independent of whether the super-class implements l'_j directly or whether it's inherited. Typing for objects in rule T-Obj works similar, where c is the class the object instantiates. As the implementation of objects embeds the implementation of methods into the object, we need to check both fields and methods here, against the interface type of class c. The rest of the rules are straightforward, including the ones for expressions from Table 4.

$$\frac{\begin{array}{c}\Gamma;\Delta \vdash c_1 : [\![(\vec{l}_f{:}\vec{T}, \vec{l}{:}\vec{U}, (\vec{l'}){:}\vec{U}')]\!] \quad \Gamma;\Delta \vdash f_k : T_k \\ \Gamma;\Delta \vdash m_i : U_i \quad m_i = \varsigma(s_i{:}c_1).\lambda(\vec{x}_i{:}\vec{T}_i).t_i \quad \Gamma;\Delta \vdash c_2.l'_j : U'_j\end{array}}{\Gamma;\Delta \vdash [\![(c_2, \vec{l}_f = \vec{f}, \vec{l} = \vec{m})]\!] : c_1} \text{ T-Class}$$

$$\frac{\Gamma \vdash c.l_i : T_i \qquad \Gamma \vdash c.l'_j : U'_j \qquad \Gamma;\Delta \vdash f_i : T_i \qquad \Gamma;\Delta \vdash m_j : U'_j}{\Gamma;\Delta \vdash [l_1 = f_1, \ldots, l_k = f_k, l'_1 = m_1, \ldots, l'_n = m_n, L] : c} \text{ T-Obj}$$

$$\frac{\Gamma, \vec{x}{:}\vec{T};\Delta, s{:}c \vdash t : T'}{\Gamma;\Delta \vdash \varsigma(s{:}c).\lambda(\vec{x}{:}\vec{T}).t : \vec{T} \to T'} \text{ T-Memb} \qquad \frac{\Gamma;\Delta \vdash c : [\![(S)]\!]}{\Gamma;\Delta \vdash \bot_c : c} \text{ T-Undef}$$

$$\frac{\Gamma;\Delta \vdash v : c \quad \Gamma;\Delta \vdash c : T \quad \Gamma;\Delta \vdash v' : T.l}{\Gamma;\Delta \vdash v.l := v' : c} \text{ T-FUpdate} \qquad \frac{\Gamma;\Delta \vdash c : [\![(S)]\!]}{\Gamma;\Delta \vdash \mathsf{new}\, c : c} \text{ T-NewC}$$

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

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

$$\frac{\Gamma;\Delta \vdash v : c \quad \Gamma;\Delta \vdash c : [\![(\ldots, l{:}T, \ldots)]\!] \quad \Gamma;\Delta \vdash e_1 : T_2 \quad \Gamma;\Delta \vdash e_2 : T_2}{\Gamma;\Delta \vdash \mathsf{if}\, \mathsf{undef}(v.l()) \,\mathsf{then}\, e_1 \,\mathsf{else}\, e_2 : T_2} \text{ T-Cond}_\bot$$

$$\frac{}{\Gamma;\Delta \vdash \mathsf{stop} : T} \text{ T-Stop} \qquad \frac{}{\Gamma;\Delta \vdash () : \mathsf{Unit}} \text{ T-Unit}$$

Table 3. Typing (1)

The next example illustrates the type system, in particular the type checking of classes and the role of the interfaces.

Example 1 (Type checking of classes). Assume two classes c_1 and c_2, where c_1 extends c_2. Assume further that c_1 implements the two methods labelled l_1 and l_3, and that the super-class c_2 implements l_1 and l_2. The expected interfaces for the two classes are therefore

$$[\![(S_1)]\!] = [\![(l_1{:}U_1, (l_2){:}U_2, l_3{:}U_3)]\!] \quad \text{and} \quad [\![(S_2)]\!] = [\![(l_1{:}U_1, l_2{:}U_2)]\!] \tag{5}$$

for c_1 and c_2 respectively. As seen in the (right-hand) interface of equation (5), the available methods of instances of c_1 are l_1 (implemented by c_1, and

$$\frac{\Gamma;\Delta \vdash p : \langle T\rangle \quad \Gamma;\Delta \vdash o{:}c}{\Gamma;\Delta \vdash \mathsf{claim@}(p,o) : T}\ \text{T-CLAIM} \qquad \frac{\Gamma;\Delta \vdash p : \langle T\rangle}{\Gamma;\Delta \vdash \mathsf{get@}p : T}\ \text{T-GET}$$

$$\frac{\Gamma(x) = T}{\Gamma;\Delta \vdash x : T}\ \text{T-VAR} \qquad \frac{\Delta(n) = T}{\Gamma;\Delta \vdash n : T}\ \text{T-NAME} \qquad \frac{\Delta \vdash o : c}{\Gamma;\Delta \vdash \mathsf{suspend}(o) : \mathsf{Unit}}\ \text{T-SUSPEND}$$

$$\frac{\Delta \vdash o : c}{\Gamma;\Delta \vdash \mathsf{grab}(o) : \mathsf{Unit}}\ \text{T-GRAB} \qquad \frac{\Delta \vdash o : c}{\Gamma;\Delta \vdash \mathsf{release}(o) : \mathsf{Unit}}\ \text{T-RELEASE}$$

$$\frac{\Gamma;\Delta \vdash v : c \quad \Gamma;\Delta \vdash c.l : \vec{T} \to T \quad \Gamma;\Delta \vdash v_i : T_i}{\Gamma;\Delta \vdash v@l(\vec{v}) : T}\ \text{T-CALL} \qquad \frac{\Gamma;\Delta \vdash t : T \quad T \leq T'}{\Gamma;\Delta \vdash t : T'}\ \text{T-SUB}$$

Table 4. Typing (2)

overriding the corresponding method of c_2), l_2 (which is *not* implemented by c_1 but inherited), and l_3, which again is implemented by c_1. The derivation for both classes ends with an instance of rule T-PAR:

$$\frac{\Delta_1 \vdash c_1[\![c_2, l_1 = m_1, l_3 = m_3]\!] : (c_1{:}[\![S_1]\!]) \quad \Delta_2 \vdash c_2[\![l_1 = m_1', l_2 = m_2]\!] : (c_2{:}[\![S_2]\!])}{\Delta_0 \vdash c_1[\![c_2, l_1 = m_1, l_3 = m_3]\!] \parallel c_2[\![l_1 = m_1', l_2 = m_2]\!] : (c_1{:}[\![S_1]\!], c_2{:}[\![S_2]\!])}\ \text{T-PAR} \tag{6}$$

Note that the interface $[\![S_2]\!]$ for c_2 is used as *assumption* to type-check c_1 and vice versa. In the derivation, we use the following abbreviations:

$$\begin{aligned} \Delta_3 &\triangleq \Delta_1, c_1{:}[\![S_1]\!] \\ \Delta_2 &\triangleq \Delta_0, c_1{:}[\![S_1]\!] \\ \Delta_1 &\triangleq \Delta_0, c_2{:}[\![S_2]\!] \end{aligned} \tag{7}$$

The first premise of equation (6) gives rise to the following sub-derivation:

$$\frac{\dfrac{\Delta_3 \vdash c_1 : [\![S_1]\!] \quad \Delta_3 \vdash m_1 : U_1 \quad \Delta_3 \vdash m_3 : U_3 \quad \Delta_3 \vdash c_2.l_2 : U_2}{\Delta_3 \vdash [\![c_2, l_1 = m_1, l_3 = m_3]\!] : c_1}\ \text{T-CLASS}}{\Delta_1 \vdash c_1[\![c_2, l_1 = m_1, l_3 = m_3]\!] : (c_1{:}[\![S_1]\!])}\ \text{T-NCLASS} \tag{8}$$

To type-check the second premise of equation (6) works similarly. □

3.3 Operational semantics

The operational semantics is given in two stages, component internal steps and external ones, where the latter describe the interaction at the interface. Section 3.3.1 starts with component-internal steps, i.e., those definable without reference to the environment. In particular, those steps have no externally observable effect. The external steps, presented afterwards in Section 3.3.2, define

the interaction between component and environment. They are defined in reference to assumption and commitment contexts. The static part of the contexts corresponds to the static type system from Section 3.2 on component level and takes care that, e.g., only well-typed values are received from the environment.

Remark 2 (Binding of fields). Objects encapsulate its state in the form of instance variables or fields. With sub-classing, the members (fields and methods) of a class may have access to the members of its super-class (even without the `super`-keyword) due to inheritance and late binding. At run-time, the access to a method is resolved by *late binding* (or dynamic binding or dynamic dispatch). In Java, access to fields and access to methods are treated differently. Listing 1.5 illustrates this.

Listing 1.5. Shadowing

```
class C1 {
  x;
  m () {.. x...}
  }

class C2 extends C1 {
  x;     // overriding/shadowing
  n () {  ... m()   ...}
}
```

The body of method n on an instance of C_2 calls m, which is inherited from class C_1 to C_2. Method m in turn refers to a field x which is defined both in C_1 and C_2. In this situation, the version of the super-class C_1 is meant, i.e., the access to x is not resolved by late binding. This is different from the situation, if x would be a method; in that case the variant from the sub-class C_2 would be meant. Hence, replacing direct access to a field by using accessor methods like get and set methods *changes* the behavior in the presence of overriding and late-binding.

Listing 1.6. Accessor methods

```
class C1 {
  x;
  getx() { x }
  m () {.. self.getx()...}
  }

class C2 extends C1 {
  x;
  getx()   { x }
  n () {  ... m()   ...}
}
```

In the code of Listing 1.6, the inherited method m refers to *getx* of the subclass C_2 and hence refers to field x of that sub-class (in contrast to the analogous situation of Listing 1.5 before). □

Remark 3 (Delegation vs. embedding). An object "contains" fields and methods. There are different ways of how to represent objects in an implementation (or here the operational semantics). One way is to consider the object as a record

containing *both* fields and methods. This approach is known as "embedding". Of course, in absence of method updates, that leads to unnecessary code duplication. So alternatively, one may choose not to embed the methods, but keep them separate, and just refer to the classes defining them. The collection of methods is also called the method suite, and a call to the object *delegates* the call to that method suite. In our semantics, we follow the "naive" approach and embed the methods into the objects. In languages with class-based inheritance, the embedding and the delegation approach are observationally indistinguishable (cf. for instance [1]), if one ignores efficiency.

Methods are late bound i.e., it's the run-time type of an object which determines the method code, and not the static type. A way of interpreting the difference between embedding and delegation is that they are different as to when the dynamic binding is resolved: At the time the object is instantiated, or at the time when the method is actually called. □

3.3.1 Internal steps The internal steps rewrite components as given in the abstract grammar (cf. Table 1). In the configurations, one can distinguish two parts, a "mutable" and a fixed one. The parts that change are the threads, which are being executed, and the objects, which form the mutable heap. Immutable are the classes which are referenced when doing method look-up and which are arranged in the inheritance hierarchy. To simplify the writing of the operational rules, we factor out the immutable class hierarchy. A configuration of the closed semantics is then of the form

$$\Gamma^c \vdash C \ , \tag{9}$$

where C contains the parallel composition of all instantiated objects and all running threads and the class table Γ^c contains all class definitions. To stress the distinction between the mutable and the immutable part, we use $\vdash$ as separator (and not the parallel composition, as in the abstract syntax). With the classes being immutable, the operational steps do not change Γ^c and are thus of the form

$$\Gamma^c \vdash C \rightarrow \Gamma^c \vdash C' \ . \tag{10}$$

In the semantics later, we will distinguish confluent steps $\rightsquigarrow$ and non-confluent ones $\xrightarrow{\tau}$; when being unspecific we simply write $\rightarrow$ for internal transition relation. Actually, the information in Γ^c is needed only at one point, namely when binding a method call resp. a field access to the corresponding code resp. to the data location. In the embedding representation, this binding is established when a new object is instantiated (cf. rule NEWO and Definition 1 below); no other (internal) step actually refers to Γ^c; in the rules of Table 6, we omit mentioning Γ^c, except in the rule NEWO for instantiation where it is needed.

The internal semantics describes the operational behavior of a *closed* system, not interacting with an environment. The corresponding reduction steps are shown in Table 6, distinguishing between confluent steps $\rightsquigarrow$ and other internal transitions $\xrightarrow{\tau}$, both invisible at the interface. The $\rightsquigarrow$-steps, on the one hand, do

not access the instance state of the objects. They are free of side effects and race conditions, and hence confluent. The $\xrightarrow{\tau}$-steps, in contrast, access the instance state, either by reading or by writing it, and may thus lead to race conditions. In other words, this part of the reduction relation is in general not confluent.

The first seven rules deal with the basic sequential constructs, all as $\rightsquigarrow$-steps. The basic evaluation mechanism is substitution (cf. rule RED). Note that the rule requires that the leading let-bound variable is replaced only by *values* v. The operational behavior of the two forms of conditionals are axiomatized by the four COND-rules. Depending on the result of the comparison in the first pair of rules, resp., the result of checking for definedness in the second pair, either the then- or the else-branch is taken. Evaluating stop terminates the thread for good, i.e., the rest of the thread will never be executed as there is no reduction rule for $p\langle \mathsf{stop} \rangle$ (cf. rule STOP).

For accessing the fields of an object (to update the field or to read it), the object containing the field is consulted.[9] Remember further that we assume that fields are never accessed directly but only via corresponding accessor methods ("get" and "set") and that we *interpret* the notations $x.l()$ and $v.l() := v$ to represent those accessor methods. Rule FGET deals with field look-up. In the rule, $F.l$ stands for $\bot_c$, resp., for v, where $o[M, F, L] = o[\ldots, l = \bot_c, \ldots, L]$, if the field is yet undefined, resp., $o[M, F, L] = o[M, \ldots, l = v, \ldots, L]$. In rule FSET, the meta-mathematical notation $F.l := v$ stands for $(\ldots, l = v, \ldots)$, when $F = (\ldots, l = v', \ldots)$. Rule NEWT captures the execution of an asynchronous method call $o@l(\vec{v})$; the step creates a new thread p which at the same time serves are future reference to the later result. As the identity is fresh and not (yet) known to threads other than the creating one, the configuration is enclosed inside a ν-scope. The expression $p\langle call\ o.l(\vec{v})\rangle!$ describes the message for the method call.

Rule CALL deals with receiving an internal method call of method l with object o as the callee. Being an internal method call means that the code of the method is implemented by the component and not the environment. In our semantic representation based on embedding, the question whether the method labelled l in object o is implemented by the component or by the environment is already resolved (see the rule for object instantiation below). Remember also that in the open semantics later, the object will be *"split"* into two halves, one of the component and one of the environment, and the component configuration and the corresponding reduction rules deal only with the component half of an object. So, if the part of the object o represented in the rule *contains* the method l means l *is* a component method (by the fact that it code has been embedding into the component half of o).

In the embedding representation of objects, the point in time where the binding is resolved is when instantiating a new object (cf. rule NEWO). To de-

[9] In the current semantics, the object contains all fields; in the open semantics later, the object members, i.e., the fields and the methods are distributed over the component and the environment, and only the fields of the object implemented by the component show up in the (internal) rules.

termine which fields and methods are meant in a call is formalized in the function *members*. The function uses the class hierarchy and implements the search through the class hierarchy collecting the members supported by an instance of the given class. We have to distinguish between fields and methods. Methods are late-bound and thus, the method nearest in the class hierarchy reachable is the one supported by an instance. To model private methods (not directly supported by the *abstract* syntax), one could assume that all private methods are named differently, i.e., a private method in a class is named differently from all other (private or public methods).[10] Fields are considered private and thus subject to the same naming convention as the one for private methods. In Listing 1.5, the two fields x would be named differently, for instance x_1 in class C_1 and x_2 in C_2, such that the late-bound method m refers to x_1, as intended. When using accessor method, as in Listing 1.6, the fields, being considered private, are to be named differently. For the corresponding accessor methods, the user has the choice: if considered public, they follow the discipline of late binding and overriding, as explained in connection with Listing 1.6. Of course, renaming a field or method does not per se render it private, since being private means some access restrictions, as well. Especially, a private method or field cannot be accessed from a subclass. But those restrictions are captured by the type system. We insist that for each pair of get/set accessor methods, either both are considered private or both public.

For the function to implement the embedding in Definition 1, Rule M-Top deals with a class without super-class (which corresponds to `Object` in Java), in which case the fields and functions available are simply the ones as defined in the class. Sub-classing is covered by rule M-Inh. Methods from an instance of the subclass c_1 are taken from c_1 and c_2, with those of c_1 taking priority, i.e., one takes only those methods available at c_2, which are not provided directly from c_1, written $M_2 \setminus M_1$. For fields, we do not need to ignore fields from c_2, since all fields are considered being named differently, so no confusion can arise. That we copy in fields also from the super-classes does not imply that they are actually accessible in the corresponding instance. Privacy restrictions, however, are dealt with statically by the type system, not by the *members*-function.

Definition 1 (Embedding). *Given a class hierarchy Γ^c and a class name c, then the function members is given inductively in Table 5:*

$$\frac{\Gamma^c \vdash c = [\![\bot, M, F]\!]}{\Gamma^c \vdash \mathit{members}(c) = M, F}\ \text{M-Top} \qquad \frac{\Gamma^c \vdash c_1 = [\![c_2, M_1, F_1]\!] \quad M = M_1, M_2 \setminus M_1 \quad F = F_1, F_2 \quad \Gamma^c \vdash \mathit{members}(c_2) = M_2, F_2}{\Gamma^c \vdash \mathit{members}(c_1) = M, F}\ \text{M-Inh}$$

Table 5. Members

[10] We furthermore do not consider overloading here.

With the embedding of Definition 1, the instantiation of rule NEWO is rather straightforward. The new-statement creates a new instance with a fresh name, o in the rule. Since the reference is fresh, it appears under the ν-binder in the post-configuration.

$$n\langle \mathsf{let}\, x{:}T = v \,\mathsf{in}\, t\rangle \rightsquigarrow p\langle t[v/x]\rangle \qquad \textsc{Red}$$

$$p\langle \mathsf{let}\, x_2{:}T_2 = (\mathsf{let}\, x_1{:}T_1 = e_1 \,\mathsf{in}\, e) \,\mathsf{in}\, t\rangle \rightsquigarrow p\langle \mathsf{let}\, x_1{:}T_1 = e_1 \,\mathsf{in}\, (\mathsf{let}\, x_2{:}T_2 = e \,\mathsf{in}\, t)\rangle \qquad \textsc{Let}$$

$$p\langle \mathsf{let}\, x{:}T = (\mathsf{if}\, v = v \,\mathsf{then}\, e_1 \,\mathsf{else}\, e_2) \,\mathsf{in}\, t\rangle \rightsquigarrow p\langle \mathsf{let}\, x{:}T = e_1 \,\mathsf{in}\, t\rangle \qquad \textsc{Cond}_1$$

$$p\langle \mathsf{let}\, x{:}T = (\mathsf{if}\, v_1 = v_2 \,\mathsf{then}\, e_1 \,\mathsf{else}\, e_2) \,\mathsf{in}\, t\rangle \rightsquigarrow p\langle \mathsf{let}\, x{:}T = e_2 \,\mathsf{in}\, t\rangle \quad \text{where } v_1 \neq v_2 \qquad \textsc{Cond}_2$$

$$p\langle \mathsf{let}\, x{:}T = (\mathsf{if}\, \mathsf{undef}(\bot_c) \,\mathsf{then}\, e_1 \,\mathsf{else}\, e_2) \,\mathsf{in}\, t\rangle \rightsquigarrow p\langle \mathsf{let}\, x{:}T = e_1 \,\mathsf{in}\, t\rangle \qquad \textsc{Cond}_1^{\bot}$$

$$p\langle \mathsf{let}\, x{:}T = (\mathsf{if}\, \mathsf{undef}(v) \,\mathsf{then}\, e_1 \,\mathsf{else}\, e_2) \,\mathsf{in}\, t\rangle \rightsquigarrow p\langle \mathsf{let}\, x{:}T = e_2 \,\mathsf{in}\, t\rangle \qquad \textsc{Cond}_2^{\bot}$$

$$p\langle \mathsf{let}\, x{:}T = \mathsf{stop} \,\mathsf{in}\, t\rangle \rightsquigarrow p\langle \mathsf{stop}\rangle \qquad \textsc{Stop}$$

$$o[c,M,F,L] \parallel p\langle \mathsf{let}\, x{:}T = o.l() \,\mathsf{in}\, t\rangle \xrightarrow{\tau} o[c,M,F,L] \parallel p\langle \mathsf{let}\, x{:}T = F.l \,\mathsf{in}\, t\rangle \qquad \text{FG}\textsc{et}$$

$$o[c,M,F,L] \parallel p\langle \mathsf{let}\, x{:}T = o.l() := v \,\mathsf{in}\, t\rangle \xrightarrow{\tau} o[c,M,F.l := v,L] \parallel p\langle \mathsf{let}\, x{:}T = o \,\mathsf{in}\, t\rangle \qquad \text{FS}\textsc{et}$$

$$p'\langle \mathsf{let}\, x{:}\langle T\rangle = o@l(\vec{v}) \,\mathsf{in}\, t\rangle \rightsquigarrow \nu(p{:}\langle T\rangle).(p'\langle \mathsf{let}\, x{:}\langle T\rangle = p \,\mathsf{in}\, t\rangle \parallel p\langle \mathit{call}\ o.l(\vec{v})\rangle!) \qquad \textsc{New}\text{T}$$

$$o[c,M,F,\bot] \parallel p\langle \mathit{call}\ o.l(\vec{v})\rangle! \xrightarrow{\tau} o[c,M,F,\top] \parallel p\langle \mathsf{let}\, x{:}T = M.l(o)(\vec{v}) \,\mathsf{in}\, \mathsf{release}(o); x\rangle \qquad \textsc{Call}$$

$$\frac{\Gamma^c \vdash \mathit{members}(c) = M, F}{\Gamma^c \vdash p\langle \mathsf{let}\, x{:}T = \mathsf{new}\, c \,\mathsf{in}\, t\rangle \rightsquigarrow \Gamma^c \vdash \nu(o{:}c).(o[c,M,F,\bot] \parallel p\langle \mathsf{let}\, x{:}T = o \,\mathsf{in}\, t\rangle)} \ \textsc{New}\text{O}$$

$$p_1\langle \mathsf{let}\, x : T = \mathsf{claim}@(p_2, o) \,\mathsf{in}\, t\rangle \parallel p_2\langle v\rangle \rightsquigarrow p_1\langle \mathsf{let}\, x : T = v \,\mathsf{in}\, t\rangle \qquad \textsc{Claim}_1$$

$$\frac{t_2 \neq v}{p_1\langle \mathsf{let}\, x : T = \mathsf{claim}@(p_2, o) \,\mathsf{in}\, t_1\rangle \parallel p_2\langle t_2\rangle \rightsquigarrow p_1\langle \mathsf{let}\, x : T = \mathsf{release}(o)\rangle; \mathsf{get}@p_2 \,\mathsf{in}\, \mathsf{grab}(o); t_1 \parallel p_2\langle t_2\rangle} \ \textsc{Claim}_2$$

$$p_1\langle \mathsf{let}\, x : T = \mathsf{get}@p_2 \,\mathsf{in}\, t\rangle \parallel p_2\langle v\rangle \rightsquigarrow p_1\langle \mathsf{let}\, x : T = v \,\mathsf{in}\, t\rangle \qquad \textsc{Get}$$

$$p\langle \mathsf{suspend}(o); t\rangle \rightsquigarrow p\langle \mathsf{release}(o); \mathsf{grab}(o); t\rangle \qquad \textsc{Suspend}$$

$$o[c,M,F,\bot] \parallel p\langle \mathsf{grab}(o); t\rangle \xrightarrow{\tau} o[c,M,F,\top] \parallel p\langle t\rangle \qquad \textsc{Grab}$$

$$o[c,M,F,\top] \parallel p\langle \mathsf{release}(o); t\rangle \xrightarrow{\tau} o[c,M,F,\bot] \parallel p\langle t\rangle \qquad \textsc{Release}$$

Table 6. Internal steps

Claiming as well as executing the get-expression fetches the value of a future reference. The two expressions differ, however, whether or not the lock may be released in case the requested future is not yet evaluated. Claiming a future fetches the value without releasing the lock, if the value is already availabe (cf. rule $\textsc{Claim}_1$), and works in that situation identical to getting the value in rule GET. If the value is not yet there, $\textsc{Claim}_2$ releases the lock temporarily, i.e.,

the thread attempts to re-acquire it immediately afterward. There is no rule corresponding to $\textsc{Claim}_2$ for get, i.e., trying to dereference a future reference via get blocks without releasing the lock. Release and grap are dual an set the lock to free resp. set it to the state $\top$ of "taken". Both operations are *not* user syntax. Suspend, finally, introduces a scheduling point by temporarily releasing then then trying to re-acquire the lock.

The above reduction relations are used modulo *structural congruence,* which captures the algebraic properties of parallel composition and the hiding operator. The basic axioms for $\equiv$ are shown in Table 7 where in the fourth axiom, n does not occur free in C_1. The congruence relation is imported into the reduction relations in Table 8. Note that all syntactic entities are always tacitly understood modulo α-conversion.

$$\mathbf{0} \parallel C \equiv C \qquad C_1 \parallel C_2 \equiv C_2 \parallel C_1 \qquad (C_1 \parallel C_2) \parallel C_3 \equiv C_1 \parallel (C_2 \parallel C_3)$$

$$C_1 \parallel \nu(n{:}T).C_2 \equiv \nu(n{:}T).(C_1 \parallel C_2) \qquad \nu(n_1{:}T_1).\nu(n_2{:}T_2).C \equiv \nu(n_2{:}T_2).\nu(n_1{:}T_1).C$$

Table 7. Structural congruence

3.3.2 Typed operational semantics for open systems Next we define the external semantics for the interaction between component and environment.

Interaction labels A component exchanges information via method calls and when getting back the result of a method call (cf. Table 9), i.e., via *call* and *get* labels (by convention, referred to as γ_c and γ_g, for short). Interaction is either incoming (?) or outgoing (!). In the labels, p is the identifier of the thread carrying out the call resp. of being queried via claim or get. Scope extrusion of fresh names across the interface is indicated by the ν-binder. In $\nu(n{:}T)_o$, the o represents the identity of the object that creates the thread or object n.

$$\frac{C \equiv \rightsquigarrow \equiv C'}{C \rightsquigarrow C'} \qquad \frac{C \rightsquigarrow C'}{C \parallel C'' \rightsquigarrow C' \parallel C''} \qquad \frac{C \rightsquigarrow C'}{\nu(n{:}T).C \rightsquigarrow \nu(n{:}T).C'}$$

$$\frac{C \equiv \xrightarrow{\tau} \equiv C'}{C \xrightarrow{\tau} C'} \qquad \frac{C \xrightarrow{\tau} C'}{C \parallel C'' \xrightarrow{\tau} C' \parallel C''} \qquad \frac{C \xrightarrow{\tau} C'}{\nu(n{:}T).C \xrightarrow{\tau} \nu(n{:}T).C'}$$

Table 8. Reduction modulo congruence

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

Table 9. Labels

Connectivity contexts and cliques An important condition in the rules of the external semantics concerns which *combinations* of names can occur in communications. This is a consequence of the fact that there are methods and fields implemented in the component and those implemented in the environment. Hence, each instance state is split into a component and into an (absent) environment half. A well-typed component thus takes into account the *relation* of objects 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. The *connectivity contexts* E_Δ and E_Θ over-approximate the heap structure, i.e., the pointer structure of the objects among each other, divided into the component part and the environment part. To facilitate notation, we use the following conventions.

Notation 1 (Contexts) *The semantics of an open component is given by labeled transitions between judgments of the form* $\Delta; E_\Delta \vdash C : \Theta; E_\Theta$*, where* Δ *and* Θ *are* name contexts *(cf. Definition 2) and* E_Δ *and* E_Θ connectivity contexts *(cf. Definition 3). We abbreviate the pair* $\Delta; E_\Delta$ *and* $\Theta; E_\Theta$ *of both assumption and commitment context by* Ξ*, i.e., we write* $\Xi \vdash C$ *for* $\Delta; E_\Delta \vdash C : \Theta; E_\Theta$*. The* Ξ_Δ *refers to the assumption context* $\Delta; E_\Delta$*, and* Ξ_Θ *to* $\Theta; E_\Theta$*. Furthermore we understand* $\acute{\Xi}$ *as consisting of* $\acute{\Delta}; \acute{E}_\Delta$ *and* $\acute{\Theta}; \acute{E}_\Theta$*, etc.*

Definition 2 (Name contexts). Δ *and* Θ *are the* assumption *and* commitment contexts *containing name bindings of the form* $n{:}T$*. More precisely, bindings* $o{:}c$ *for object names and* $p{:}\langle T\rangle$ *for future references/thread names. Additionally, we use* $\odot$ *to represent the initial activity/initial clique. The pair of* Δ *and* Θ *satisfy the following invariants. The* $\odot$ *is contained in either* Δ *or in* Θ *(indicating where the initial activity at the program start is located). Furthermore, if* $\Delta \vdash o{:}c_1$ *and* $\Theta \vdash o{:}c_2$*, then* $c_1 = c_2$*. Wrt. future references, the domains of* Δ *and* Θ *are disjoint, i.e., if* $\Delta \vdash p : \langle T\rangle$*, then* $\Theta \nvdash p : \langle T\rangle$*, and conversely. We write* Δ, Θ *for the "union" of both bindings, i.e.,* $\Delta, \Theta \vdash n : T$ *if* $\Delta \vdash n : T$ *or* $\Theta \vdash n : T$*.*

Definition 3 (Connectivity contexts). *The* assumption connectivity context *is a binary relation of the following form, where* Δ_o *refers to the object identities of* Δ*,* Θ_p *to the thread identities of* Θ*, etc.:*

$$E_\Delta \subseteq (\Delta_o \times \Delta_o) \cup (\Delta_o \times \Xi_p) \cup (\Delta_p \times \Delta_o). \tag{11}$$

Dually, $E_\Theta \subseteq (\Theta_o \times \Theta_o) \cup (\Theta_o \times \Xi_p) \cup (\Theta_p \times \Theta_o)$ *is the* commitment connectivity context*. We write* $n_1 \hookrightarrow n_2$ *("*n_1 *may know* n_2*") for pairs from these relations.*

In analogy to the name contexts Δ, connectivitycontexts E_Δ express assumptions about the environment, and E_Θ commitments of the component.

Remark 4 (Invariant). The connectivity context of equation (11) consists of three "parts". The part from $\Delta_o \times \Delta_o$ over-approximates which environment fields of objects may know which objects. Similarly a pair $o \hookrightarrow p$ from $\Delta_o \times \Xi_p$ indicates that the (environment half of) object o may know future p. Since we do not support first-class futures, which means, future references cannot be passed around as arguments, there is *exactly* one object with $o \hookrightarrow p$, which is the creator of that future. In our setting that is the *caller* of the corresponding method. The intuition for the pair of the form $p \hookrightarrow o$ is slightly different; it means that thread p is executing inside object o (and thus "knows" o via the self-parameter). More precisely, the thread has *started* executing in o by acquiring the lock, but so far the result has not been obtained via executing get, so the thread p is not yet garbage collected. As an invariant of the semantics, there is *at most* one object such that $p \hookrightarrow o$.

There is a further invariant, concerning $o_1 \hookrightarrow p$ and $p \hookrightarrow o_2$: if o_1 and p are both on "the same side", say $\Delta \vdash o_1$ and $\Delta \vdash p$, then there exists no o_2 such that $E_\Delta \vdash p \hookrightarrow o_2$ or $E_\Theta \vdash p \hookrightarrow o_2$. And conversely: If $E_\Delta \vdash p \hookrightarrow o_2$ (i.e., p executes in an environment object o_2), then $E_\Theta \vdash o_1 \hookrightarrow p$ for some o_1 with $\Theta \vdash o_1$ (i.e., the caller o_1 is active in Θ and its component fields know the thread/future p). □

As mentioned, the component has to over-approximate via E_Δ which environment parts of the objects are potentially connected, and, symmetrically, for the own part of the heap via E_Θ. The worst case concerning possible connections is represented by the reflexive, transitive, and symmetric closure of the $\hookrightarrow$-relation:

Definition 4 (Acquaintance). *Given Δ and E_Δ, we write $\leftrightharpoons$ for the* reflexive, transitive, *and* symmetric *closure of the $\hookrightarrow$-pairs of objects from the domain of Δ, i.e.,*

$$\leftrightharpoons \triangleq (\hookrightarrow\downarrow_{\Delta_o \times \Delta_o} \cup \hookleftarrow\downarrow_{\Delta_o \times \Delta_o})^* \subseteq \Delta_o \times \Delta_o \ , \tag{12}$$

where we write shorter $\Delta_o \times \Delta_o$ for $dom(\Delta_o) \times dom(\Delta_o)$.

Note that we close the $\hookrightarrow$-relation concerning the environment-part of the heap, only. As judgment, we use

$$\Delta; E_\Delta \vdash o_1 \leftrightharpoons o_2 \ . \tag{13}$$

For Θ and E_Θ, the definitions are applied dually. Furthermore we write $\Delta; E_\Delta \vdash o \hookrightarrow p$ if $o \hookrightarrow p \in E_\Delta$, an analogously $\Delta; E_\Delta \vdash p \hookrightarrow o$. Note that we use the transitive and reflexive closure for the connectivity among object identities, only.

Typed configurations The assumption contexts are an abstraction of the (absent) environment, consulted to *check* whether an *incoming* action is currently possible, and *updated* in an *outgoing* communication. The commitments play a dual role, i.e., they are updated in incoming communication. With the code of the component present, the commitment contexts are not used for checks for outgoing communication. Part of the check concerns type checking, i.e., basically whether the values transmitted in a label correspond to the declared types for

$$\frac{\Delta,\Theta \vdash c : [\![\vec{l}{:}\vec{U}, (\vec{l'}){:}\vec{U'}]\!] \qquad l \in \vec{l}}{\Delta,\Theta \vdash \mathit{find}(c,l) = c} \ \textsc{Find}_1$$

$$\frac{\Delta,\Theta \vdash c_1 : [\![\vec{l}{:}\vec{U}, (\vec{l'}){:}\vec{U'}]\!] \qquad l \notin \vec{l} \qquad \Delta,\Theta \vdash c_1 \leq_1 c_2 \qquad \Delta,\Theta \vdash \mathit{find}(c_2,l) = c_3}{\Delta,\Theta \vdash \mathit{find}(c_1,l) = c_3} \ \textsc{Find}_2$$

Table 10. Binding

the corresponding method. This is covered in the following two definitions, where then first one searches the class hierarchy to determine the class that implements a given member.

Definition 5 (Find). *Given Δ,Θ, the function find takes a class name and a member label l and returns the class which implements the member. The function is inductively given in Table 10.*

The rules for the find function of Table 10 work straightforwardly, determining the class a member is defined in. Unlike member function from Definition 1, the functions here uses the *interface* information to find the class. The member function from Table 5 for the closed semantics consults the class table to do the same; this is no longer possible, as we do not have the complete class table at hand in the open semantics.

Basically, the function searches the class hierarchy starting from c and moving to the super-classes and returns the first class that implements the member labelled l. In the base case of rule $\textsc{Find}_1$, the member l is found in the current class c: the signature $[\vec{l}{:}\vec{U}, (\vec{l'}){:}\vec{U'}]$ indicates that the member l is implemented by c as opposed to being inherited from a super-class. If c does *not* implement the member in that $l \notin \vec{l}$ (cf. rule $\textsc{Find}_2$), the function continues the search recursively with the immediate super-class c_2 of c_1, as stipulated by the premise $\Delta,\Theta \vdash c_1 \leq_1 c_2$. Note that the implementing class is found based on the *interface* information Δ,Θ, only.

Definition 6 (Well-typedness). *Let a be an incoming communication label. The assertion*

$$\Xi \vdash a \tag{14}$$

("under the context Ξ, label a is well-typed") is given by the rules of Table 11. For outgoing communication, the definition is dual.

For an incoming call to be well-typed (cf. rule LT-CallI), the callee name o and future/thread name p must already be known at the interface (as required by the first and the last premise). To be an interface interaction —here an incoming call from the environment to the component— the code of the method l must be located at the component side. This is assured by the second and third premise:

$$\frac{\Xi \vdash o : c \quad \Xi \vdash \mathit{find}(c,l) = c' \quad \Gamma^c_\Theta \vdash c' \quad \Xi \vdash c' : [\![\ldots, l{:}\vec{T} \to T, \ldots]\!] \quad \Xi \vdash \vec{v} : \vec{T} \quad \Delta \vdash p : \langle T \rangle}{\Xi \vdash p\langle call\ o.l(\vec{v})\rangle?} \text{ LT-CALLI}$$

$$\frac{\Delta \vdash p : \langle T \rangle \qquad \Xi \vdash v : T}{\Xi \vdash p\langle get(v)\rangle?} \text{ LT-GETI} \qquad \frac{\Xi \not\vdash n \qquad \Delta \vdash o{:}c \qquad \Xi \vdash T}{\Xi \vdash \nu(n{:}T)_o?} \text{ LT-NEWI}$$

Table 11. Checking static assumptions

The find-function determines the class c' where the method is implemented and $\Gamma^c_\Theta \vdash c'$ assures that the class is part of the component, as in the open semantics, only the class table Γ^c_Θ of the component is available. Finally, the declared type $\vec{T} \to T$ of the method is checked against the communicated values $\vec{v}$ and the future reference p, which is to reference the method's return value, must be of the matching type $\langle T \rangle$. Note that the last premise requires that the p is part of assumption environment Δ. Well-typedness for get-labels used to fetch the result from an asynchronous method calls is covered by rule LT-GETI, basically requiring that type $\langle T \rangle$ of p corresponds to the type T of the value v the name p references. Rule LT-NEWI finally deals with incoming communication of a fresh name n, either an object reference or a future reference. The only requirement is that the name is indeed fresh, and that the type mentioned in the label is actually a type (stipulated by $\Xi \vdash T$).

The interface interaction of the open system provides also information that updates the contexts.

Definition 7 (Name context update). *Let Ξ be a context and a an* incoming *label, with $\Xi \vdash a$ (cf. Definition 6). The updated context $\acute{\Xi} = \Xi + a$ is defined as follows (dually for outgoing communication):*

1. *If $a = \nu(n{:}T)_{o'}?$, then $\acute{\Delta} = \Delta, n{:}T$ and $\acute{\Theta} = \Theta$.*
2. *If $a = p\langle call\ o.l(\vec{v})\rangle?$, then $\acute{\Delta} = \Delta \setminus p$ and $\acute{\Theta} = \Theta, p{:}\langle T \rangle \cup (o{:}c, \vec{v}{:}\vec{T})$, where $\Delta \vdash p : \langle T \rangle$ and where the types $\vec{T}$ of the arguments $\vec{v}$ are determined by $\Delta \vdash v_i : T_i$.*
3. *If $a = p\langle get(v)\rangle?$, then $\acute{\Delta} = \Delta \setminus p$ and $\acute{\Theta} = \Theta \cup v{:}T$, where $\Delta \vdash p : \langle T \rangle$.*

Part 1 covers communication of a fresh identity n where the assumption context Δ is extended by the type information for the new identifier n; the commitment context Θ is left unchanged. If n represents a future reference, (assumed to be) freshly created by the environment, the update from Δ to $\acute{\Delta}$ captures the intuition that the new thread/future reference is issued by an *asynchronous* call by (another) thread in the environment, and that initially, before actually grabbing the lock, the activity resides in the environment. If n represents a reference to an object instantiated by the environment, the intuition is as follows. As mentioned, the instance state of an object in the open semantics is split into

two halves, one implemented by the component and one by the environment, which therefore is not represented in the open configuration. At the time, when an object is instantiated by the environment (which is the situation for incoming communication), the new object identifier is communicated at the interface through the ν-label, and the *half* of the object belonging to the *environment* is already instantiated, i.e., its fields and methods are (assumed to be) embedded. The members of the component, however, are *not yet* embedded, i.e., after the fresh object identifier has been communicated, only *one half* of the object is instantiated, namely the half at the side, which executed the instantiation command; in the case of incoming communication, that is the environment. Remember from the conditions on Δ and Θ from Definition 2, that a binding $o{:}c$ for an object identifier can be contained in Δ or Θ or in both (in the latter case with the same type c). After $\nu(o{:}c)_{o'}?$, the o is given a type in the environment context, only.

That changes in part 2 which deals with incoming call labels. The communication of the call label at the interface represents the moment where the method actually grabs the lock of the callee, o in this case. At that point, the thread p changes from the side of the caller to the side where the method body is implemented. This means, the corresponding binding $p{:}\langle T\rangle$ is removed from Δ and added to Θ. That preserves the invariant from Definition 2, that future/thread names are either bound in Δ or in Θ but not in both. Part 2 updates Θ also wrt. the callee identity o. Remember from the discussion in part 1 that in the communication step $\nu(o{:}c)_{o'}?$, the corresponding binding $o{:}c$ is added to Δ, only. In the call-step now, also Θ is extended by that binding. Part 3 finally updates the name contexts in case of an incoming get-communication. As our language does not support first-class futures, each future is referenced at most once; afterwards it can be garbage collected. This is reflected in the update, in that we remove the binding from the corresponding context, here Δ.

Remark 5 (Instantiation). In the closed semantics, when an object is instantiated, a new object reference is created and the code of the members (fields and methods) is embedded into the newly created object. In the open semantics, only one half of the members are actually available for the component, namely those members whose code is implemented by a component class. So conceptually the object state is split into two halves, only implemented by the component, the other half belonging to the environment and only abstractly represented by the assumption contexts. One question in the open semantics is: at which point becomes an instantiation visible at the interface (via a $\nu(o{:}c)_{o'}$-label) and when the object is actually instantiated in the sense that the member record is embedded into the object.

From the standpoint of observability, the point where the new identifier is created is not necessarily identical with when that becomes known at the interface. Our calculus does not support *constructors,* which means that an instantiation alone has no real observable effect. Nonetheless, in the open semantics, an ν-label is exchanged at the exact point of instantiation. This is in contrast with the treatment in related work dealing with open semantics and full abstraction

$$\frac{}{\Xi \vdash \nu(n{:}T)_o?} \qquad \frac{\Delta \vdash o' : c \quad E_\Delta \vdash o' \hookrightarrow p \quad E_\Delta \vdash o' \hookrightarrow o, \vec{v}}{\Xi \vdash p\langle call\ o.l(\vec{v})\rangle?} \qquad \frac{E_\Theta \vdash o' \hookrightarrow p \quad E_\Delta \vdash p \hookrightarrow o \hookrightarrow v}{\Xi \vdash p\langle get(v)\rangle?}$$

Table 12. Connectivity check

for object-oriented languages (but without inheritance), where the ν-label is communicated only at the point in time where the object is first accessed ("lazy instantiation"). See for instance [21,4].

A complication in the setting with inheritance is the fact that the object state is split into two halves. Indeed, both halves are *not* instantiated at the same time: The half belonging to the side of the instantiator is instantiated immediately, whereas the half on the opposite side is instantiated only at the time of the first method call (as in lazy instantiation). The delay of instantiation is reflected in the two cases (1) and (2) of the name context update of Definition 7.

There is a reason why the second half is not instantiated eagerly. Each call constitutes a communication from a caller object to a callee object. For the purpose of checking possible connectivity, it is important to remember the caller in a call; in case of an incoming call, where (the environment half of) the caller is not available, the origin of the call is *guessed* and remembered (the guessing is not done as part of the incoming call communication, but at the point where the new thread identity is communicated in rule NewTI). However, immediately after object instantiation, an object cannot be the source of the call. That is only possible if the object had been the target of a call itself. That an object (half) can be the source of a call is part of the connectivity check in Definition 8, in the case for calls. □

The checks of the *connectivity* assumptions are formalized as follows:

Definition 8 (Connectivity context check). *Let Ξ be a context and a be an incoming communication label. Overloading the notation from Definition 6, we write $\Xi \vdash a$ if the conditions of Table 12 are met. For outgoing communication, the definition works dually. In the semantical rules, $\Xi \vdash a$ means that both typing and connectivity are checked.*

For incoming ν-labels, the connectivity is not checked. Remember from rule LT-NewI, that for well-typedness, the environment on the other side needs to contain at least one object, required by $\Delta \vdash o{:}c$ in the premise of the rule. For incoming method calls, the caller, o' in the rule is the object that issued the asynchronous method call, checked by $o' \hookrightarrow p$, where p is the thread to execute the method body and furthermore o' must be contained in the environment (by $\Delta \vdash o'$). For fetching the result of a method call via get, the caller o' must know the thread/future reference p, and since it is an incoming communication, the acquaintance must follow from the *commitment* context E_Θ which implies that

the call had been issued by the half of o' contained in the component, not the environment. Note further that the well-typedness assumption for incoming get-communication requires (by the premise $\Delta \vdash p : \langle T \rangle$), that the thread is actually on the environment side, not the component side. The last two conditions assure that that prior call had been issued already (as an outgoing call from the component to the environment) and that the thread p is not just been created without actually having started executing. The remaining premises in the rules for calls, resp. for get-labels require that the "sender" of the information know the transmitted arguments. In the case of incoming calls, the sender is the caller, o' in the rules. That it knows the arguments and caller o is required by $E_\Delta \vdash o' \hookrightarrow o, \vec{v}$. For the incoming get-label, the sender of the information is the callee o, which is required to know the argument v. In the premise $E_\Delta \vdash p \hookrightarrow o \hookrightarrow v$, the part $E_\Delta \vdash p \hookrightarrow o$ determines o as the caller; in the connectivity update later, by adding a pair $p \hookrightarrow o$ for method calls, the caller is remembered.

For *updating* connectivity, communication may bring objects in connection which had been separate before, i.e., it may merge object cliques. For an incoming call, this can be directly formulated by adding the fact that the receiver of the communication now is acquainted with all transmitted arguments. As far as the thread p is concerned: the fact that p starts executing in the callee o after the call is remembered by adding $p \hookrightarrow o$ to the commitment connectivity. See part 2 of Definition 9 below. Similarly in part 3 for incoming get information: the object o dereferencing the future p now knows the value v communicated in the communication. The object o is determined by the condition $E_\Theta \vdash o \hookrightarrow p$. Since the future reference/thread is garbage collected after dereferencing, the connection $o \hookrightarrow p$ is removed from the connectivity context, as well. Furthermore removed is the pair $p \hookrightarrow o'$, where o' indicates the object that has executed the method body leading to the result v. As mentioned, the incoming information updates basically the connectivity for the commitment, but that is the case only for the two cases 2 and 3 just discussed. For incoming *fresh* identifiers in case 1, the *assumed* connectivity of the environment is updated, namely by the assumption that the originator o' of the new identifier n knows it.

Definition 9 (Connectivity context update). *Assume $\Xi \vdash a$. The update of the connectivity contexts $\acute{\Xi} = \Xi + a$ is defined as follows.*

1. *If $a = \nu(n{:}T)_{o'}?$, then $E'_\Delta = E_\Delta, o' \hookrightarrow n$.*
2. *If $a = p\langle call\ o.l(\vec{v})\rangle?$, then $E'_\Theta = E_\Theta, o \hookrightarrow \vec{v}, p \hookrightarrow o$.*
3. *If $a = p\langle get(v)\rangle?$, then $E'_\Theta = (E_\Theta, o \hookrightarrow v) \setminus (o \hookrightarrow p)$, where $E_\Theta \vdash (o \hookrightarrow p)$, and $E'_\Delta = E_\Delta \setminus (p \hookrightarrow o')$ (where $E_\Delta \vdash p \hookrightarrow o'$).*

The definition for outgoing communication is dual.

External steps The semantics is given as labeled transitions between typing judgments of the form

$$\Delta; E_\Delta \vdash C : \Gamma^c_\Theta, \Theta; E_\Theta; \qquad (15)$$

Note that only the class table Γ^c_Θ of class definitions of the component is available, the environment classes are missing. As Γ^c_Θ does not change during execution, we assume it given implicitly and do not mention it explicitly in the rules.

As before, we abbreviate the judgment of equation (15) as $\Xi \vdash C$ (cf. Notation 1). The steps of the external semantics are of the form

$$\Xi \vdash C \xrightarrow{a} \acute{\Xi} \vdash \acute{C} \ . \tag{16}$$

Based on the previous definitions to check and update the context information, the typed operational rules of the external semantics are given in Table 13. Conceptually, the rules fall into two groups, namely those for incoming communication and those for outgoing communication (plus a few internal ones).

As shown in equation (15) also the class table Γ^c is split into an assumption and a commitment half (Γ^c_Δ and Γ^c_Θ). As the environment part Γ^c_Δ is not available, instantiation can embed only those members of a new object which are actually provided by Γ^c_Θ. We have to adapt Definition 1 for embedding fields and methods during instantiation to deal with the fact that the whole class table is no longer available. Given a class table Γ^c_Θ plus the *interface information,* which in particular contains information about the class hierarchy, the function *members* looks up the implementation of the members of an instance of class c.

Definition 10. *Let Δ, Θ be a well-formed typing context, Γ^c_Θ the component half of the class table, and c a class name. Given Δ, Θ and Γ^c_Θ, the function members on class names c is defined as follows:*

$$members(c) = \{R.l \mid \Delta, \Theta \vdash find(c, l) = c' \text{ and } \Gamma^c_\Theta = c'[\![R]\!], {\Gamma^c}'_\Theta \}. \tag{17}$$

The definition for Γ^c_Δ works dually.

Using the *find*-function from Definition 5, the function *members* finds the implementation for all (public) component members of a class c. As the function returns the code, the interface information Δ, Θ alone is not good enough, we need the class table. Once, *find* has determined the (name of the) class, the class table Γ^c_Θ is consulted to extract the methods and fields of the class, from which $R.l$ pics the intended one.

Now to the operational rules of the open semantics. The first four rules of Table 13 deal with exchange of "new" information, i.e., with identifiers created at one side and communicated to the other. In rule NewOO, the component instantiates a new object. Executing the new c-expression creates o as a fresh identifier and the component heap is extended by the new object instance $o[c, M, F, \bot]$. In our semantics, that object represents only one half of the global view on the object, namely the half which contains the record M, F of those members (methods and fields) actually implemented by component classes. The function *members* determines that record and embeds it into o, consulting the interface information Δ and Θ (as part of Ξ) and the component half of the class-table Γ^c_Θ. Immediately after instantiation, the lock is free, represented by $\bot$. The step of the component is labelled by $\nu(o{:}c)_{o'}!$, which is used to update the interface information in Ξ to $\acute{\Xi} = \Xi + a$. Part of the label is the creating object o' whose identity is *determined* by the premise $E_\Theta \vdash p \hookrightarrow o'$. The second part of the context information which is updated by $\Xi + a$ is the *connectivity.* In case

of NewOO, the label communicates information about a new identity and it's the sender's connectivity information which is updated, which means for outgoing communication, the connectivity of the component side. For the receiving, environment side, the object o is not yet added to the corresponding context Δ (see Definition 7(1)). For the communication labels later, which do not deal with transmitting fresh information, the situation is dual: sending information from the component to the environment updates the environment information (especially connectivity), not the component information. Rule NewOI is dual to NewOO and deals with the situation that a new object identity is transmitted from the environment to the component, indicated by a label of the form $\nu(o{:}c)_{o'}?$. The premises $\Xi \vdash a$ and $\acute{\Xi} = \Xi + a$ check whether the communication is possible, resp., update the context appropriately. The premise $\Xi \vdash a$ for checking whether the interaction a is possible as a next step has not been present (in dual form) in NewOO: For steps initiated by the component, such as creating a new object and publishing its identity at the interface, it is not necessary whether the step is actually possible: the fact that the code executes the state shows that it's possible. Note that unlike in rule NewOO, no object half is actually instantiated in step (see Remark 5). Outgoing calls are dealt with by the rules NewTO and CallO. In NewTO, the component executes the expression $o@l(\vec{v})$ for asynchronous method calls, creating a new process (and future reference) p and a message $p\langle call\ o.l(\vec{v})\rangle!$. The step does not distinguish between internal and external method calls. The fresh identity p of the new thread is immediately communicated to the environment by the label $\nu(p{:}\langle T\rangle)_{o'}!$, and the contexts Ξ is updated to $\acute{\Xi}$ appropriately (the creator o' of the thread is determined in the same way as in rule NewOO). Rule NewTI deals with the dual situation. As in general for steps of the environment, we need to check whether the step is possible, which is done by the premise $\Xi \vdash a$.

The message for an outgoing call is communicated at the interface in rule CallO, i.e., the rule describes as situation continuing from a configuration after a NewTO-step. To be an external call requires that the callee object o does *not* implement the called method l (formulated by the premise $M.l = \bot$). Since $M.l = \bot$ and since we assume all programs to be well-typed, the method must be implemented by the environment and thus is assumed to be embedded in the environment part of the object. Another pre-condition for the step concerns the *lock* of the object. Note that we assume that the interface interaction representing an outgoing call *atomically* captures the step when the lock is actually taken. Since in the configuration, we conceptually represent *only* the perspective of the component on the "shared" lock, we require that, from the perspective of the component, it is free by requiring that the object is of the form $o[c, M, F, \bot]$. Even if we don't know whether the environment has "actually" taken the lock or not, the CallO-step is enabled based on that fact that the *component does not* hold the lock. Having abstracted away from the environment, it's enough to know that there *exists* an environment that currently does not hold the lock, in other words, that the lock *may* be free. Note further that *after* the CallO-step, the lock, as represented in the semantics, *is free still!* Even if the interface step

is understood as atomically taking the lock, it is unobservable when it frees it again, so in absence of an environment and in absence of interference, it's possible that the lock is free again next time the component does a step, so the semantics might as well not take the lock at all. In other words: whether or not the environment is in possession of the lock or not is unobservable for the component. See also the discussion in Remark 10.

The two CallI-rules are dual to CallO and deal with incoming calls. As objects created by the environment are instantiated at the component only when they are called for the first time, we distinguish two situations: the object half is not yet instantiated or it is already (rules CallI_1 and CallI_2). In the first case, the new object needs to be instantiated, using the *members*-function analogously to the instantiation in rule NewOO to embed the members of the object. After the instantiation, the lock is taken, since the communication step corresponds to the point in time where the method actually starts executing. In case of CallI_2, the callee object is already present in the component. The same is done for all object reference arguments from the actual parameters $\vec{v}$; we simply write $C(\vec{v})$ do denote the corresponding newly instantiated object-halves (cf. also Remark 6). To be able to accept the incoming call, the lock must be free before the step, and is it taken afterwards. Again, by writing $M.l(o)(\vec{v})$ we mean especially, that the methods M of the callee o actually contain the method labelled l and hence it is an incoming call from the environment to the component. In both CallI rules, the well-typedness and connectivity is checked in the premises, and the contexts updated appropriately.

The ClaimI- and GetI-rules all deal with the component receiving the result of a method call by referencing the corresponding future reference, p' in the rules. Remember that there are two constructs with which to obtain the return value of a method call: claim and get. Both have the same "functional behavior" but behave differently as far as the lock-handling is concerned (cf. also the rules of the internal semantics of Table 6). That means that the checks for well-formedness, typing, and connectivity coincide for both kinds of interactions. The same applies for the context *updates*. When claiming a future, there are two possible reactions of the thread executing the claim: either the claim is immediately successful (in rule ClaimI_1) and the value is imported, or the future is not yet evaluated in which case claiming thread releases the lock temporarily in an internal step (cf. rule ClaimI_2). In both cases, the future is located in the environment, as requested by $\Delta \vdash p'$; in case of ClaimI_1, that is part of the check $\Xi \vdash a$. An outgoing get-communication in rule GetO simply updates the contexts and removes the consumed future from the component.

Remark 6 (Incoming object references as arguments). In the CallI-rules but also for incoming arguments via claim or get, objects whose references are transmitted as *arguments* of the communications are *instantiated* in case they are not already. More precisely, the *component half* of the object is instantiated at that point. Looking specifically at rule CallI_1, which captures the situation where the *callee* object is not yet instantiated, it is clear that as an effect of that rule, the callee is freshly instantiated. Whether also the objects transmitted as

$$\frac{a = \nu(o{:}c)_{o'}! \qquad E_\Theta \vdash p \hookrightarrow o' \qquad \Delta, \Theta; \Gamma_\Theta \vdash \mathit{members}(c) = M, F \qquad \acute{\Xi} = \Xi + a}{\Xi \vdash C \parallel p\langle \mathsf{let}\, x{:}c' = \mathsf{new}\, c\, \mathsf{in}\, t\rangle \xrightarrow{a} \acute{\Xi} \vdash C \parallel p\langle \mathsf{let}\, x{:}c' = o\, \mathsf{in}\, t\rangle \parallel o[c, M, F, \bot]} \;\textsc{NewOO}$$

$$\frac{a = \nu(o{:}c)_{o'}? \quad \Xi \vdash a \quad \acute{\Xi} = \Xi + a}{\Xi \vdash C \xrightarrow{a} \acute{\Xi} \vdash C} \;\textsc{NewOI} \qquad \frac{a = \nu(p{:}\langle T\rangle)_{o'}? \quad \Xi \vdash a \quad \acute{\Xi} = \Xi + a}{\Xi \vdash C \xrightarrow{a} \acute{\Xi} \vdash C} \;\textsc{NewTI}$$

$$\frac{a = \nu(p{:}\langle T\rangle)_{o'}! \qquad E_\Theta \vdash p' \hookrightarrow o' \qquad \acute{\Xi} = \Xi + a}{\Xi \vdash C \parallel p'\langle \mathsf{let}\, x{:}\langle T\rangle = o@l(\vec{v})\ \ \mathsf{in}\, t\rangle \xrightarrow{a} \acute{\Xi} \vdash C \parallel p'\langle \mathsf{let}\, x{:}\langle T\rangle = p\ \ \mathsf{in}\, t\rangle \parallel p\langle \mathit{call}\ o.l(\vec{v})\rangle!} \;\textsc{NewTO}$$

$$\frac{a = p\langle \mathit{call}\ o.l(\vec{v})\rangle! \qquad C = C' \parallel o[c, M, F, \bot] \qquad M.l = \bot \qquad \acute{\Xi} = \Xi + a}{\Xi \vdash C \parallel a \xrightarrow{a} \acute{\Xi} \vdash C} \;\textsc{CallO}$$

$$\frac{a = p\langle \mathit{call}\ o.l(\vec{v})\rangle? \quad \Xi \vdash a \quad \acute{\Xi} = \Xi + a \quad \Theta \not\vdash o \quad \Delta \vdash o{:}c \quad \Delta, \Theta; \Gamma_\Theta \vdash \mathit{members}(c) = M, F}{\Xi \vdash C \xrightarrow{a} \acute{\Xi} \vdash C \parallel p\langle \mathsf{let}\, x{:}T = M.l(o)(\vec{v})\ \mathsf{in}\ \mathsf{release}(o); x\rangle \parallel o[c, M, F, \top] \parallel C(\vec{v})} \;\textsc{CallI}_1$$

$$\frac{a = p\langle \mathit{call}\ o.l(\vec{v})\rangle? \qquad \Xi \vdash a \qquad \acute{\Xi} = \Xi + a}{\Xi \vdash C \parallel o[c, M, F, \bot] \xrightarrow{a} \acute{\Xi} \vdash C \parallel p\langle \mathsf{let}\, x{:}T = M.l(o)(\vec{v})\ \mathsf{in}\ \mathsf{release}(o); x\rangle \parallel o[c, M, F, \top] \parallel C(\vec{v})} \;\textsc{CallI}_2$$

$$\frac{a = p'\langle \mathit{get}(v)\rangle? \qquad \Xi \vdash a \qquad \acute{\Xi} = \Xi + a}{\Xi \vdash C \parallel p\langle \mathsf{let}\, x{:}T = \mathsf{claim}@(p', _)\ \mathsf{in}\ t\rangle \xrightarrow{a} \acute{\Xi} \vdash C \parallel p\langle \mathsf{let}\, x{:}T = v\ \mathsf{in}\ t\rangle \parallel C(v)} \;\textsc{ClaimI}_1$$

$$\frac{\Delta \vdash p'}{\Xi \vdash C \parallel p\langle \mathsf{let}\, x{:}T = \mathsf{claim}@(p', o)\ \mathsf{in}\ t\rangle \rightsquigarrow \Xi \vdash C \parallel p\langle \mathsf{release}(o); \mathsf{let}\, x{:}T = \mathsf{get}@p'\ \mathsf{in}\ \mathsf{grab}(o); t\rangle \parallel C(v)} \;\textsc{ClaimI}_2$$

$$\frac{a = p'\langle \mathit{get}(v)\rangle? \qquad \Xi \vdash a \qquad \acute{\Xi} = \Xi + a}{\Xi \vdash C \parallel p\langle \mathsf{let}\, x{:}T = \mathsf{get}@p'\ \mathsf{in}\ t\rangle \xrightarrow{a} \acute{\Xi} \vdash C \parallel p\langle \mathsf{let}\, x{:}T = v\ \mathsf{in}\ t\rangle \parallel C(v)} \;\textsc{GetI}$$

$$\frac{a = p\langle \mathit{get}(v)\rangle! \qquad \acute{\Xi} = \Xi + a}{\Xi \vdash C \parallel p\langle v\rangle \xrightarrow{a} \acute{\Xi} \vdash C} \;\textsc{GetO}$$

Table 13. External steps

argument should be instantiated at that point is not so clear. We decided they should be instantiated basically to make the treatment *symmetric.*

In more detail: Remember that the object halves are instantiated not at the same time: The half at the side of the creator is instantiated *immediately,* the other side only when the instance (not just the reference) is actually "needed". The reason why we make that distinction is to characterize when an object (which "in reality" of course is instantiated always fully, not half-way) cannot be already active on one side, i.e., it cannot be the source of a method call. This "activation" occurs only at the time when it is the callee of a method call. That alone would be an argument that being mentioned as parameter in a communication does *not* instantiate the object half, i.e., an argument for *not* writing $C(\vec{v})$ in the corresponding rules for incoming communication.

However, for the situation of outgoing communication, we need to formalize *assumptions* about whether or not the environment half is active or not. More precisely we must characterize whether it is *possible* that the environment half of the object is active or not. Being transmitted as argument, in that situation, makes it indeed *possible* that the object is active, which means that we extend the assumption Δ accordingly by adding the object reference to the domain of Δ. Therefore, to keep it symmetric, we extend also for an incoming communication the commitment by the information that the object half is instantiated (and thus, since the commitment reflects what is going on in the component, we instantiate that half in the same step). □

Remark 7 (Communication of fresh identifiers). To communicate freshly created identities, we use the ν-binder as known from the π-calculus expressing a new, fresh identity with dynamic scope, which is communicated across the interface by the rules NewOO and NewTO, resp. the dual counterparts. The identity of a freshly created object or thread is immediately communicated to the environment. An alternative would be to publish new identities non-deterministically at an arbitrary point, or at the latest possible, when (and if) they are actually transmitted as argument. Instead of having a separate communication label $\nu(n{:}T)$ in Table 9 (we omit the information about the creator of the fresh identifier in this informal discussion), the labels would then be of the form

$$\nu(n_1{:}T_1, \ldots n_l{:}T_k).\gamma \ ,$$

where all fresh identities mentioned in γ are mentioned it the binding prefix $\nu(\vec{n}{:}\vec{T})$. This representation has been used in our previous work, for instance in [21,4]. In this paper, the semantics records the creation of a new object, resp. of a new thread/future identifier immediately at the point when it is created. This makes the individual rules slightly simpler (at the price of extra rules to deal with the ν-part of the labels). □

Remark 8 (Interface information). The interface information, as far as typing is concerned, is kept in Δ and Θ and contains the names of the (publicly available) interface types, i.e., their signature. Furthermore, the class hierarchy is part of the interface information, i.e., which class extends which one. A final piece of information relevant at the interface is not only to mention the available methods, but also, whether a method needs actually to be implemented by the class, or whether it's inherited from a super-class.

The last piece of information is typically not part of an interface description; interfaces in *Java*, for instance, do not specify that. There is a good reason why it's included in our representation, namely: whether or not a class overrides a method or inherits it is *observable* from the outside. □

Remark 9 (Object instantiation). As mentioned, the representation of an object in the open semantics is *split* into two halves, one half resides at the component and the other half is part of the environment (only the *lock* is conceptually shared). The lock of a newly instantiated object is *free,* as given by the internal

semantics. The locks are *binary* and have the two possible values $\bot$ and $\top$. In the open semantics, only the half of the object implemented by component classes is represented in the semantics, the half belonging to the environment is missing, or more precisely, the environment part is represented abstractly by the assumption contexts, only. In the open semantics, the state of the lock does *not* represent the actual value of the object lock, only the state of the lock *from the perspective* of the component. So the two states $\bot$ and $\top$ are understood as follows. $\top$ means that the lock is taken *by the component,* which implies that it is *not* taken by the environment. Conversely, $\bot$ means the lock is not taken by the component, but it may or may not be taken by the environment. □

Remark 10 (Lock). The state of an object, i.e., its fields, are represented in the open semantics *split*: only the fields pertaining the component are represented, those of the environment are not. The lock can be seen as part of the instance state, but it does not belong exclusively to one of the two sides, it is *shared.* In a configuration of the open semantics, each object represented therefore contains "one half" of its lock, interpreted as follows. A lock taken $\top$ represents the situation that a *component thread* is in possession of the lock. A free lock $\bot$ means the opposite: no component thread currently holds the lock. This, however, does not represents information about the status of the lock as far as the environment is concerned. A lock status $\bot$ means that the environment may or may not currently hold the lock. Due to the asynchronous nature of communication and (related to that) due to absence of re-entrant threading, a lock status of $\bot$ from the perspective of the component has *no* implications about whether the environment holds the lock or not. Even if the component has issued a call to an environment method, which during execution holds the lock, the component does not know whether the execution has not yet started, is under way, or is already finished. The latter case, that a particular method that has been called by the component and executed by the environment has finished can be "observed" by the fact that the methods return value is available. But then again, the way to "observe" that is via claim or get, which do not allow to observe the negative fact that the value is *not yet* there and that consequently the particular method has not yet given back that lock. And after the value is available, it's unobservable from the perspective of the component, whether or not another thread has taken the lock. In summary: if, from the perspective of the component, the lock is free, the component can never be sure about the lock-status as far as the environment is concerned. In that sense, the component and the environment are decoupled. In a Java-like setting with synchronous method calls and re-entrant monitors, this is not the case and complicates matters considerably (cf. [6], which deals with re-entrant monitor behavior). □

Remark 11 (Concurrency model). The results of this paper are formulated for a concurrent, object-oriented language based on active objects and asynchronous method calls. The concurrency model is thus different from the concurrency model based on *multi-threading* used in languages as Java and C$^\sharp$. As far as the inheritance is concerned, the situation in our calculus resembles closely to

the one in those mentioned languages, representing the mainstream of object-oriented languages: late-bound methods and a single-inheritance class-hierarchy.

This means that in principle the results of this work apply to a multi-threaded setting, as well, namely that inheritance makes self-calls observable, and that approximation of the heap structure is relevant interface information. Concerning the details, using a language based on multi-threading, *re-entrant* monitors, and inheritance, would considerably complicate the interface behavior.

One reason is that the presence of the `synchronized` keyword as in Java complicates the setting in at least one of the following two ways, depending on which decision is taken wrt. whether being synchronized or not is public interface information.

If the question of being `synchronized` is part of the interface information of a method, the interaction trace reveals in many cases information, that the re-entrant lock of a given object is definitely taken, and that information must be taken into account. In our setting here, the information that a lock is taken is *not* part of the interface information which simplifies the treatment considerably. The consequences of multi-threading with re-entrant locks are explored in [5], but without inheritance. If, alternatively, the decision is taken that `synchronized` is not part of the interface information, a synchronized method does not really provide protection against interference, especially if an unsynchronized method is inherited.

We consider these (considerable) complications as a serious counter-argument against the multi-threading concurrency model. □

4 Interface behavior

Next we characterize the possible ("legal") *interface behavior* as interaction traces between component and environment. Half of the work has been done already in the definition of the external steps in Table 13: For incoming communication, for which the environment is responsible, the assumption contexts are consulted to check whether the communication originates from a realizable environment. Concerning the reaction of the component, no such checks were necessary. To characterize when a given trace is *legal,* the behavior of the component side, i.e., the outgoing communication, must adhere to the dual discipline we imposed on the environment for the open semantics. This means, we analogously abstract away from the program code, rendering the situation symmetric.

4.1 Legal traces system

The rules of Table 14 specify legality of traces. We use the same conventions and notations as for the operational semantics (cf. Notation 1). The judgments in the derivation system are of the form

$$\Xi \vdash s : \mathit{trace} \ . \tag{18}$$

$\Xi \vdash \epsilon : trace$ L-Empty

$$\frac{a = \nu(n{:}c)_{o'}? \quad \Xi \vdash a \quad \acute{\Xi} = \Xi + a \quad \acute{\Xi} \vdash s : trace}{\Xi \vdash a\ s : trace} \text{ L-NewI}$$

$$\frac{a = p\langle call\ o.l(\vec{v})\rangle? \quad \Xi \vdash a \quad \acute{\Xi} = \Xi + a \quad \acute{\Xi} \vdash s : trace}{\Xi \vdash a\ s : trace} \text{ L-CallI}$$

$$\frac{a = p'\langle get(v)\rangle? \quad \Xi \vdash a \quad \acute{\Xi} = \Xi + a \quad \acute{\Xi} \vdash s : trace}{\Xi \vdash a\ s : trace} \text{ L-GetI}$$

Table 14. Legal traces (dual rules omitted)

We write $\Xi \vdash s : trace$, if there exists a derivation according to the rules of Table 14 with an instance of L-Empty as axiom. The empty trace is always legal (cf. rule L-Empty), and distinguishing according to the first action a of the trace, the rules check whether a is possible. Furthermore, the contexts are updated appropriately, and the rules recur checking the tail of the trace. With the connectivity contexts E_Δ and E_Θ as part of the judgment, we must still clarify what it "means", i.e., when does $\Xi \vdash C : ok$ hold? Besides the typing part, this concerns the commitment part E_Θ. The relation E_Θ asserts about the component C that the connectivity of (mainly) the objects halves from the component is *not larger than* the connectivity entailed by E_Θ, i.e., E_Θ is a conservative over-approximation of the component connectivity. Given a component C and two names o from Θ and n from Θ, Δ, we write $C \vdash o \hookrightarrow n$, if $C \equiv C' \parallel o[\ldots, f = n, \ldots]$, i.e., o contains in one of its fields a reference to n. Furthermore, for a thread name p in Θ, we write $C \vdash p \hookrightarrow o$, if either $C \equiv C' \parallel p\langle \ldots \mathsf{release}(o); v\rangle$ or $p\langle v\rangle$. We can thus define:

Definition 11. *The judgment $\Xi \vdash C$ holds, if*

1. $\Delta \vdash C : \Theta$ *(well-typedness)*
2. *Connectivity:*
   (a) $C \vdash o_1 \hookrightarrow o_2$ *implies* $E_\Theta \vdash o_1 \leftrightharpoons o_2$
   (b) $C \vdash o \hookrightarrow p$ *implies* $E_\Theta \vdash o \hookrightarrow p$.
   (c) $C \vdash p \hookrightarrow o$ *implies* $E_\Theta \vdash p \hookrightarrow o$.

We simply write $\Xi \vdash C$ to assert that the judgment is satisfied. Note that references mentioned in threads do not "count" as acquaintance.

We need to show that the behavioral description of Table 14, actually does what it claims to do, to characterize the possible interface behavior. We show the soundness of this abstraction plus the necessary ancillary lemmas such as subject reduction. Subject reduction means, preservation of well-typedness under reduction.

Lemma 1 (Subject reduction). *Assume $\Xi \vdash C$*

1. (a) *If* $C \rightsquigarrow C'$, *then* $\Xi \vdash C$.
   (b) *If* $C \xrightarrow{\tau} C'$, *then* $\Xi \vdash C$.
   (c) *If* $C \equiv C'$, *then* $\Xi \vdash C$.
2. *If* $\Xi \vdash C \xrightarrow{a} \acute{\Xi} \vdash \acute{C}$, *then* $\acute{\Xi} \vdash \acute{C}$.

Proof. Straightforward. □

Lemma 2 (Subject reduction). $\Xi \vdash C$ *and* $\Xi \vdash C \stackrel{s}{\Longrightarrow} \acute{\Xi} \vdash \acute{C}$ *imply* $\acute{\Xi} \vdash \acute{C}$.

Proof. A consequence of preservation under single steps (Lemma 1). □

An interesting invariant concerns the connectivity of names transmitted boundedly. Incoming communication, e.g., not only updates the commitment contexts —something one would expect— but also the *assumption* contexts. The fact that no new information is learnt about already known objects ("no surprise") in the assumptions can be phrased using the notion of conservative extension.

Definition 12 (Conservative extension). *Given two contexts* Ξ_Δ *and* $\acute{\Xi}_\Delta$ *where* $\acute{\Delta}$ *is an extension of* Δ. *Then we write* $\Xi_\Delta \vdash \acute{\Xi}_\Delta$ *if* $\acute{\Xi}_\Delta \vdash n_1 \leftrightharpoons n_2$ *implies* $\Xi_\Delta \vdash n_1 \leftrightharpoons n_2$, *for all* n_1, n_2 *with* $\Delta \vdash n_1, n_2$.

Lemma 3 (No surprise). *Let* $\Xi \vdash C \xrightarrow{a} \acute{\Xi} \vdash \acute{C}$ *for some incoming label* a. *Then* $\Xi \vdash \acute{\Xi}$. *For outgoing steps, the situation is dual.*

Proof. By definition of the incoming steps from Table 13, using the context update from Definition 7 and 9. □

Lemma 4 (Soundness). *If* $\Xi_0 \vdash C$ *and* $\Xi_0 \vdash C \stackrel{s}{\Longrightarrow}$, *then* $\Xi_0 \vdash s : trace$.

Proof. By induction on the number of steps in $\stackrel{s}{\Longrightarrow}$. The base case of zero steps (which implies $s = \epsilon$) is immediate, using L-Empty. The induction for *internal* steps of the form $\Xi \vdash C \Longrightarrow \Xi \vdash \acute{C}$ follow by subject reduction for internal steps from Lemma 2; in particular, internal steps do not change the context Ξ. Remain the external steps of Table 13. First note the contexts Ξ are *updated* by each external step to $\acute{\Xi}$ the same way as the contexts are updated in the legal trace system.

The cases for *incoming* communication are checked straightforwardly, as the operational rules check incoming communication already, i.e., the premises of the operational rules have their counterparts in the rules for legal traces.

Case: NewOI
Immediate, as the premises of L-NewI coincide with the ones of NewOI; note that the name n included object names o. The case for NewTI works analogously.

Case: CALLI_1 and CALLI_2
Both cases are covered immediately by L-CALLI. The cases for incoming get labels are likewise immediate.

The cases for outgoing communication are slightly more complex, as the label in the operational rule is not type-checked or checked for well-connectedness as for incoming communication and as is done in the rules for legality. For all cases of outgoing communication we need therefore to check that the condition $\Xi \vdash a$, stating that the (legal) trace can be extended by label a is actually satisfied. We concentrate in the argument on the connectivity part, as the typing part is checked straightforwardly. Cf. Table 12.

Case: NEWOO with $a = \nu(o{:}c)_{o'}!$
The connectivity part of $\Xi \vdash a$ for a ν-label is empty. Concerning typing: As for LT-NEWO of Table 11, the premise $\Theta \vdash o'$ follows from the premise $E_\Theta \vdash p \hookrightarrow o'$.

Case: CALLO with $a = p\langle call\ o.l(\vec{v})\rangle!$
The open semantics specifies, that a CALLO-step (sending the call message) must be preceded by a NEWTO-step, which creates the new future/thread reference, p in this case. The premise of NEWTO implies $\tilde{E_\Theta} \vdash p' \hookrightarrow o'$ (where $\tilde{E}_\Theta$ is the connectivity context before that step, o' is the creating object and p' the spawning thread). Furthermore, the update premise $\tilde{\Xi}' = \tilde{\Xi} + \nu(p{:}\langle\tilde{T}\rangle)_{o'}!$ of the NEWTO-step implies for the connectivity after that step: $\tilde{E}'_\Theta \vdash o' \hookrightarrow p$. Since no information is ever forgotten, also $E_\Theta \vdash o' \hookrightarrow p$ and $\Theta \vdash o'{:}c$. Finally, $E_\Theta \vdash o' \hookrightarrow o, \vec{v}$, since we have $\Xi \vdash C$ before the step (by subject reduction), i.e., E_Θ is a sound over-approximation of the connectivity of C.

The remaining cases work similarly. □

5 Conclusion

This paper investigates in a formal manner the interface behavior of a (concurrent) object-oriented languages with class *inheritance.* The interface behavior is characterized in the form of a typed operational semantics of an open system, consisting of a set of classes. The semantics is formalized in the form of commitments of the component and in particular *assumptions* about the environment. The fact that the components are open wrt. inheritance, i.e., a component can inherit from the environment and vice versa, has as a consequence that the assumptions and commitments need contain an abstraction of the heap topology, keeping track of which object may be in connection with other objects. We show the soundness of the abstractions.

Related work Banerjee and Naumann [7] are concerned with observable equivalence of classes resp. objects and substitutability in a setting of a class-based, object-oriented language with inheritance. Whereas in our approach, where objects are inherently concurrent, they are focusing on the "data" aspect of object-oriented languages, i.e., they are interested in whether two class-based implementations of some data structure are indistinguishable by any observer or context.

To capture observable equivalence they use the well-known notion of *representation independence* [10] (cf. also [15] [9] [17] [18]). It is a formal definition of when the representation of a data type does not influence the rest of the program and thus it is a contextual characterization of *encapsulation.* Technically, representation independence is defined as follows: the internal states of the two data types are related by a simulation relation, called local coupling relation in [7], and the two implementations are representation independent if the two locally coupled internal representations do not lead to an observable difference in the global system, which is formalized by stating that the two global systems are connected by a *global* (or induced) coupling relation. While in our setting we aim for a *behavioral* interface description ensuring substitutability, representation independence, e.g. in [7], defines criteria on the *internal representation* of a data type to assure that two "components" with the same *static* interface (the method signature) have the same "dynamic" interface behavior. Those criteria boil down to the following: Encapsulation or confinement of the representation assures representation independence and thus observable equivalence. Encapsulation is ensured statically in [7] by ownership restrictions. In contrast, our behavioral interface description takes a "black-box" view and considers two system to be equivalent, if they exhibit the same traces at the interface. Also [16] uses the notion of representation independence as a criterion of what is a good description of an interface behavior. In the tradition of Featherweight Java and related proposals, the language they study is an object-oriented calculus similar to the one we use with mainly two differences: their language is *sequential* (and thus deterministic) and they don't use an unstructured heap. Instead, inspired by ownership concepts, the heap is hierarchically structured into nested "boxes". Each object belongs to exactly one (directly surrounding) box. Important for the question of interface behavior is that the boxes form one basis for their notion of run-time *component.* Statically and as in our framework, a component consists of a set of classes. There is, however, an important restriction in [16]: to form a component, the corresponding set of classes must be "closed" in that all classes, methods, etc. *used* in code of the component are actually *defined* in the component itself (which is "declaration complete" in the terminology of [16]; in our notation, the component C is defined with *an empty assumption context*, i.e., $\bullet \vdash C : \Theta$). Hence a component cannot instantiate classes of the environment nor can it *inherit* from environment classes. Note that dually the environment needs not to be declaration complete: The environment can mention component classes and methods, but not vice versa. Conceptually, one can think of a definition complete component as a form of *library*, where the program can refer to the library, but not vice versa. Technically that restriction implies that when describing the possible interface behavior of a component, *connectivity* is irrelevant, as the component can neither instantiate classes outside the component nor can it inherit methods from outside. In our setting, a component is *not* definition complete. However, the environment is represented abstractly as *assumption* (and the component announces its classes and methods in the form of commitments), i.e., the assumption-commitment formulation allows to avoid the

(severe) restriction requiring declaration completeness. Similarly as in our work, [16] needs to characterize allowed interactions at the interface of the component or box, in their case to be able to define properly their "behavior semantics" and representation independence. This involves answering the question that given a trace (called history in [16]), what is the reaction of the component. Such a reaction is defined only when the history is actually well-formed, which basically corresponds conceptually to our formalization of legal traces. Again, however, connectivity does not play a role due to their restrictions. Similarly, in the context of observable equivalence and a fully abstract semantics based on interface traces, [12] and [11] do not need to consider connectivity: in [11], because the language is object-*based*, i.e., without classes at all.; [12], in contrast, avoids considering connectivity by introducing packages as units of composition, which, in the terminology of [16] are definition complete. Also [22] consider an object-based setting. In absence of class inheritance and method overriding, object-based languages (or proto-type based languages) typically support method update, i.e., the replacement of methods at run-time. Apart from the technical results in the paper, which is not a trace based formulation of the semantics but the observable equivalence between an object-oriented program and its translation into a lower level representation (translational full abstraction), their results show that self-calls become observable when considering late-binding and method update. This is similar to the observable semantics here which shows that with late-binding and method overriding, self-calls must be considered in the interface behavior. Compared to our setting, the calculus is simpler in that it does not have pointers at all (hence the question of connectivity does not arise in the first place). Neither do they consider concurrency. The enhanced "distinguishing power" when adding inheritance is also relevant proof-theoretically, i.e., when trying to verify object-oriented programs and design proof systems for that. [13] develop a proof technique based on bisimulations to capture contextual equivalence for a class-based language (without and with inheritance).

The results here extends previous work namely by considering inheritance. Earlier we considered the problem of characterizing the interface behavior of an open system for different choices of language features (but without inheritance). E.g., [4] deals with futures and promises, i.e., using a similar concurrency model than the one here. One of the challenges there was to capture the influence of promises by a "resource aware" type and effect system as promises can be "fulfilled", i.e., bound to code, only once. [5] investigates the influence of locks and monitors on the interface behavior. Again, the results reported therein are rather similar as far as the goals and general setting is concerned. Unlike here, the calculus is inspired by Java's model of concurrency, i.e., based on multi-threading and re-entrant locks, whereas here we are basing our study on active objects. The seemingly innocent change of the communication and synchronization model (from rpc or remote method call communication to *asynchronous* method calls, from re-entrant locks to binary locks) leads to a quite more complicated interface behavior for Java-like monitors. Ultimately, the reason for that complication can be attributed to the more tighter coupling of objects in the multi-threaded

setting. Object-connectivity for the interface behavior was first investigated in [3] [2], as a consequence of cross-border instantiation, but not inheritance (see also [21]).

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. Object connectivity and full abstraction for a concurrent calculus of classes. In *ICTAC'04*, volume 3407 of *LNCS*. Springer, July 2004.
3. E. Ábrahám, F. S. de Boer, M. M. Bonsangue, A. Grüner, and M. Steffen. Observability, connectivity, and replay in a sequential calculus of classes. In *FMCO'04*, volume 3657 of *LNCS*. Springer, 2005.
4. E. Ábrahám, I. Grabe, A. Grüner, and M. Steffen. Behavioral interface description of an object-oriented language with futures and promises. *JLAP*, 78(7), 2009.
5. E. Ábrahám, A. Grüner, and M. Steffen. Abstract interface behavior of object-oriented languages with monitors. In *FMOODS '06*, volume 4037 of *LNCS*. Springer, 2006.
6. E. Ábrahám, A. Grüner, and M. Steffen. Abstract interface behavior of object-oriented languages with monitors. *Theory of Computing Systems*, 43(3-4), Dec. 2008.
7. A. Banerjee and D. A. Naumann. Ownership confinement ensures representation independence for object-oriented programs. *Journal of the ACM*, 52(6), 2005.
8. F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In *ESOP'07*, volume 4421 of *LNCS*. Springer, 2007.
9. J. Donahue. On the semantics of data type. *SIAM J. Computing*, 8, 1979.
10. C. T. Haynes. A theory of data type representation independence. In *Semantics of Data Types*, volume 173 of *LNCS*. Springer, 1984.
11. 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.
12. A. Jeffrey and J. Rathke. Java Jr.: A fully abstract trace semantics for a core Java language. In *Proceedings of ESOP 2005*, volume 3444 of *LNCS*. Springer, 2005.
13. V. Koutavas and M. Wand. Reasoning about class behavior. In *Informal Workshop Record of FOOL 2007*, Jan. 2007.
14. L. Mikhajlov and E. Sekerinski. A study of the fragile base class problem. In *ECOOP'98*, volume 1445 of *LNCS*. Springer, 1998.
15. J. C. Mitchell. Representation independence and data abstraction. In *Thirteenth POPL (St. Peterburg Beach, FL)*. ACM, January 1986.
16. A. Poetzsch-Heffter and J. Schäfer. A representation-independent behavioral semantics for object-oriented components. In *FMOODS '07*, volume 4468 of *LNCS*. Springer, June 2007.
17. J. Reynolds. Towards a theory of type structure. In *Colloque sur la programmation (Paris, France)*, volume 19 of *LNCS*. Springer, 1974.
18. J. Reynolds. Types, abstraction, and parametric polymorphism. In *Information Processing 83*. IFIP, North-Holland, 1983.
19. A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *OOPSLA '86*. ACM, 1986. In *SIGPLAN Notices* 21(11).

20. R. Stata and J. V. Guttag. Modular reasoning in the presence of subclassing. In *OOPSLA '95*. ACM, 1995. In *SIGPLAN Notices*) 30(10).
21. M. Steffen. *Object-Connectivity and Observability for Class-Based, Object-Oriented Languages*. Habilitation thesis, Technische Faktultät der Christian-Albrechts-Universität zu Kiel, July 2006.
22. R. Viswanathan. Full abstraction for first-order objects with recursive types and subtyping. In *Proceedings of LICS '98*. IEEE, Computer Society Press, July 1998.