Under consideration for publication in Math. Struct. in Comp. Science

Observable interface behavior and inheritance

Erika Ábrahám and Thi Mai Thuong Tran and Martin Steffen

Received 29 December 2013

This paper formalizes the observable interface behavior of *open* systems for a strongly-typed, concurrent object-oriented language with single-class inheritance. We formally characterize the observable behavior in terms of interactions at the program-environment interface. The behavior is given by transitions between contextual judgments, where the absent environment is represented abstractly as assumption context. A particular challenge is the fact that, when the system is considered as open, code from the environment can be inherited to the component and vice versa. This requires to incorporate an abstract version of the heap into the environment assumptions when characterizing the interface behavior. We prove the soundness of the abstract interface description.
Keywords: inheritance, object-oriented languages, formal semantics, concurrency, open systems, observable behavior

1. Introduction

A *component* is a part of a larger system, which interacts with its environment, and can be considered as a black box whose 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.

In this paper we consider components as sets of classes with an inheritance structure. As open and being part of a overall program, they they cannot execute on their own, but only in interaction with their environment. They are furthermore open in that component classes may inherit from classes specified in the environment and conversely also the environment may extend component classes using inheritance.

The openness of a system in the presence of inheritance and late binding is problematic. With a standard behavioral interface specification given as pre- and post-conditions for the available methods, replacing one super or base class by another satisfying the same interface description, may break the code of the client of the super class, i.e., change the behavior of the "environment" of the super class. Consider the following code fragment.

Listing 1: Fragile base class

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

The two methods *add* and add_2 are intended to add one resp. 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. A sub-class could *observe* a difference, namely by overriding the auxiliary method which is invoked by the inherited method in one situation but not in the other. A similar phenomenon is also described as fragile base class problem (Mikhajlov and Sekerinski [1998], Stata and Guttag [1995], Snyder [1986], Ruby and Leavens [2000]). *Nothing*, however, in the interface distinguishes the two different super-classes: The interface specification is too weak to allow to consider the super-class as a black box which can be safely substituted based on its interface specification only, i.e., ignoring this phenomenom results in a non-compositional semantical description.

The challenge therefore is to give a formal, behavioral interface description which *matches* what can be observed by client code in the presence of inheritance and late-binding. A basic soundness requirement (also known as adequacy) is that two open systems with the same interface desciption should be safely exchangable for each other without leading to different outcomes when used in arbitrary contexts. Therefore, ignoring observable details (such as self-calls) in the interface description would lead to an *unsound* semantical description. Soundness, however, is not the only desirable property when characterizing the open semantics. The semantics should not contain unnecessary, unobservable details and in particular the semantics should only include *possible* behaviors, i.e., behaviors generated by *some* actual (well-formed, well-typed) program in the given language. This is related to the notion of definability in fully abstract semantics (cf. e.g. Curien [2007] for a discussion of definability) where typically the hard part of achieving full abstraction is to design a semantic domain where each element in the semantics of an actual (open) program fragment, i.e., the semantic function should be surjective such that the semantic domain only contains *definable* elements. For illustration: the classical example of course is domain theory for functional languages, where programs are not interpreted over sets of arbitrary functions, but restricted to *continuous* functions (in appropriately defined cpo's) as those are the only functions which are definable, i.e., computable. In languages with mutable state, as here, the observable behavior takes the form of a set of interaction sequences or traces, here consisting of method invocations and the eventual returning of their results, and we are interested in capturing the interface traces as precise as possible. A rigorous account of such an interface behavior is important also for formal, compositional verification of open programs. In settings with more complex forms of program composition than plain sequential composition (in particular

in the presence of concurrency but also for object-oriented languages), a key ingredient to obtain a modular Hoare-style reasoning system is to record the interacting behavior appropritately in a logical history variable. Capturing the potentially possible histories not precisely then leads to *incomplete* reasoning for verifying an open component indepedent from its environment. Not realizing that no environment exists which able to engage in a given history or trace, the proof method would work with weaker assumptions than otherwise possible, potentially unable to prove assertions which actually do hold in all concrete programs. A precise account of the open semantics is also beneficial 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, and being able to ignore traces which cannot actually occur makes programs the observable equivalence more coarse-grained, thus potentially allowing more optimizations.

This paper formalizes an open semantics for a statically typed object-oriented calculus featuring concurrency, dynamic object creation, mutable heap, and single inheritance. The behavioral interface description is phrased in a typed assumption-commitment framework. The setting allows that component classes to inherit from environment classes and vice versa. Thus, the account really captures the observable, behavioral aspects of class inheritance without restrictions, for instance by allowing inheritance only *within* the component. A consequence of that set-up is that a precise charaterization of the open semantics and of the legal traces needs to take an abstraction of the heap into account. We prove the soundness of the abstractions. The results here extend previous work with inheritance, which is a central feature for object-oriented languages. Earlier we considered the problem of open systems for different choices of language features (but without inheritance), e.g., for futures and promises (Ábrahám et al. [2009]), for Java-like monitors (Ábrahám et al. [2006]). Object-connectivity plays a crucial role in the current work (as in Ábrahám et al. [2005]) but is here a semantical consequence of inheritance. Including inheritance influences in subtle ways what is observable, e.g., the observer may override component methods or inherit its own methods to the component which then are rebound by late binding. Capturing the resulting interface behavior accurately complicates the semantics considerably. As mentioned, we consider a concurrent, object-oriented calculus, and the model of concurrency used here is based on active objects using asynchronous method calls and futures. It thus resembles the communication mechanisms of loosely coupled interacting objects known from Actor-based Agha and Hewitt [1987] languages such as Erlang Armstrong et al. [1996] and Scala Odersky et al. [2011]. We stress, however, that the particular choice of the concurrency model is, to a certain degree, orthogonal to the results; particular details concerning the exact format of the interface interaction of course depend on the details of the chosen model. But the core message of the paper, namely that capturing the influence of inheritance requires to take into account an abstract representation of the heap topology is independent from the chosen concurrency model.

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 and static type system of the calculus. The main contributions (the typed open semantics, the legal

traces, and the soundness of the abstractions) are presented in Sections 4 and 5. We conclude in Section 6 discussing related work.

2. Interface behavior, inheritance, and object connectivity

We start by giving more technical intuition to the challenges when defining an open semantics in our setting. E.g., as self-calls lead to observable differences in the presence of inheritance, they are part of the observable behavior. On the other hand, behavior which is impossible should not be included in the open semantics.

2.1. *Existential abstraction of the environment*

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 restricting ourselves to run-of-the-mill notion of single-inheritance between classes with sub-type 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 object-oriented pseudo-code for illustration. An interface interaction happens if a step of the component affects the environment and vice versa. Objects encapsulate their states, and thus the interaction takes the form of method calls and returns, where the control changes from executing component code to environment code (outgoing message) and vice versa (incoming message)[†]. Thus the interface behavior will be given in terms of traces of call and return labels exchanged at the interface, where in our setting component classes can extend those from the environment via inheritance, and vice versa. Writing $C \stackrel{t}{\Longrightarrow} \acute{C}$, the t denotes the *trace* of interface actions by which component C evolves into $\acute{C}$, potentially executing internal steps, as well, not recorded in t. Being open, C 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 $C \parallel E \stackrel{t}{\Longrightarrow}_{\bar{t}} \acute{C} \parallel \acute{E}$ 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. Our goal is an 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: 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 $\Xi_E \vdash C$. Reductions thus will look like[‡]

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

Such a characterization of the abstract interface behavior is relevant and useful for

[†] Note in passing: if the language allowed shared variables, an interface interaction would not necessarily mean that the *control flow* passes in the interface step from component to environment or vice versa.

[‡] 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.

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, since in this case one could not generate a corresponding tester. Finally, and not as the least gain, the formulation gives *insight* into the inherent semantical nature of the language, as the assumptions Ξ_E capture the existentially abstracted environment behavior.

Similarly to the representation of the environment by an assumption context in Equation (1), one can additionally abstract away from any concrete component C and replace it by a commitment context, obtaining a formalization of possible interface interaction in the language which we call *legal* traces. The following two sections explain two important (and technically challenging) consequences of inheritance and late-binding for the observable behavior: one showing that self-calls may be observable and thus need to be included in the traces and secondly that one needs an abstract existential over-approximation of the heap structure to avoid "illegal" traces. In the technical part afterwards, the semantics in the form of (1) is given in Section 4 and Section 5 presents possible interface behavior in the form of legal traces.

2.2. *Self-calls and cross-border inheritance*

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. Figure 1a illustrates the situation where an instance 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 outgoing call $o_E.m_E!$ which crosses the interface. In general, we use ! to denote outgoing communication from the perspective of the component and ? for incoming communication. Even if both caller and callee *objects* are instances of the component class C_C, the call from m_C to m_E still crosses the border, provided m_C is implemented in C_C and m_E is *inherited* from class C_E to C_C (cf. Figure 1b and Listing 2). Especially, if caller and callee are the same object, i.e., if m_C calls the (inherited) m_E via a *self-call,* it is still an interface interaction, as the code of m_E is given by the environment (Figure 1c).

Listing 2: Late binding

```
class C_E {                          class C_C extends C_E {
  ..                                     ...
  public void m_E () {...} ..            public void m_C () {...x.m_E...}
}                                        }
```

Likewise in the inverse situation in Listing 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

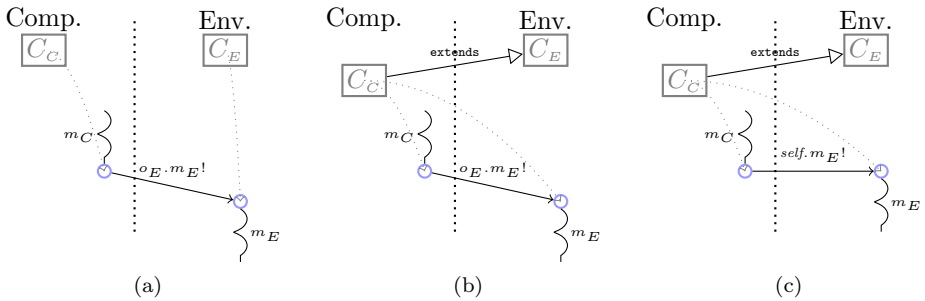

Fig. 1: Calls across the interface

call from the inherited m_1 to the overridden m_2 is also called a *downcall* (Ruby and Leavens [2000]) which can be seen as a special form of a call-back.

Listing 3: Overriding

```
class C_C {                                  class C_E extends C_C {
  ...                                            ...
  public void m1() {...self.m2 .. .}           public void m2() {...}
  public void m2() {...}                         ...
}                                              }
```

2.3. *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_C and C_E, with C_E a subclass of C_C, does it make a difference to have an instance of C_C or of C_E? Consider the following two expressions:

$$\mathsf{let}\, x{:}C_C = \mathsf{new}\, \mathbf{C_C}\, \mathsf{in}\, t \quad \text{and} \quad \mathsf{let}\, x{:}C_C = \mathsf{new}\, \mathbf{C_E}\, \mathsf{in}\, t\ . \tag{2}$$

In the first case, the dynamic type of the instance is C_C, in the second case it's the subclass/subtype C_E. Can one distinguish the two situations? If the super-class C_C is a component class and C_E is an observer class, the two situations of Equation (2) are distinguishable: by *overriding* a method of C_C in C_E, the behavior of instances of C_C differs from instances of C_E. An illustrative example is given by the Java-code in Listing 4, which shows the situation where an instance of the sub-class is created.[§]

[§] Since the observer class `Dynamictypeobs1` literally mentions `new C_C()` resp. `new C_E()`, one might 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 $\mathtt{C}_E$, but it would receive it as handed over from the environment, for instance as return value of a method call.

Listing 4: Dynamic type

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

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

class C_E extends C_C {
    void m () {System.out.print("C_E");};

}
```

Also in the inverse situation that C_E is component class and C_C a class of the environment, the two situations of Equation (2) are distinguishable.

2.4. *Connectivity as abstract heap representation in the interface*

Objects encapsulate their instance states such that fields of an object cannot be accessed from outside the instance, i.e. the field can be referenced only via the `this`-identifier (when following *Java*-like notation). This is slightly stronger than the restriction for `private` fields in Java, which allow access among instances of the same class. In particular, each method can access only the fields of the class that the method is *defined* in. In the presence of inheritance between component and environment, each object may contain fields defined by the component and fields defined by the environment. Since fields are private (per instance), 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 2a, where the environment part, coming from the abstract environment, is grayed out).

In Figure 2b, 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 2c. 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, the call is a *visible* interface interaction (Figure 2d). This fact should be reflected in the open semantics.

In general, we can see an instance to be split into two halves, one containing the component fields and methods and the other the ones provided by the environment. The environment part of the objects *created by the component* is *unconnected* unless brought in connection by (outgoing) communication, sending some object identities as parameter or return values across the border. In the above example, if the set-method is defined in

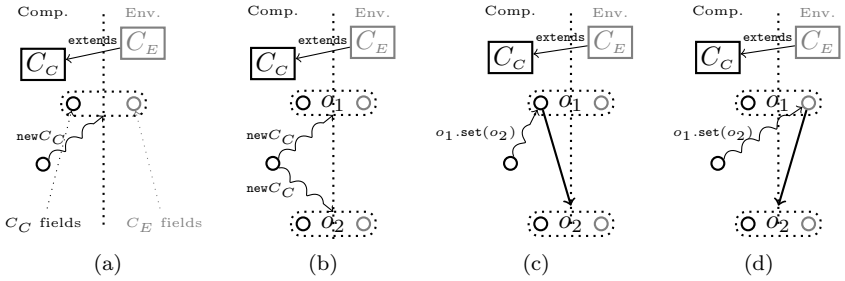

Fig. 2: Heap structure and connectivity

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 2d). 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.

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 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 assumptions concerning the heap structure ("connectivity"), 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 4 formalizes these intuitions.

3. Calculus

This section presents the calculus, its syntax and operational semantics. It is a concurrent variant of an imperative, object-calculus in the style of the calculi from Abadi and Cardelli [1996] with asynchronous method calls. Unlike in de Boer et al. [2007], Ábrahám et al. [2009], 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 and present the type system in Section 3.2.

bi

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	::=	$t \mid \mathsf{if}\, v = v\, \mathsf{then}\, e\, \mathsf{else}\, e \mid \mathsf{if}\, \mathit{undef}(v.l())\, \mathsf{then}\, e\, \mathsf{else}\, e$	expr.
	|	$n@l(\vec{v}) \mid v.l() \mid v.l() := v$	
	|	$\mathsf{new}\, n \mid \mathsf{claim}@(n, n) \mid \underline{\mathsf{get}@n} \mid \mathsf{suspend}(n) \mid \underline{\mathsf{grab}(n)} \mid \underline{\mathsf{release}(n)}$	
v	::=	$x \mid n \mid ()$	values
L	::=	$\bot \mid \top$	lock status

Table 1: Abstract syntax

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, 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; in an expression $\mathsf{let}\, x{:}T = e\, \mathsf{in}\, t$, the $\mathsf{let}\, x$ acts as a binder for occurrences of x in t. As usual, sequential composition $t_1; t_2$ abbreviates $\mathsf{let}\, x{:}T = t_1\, \mathsf{in}\, t_2$, where x does not occur free in t_2. During 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 $\varsigma(n{:}T).\lambda(x_1{:}T_1, \ldots x_k{:}T_k).t$ provides the method body t abstracted over the ς-bound "self" parameter, here n, and the formal parameters $x_1, \ldots, x_k$. For fields, they are either a value or yet undefined. In freshly created objects, the lock is free, and all class-typed fields carry undefined references $\bot_c$, where class name c is the type of the field. For basic types such as integers, booleans, etc. fields carry concrete values like true, false, 0, 1, ... of appropriate types as initial values; in the theoretical development, any built-in basic types, their values, and appropriate operations do not play a role and left out mostly; for instance, we don't formalize well-typedness conditions for those basic types and values. We allow, however, to use of them in illustrative examples.

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. Note further that the syntax does not allow to set a field back to undefined. Direct access (read or write) to fields of objects other than oneself is forbidden by convention, i.e., it is forbidden that a method ever executes $o.f()$ resp. $o.l() := v$ for an object different from "self". More precisely, we assume that field accesses $v.l()$ and field updates $v.l() := v$ in the static code, i.e., in the method bodies, can use the ς-bound self-parameter as v, only; the parameter corresponds to the reserved word `this` in Java. 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 superclass, 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 where the caller blocks for the return of the result. 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 $\mathsf{claim}@(p, o)$ 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* losing 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 $\mathsf{suspend}(o)$ 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 $\mathsf{claim}@(p, \mathit{self})$ and $\mathsf{suspend}(\mathit{self})$.

3.2. *Type system*

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

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

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. The member types U serve to give types for methods and fields in classes. As auxiliary type constructions (i.e., not as part of the user syntax,

but to formulate the type system) we need 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 $(l){:}U$). For a given signature, we write $S.l$ to mean that S contains a member labelled l and that $S.l$ denotes its type. 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 matches in the rules, and write $\mathsf{Unit} \to T$ for $T_1 \times \ldots \times T_k \to T$ 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 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.[¶] 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 $C_1 \parallel C_2$.

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 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 introduces 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

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

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

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

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

Table 2: Typing (component level)

for the lower level entities (in this particular case, by rule T-Obj from Table 3). The judgments 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 $\bullet; \Delta, c{:}[\![S]\!] \vdash [\![O]\!] : c$; 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 $\Gamma; \Delta \vdash f_i : T_i$ and $\Gamma; \Delta \vdash m_j : U_j$ check the well-typedness of all implemented members of the class. We silently assume that f_i ranges over all fields and m_j 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 with 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 similarly, 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.

$$\frac{\Gamma;\Delta \vdash c_1 : [\![\vec{l_f}{:}\vec{T}, \vec{l}{:}\vec{U}, (\vec{l'}){:}\vec{U'}]\!] \quad \Gamma;\Delta \vdash f_i : T_i \quad \Gamma;\Delta \vdash m_j : U_j \quad m_j = \varsigma(s_j{:}c_1).\lambda(\vec{x}_j{:}\vec{T}_j).t_j \quad \Gamma;\Delta \vdash c_2.l'_{j'} : U'_{j'}}{\Gamma;\Delta \vdash [\![c_2, \vec{l_f} = \vec{f}, \vec{l} = \vec{m}]\!] : c_1}\ \text{T-Class}$$

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

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

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

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

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

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

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

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_2 extends c_1. Assume further that c_2 implements the two methods labelled l_1 and l_3, and that the super-class c_1 implements l_1 and l_2. The expected interfaces for the two classes are therefore

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

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

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

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

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

Table 4: Typing (2)

for c_1 and c_2 respectively. As seen in the (right-hand) interface of Equation (5), the available methods of instances of c_2 are l_1 (implemented by c_2, and overriding the corresponding method of c_1), l_2 (which is *not* implemented by c_2 but inherited), and l_3, which again is implemented by c_2. The derivation for both classes ends with an instance of rule T-Par:

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

Note that the interface $[\![S_1]\!]$ for c_1 is used as *assumption* to type-check c_2 and vice versa. In the derivation, we use the following abbreviations:

$$\begin{array}{rcl} \Delta_3 & \triangleq & \Delta_1, c_2{:}[\![S_2]\!] \\ \Delta_2 & \triangleq & \Delta_0, c_2{:}[\![S_2]\!] \\ \Delta_1 & \triangleq & \Delta_0, c_1{:}[\![S_1]\!] \end{array} \quad (7)$$

The second premise of Equation (6) gives rise to the following sub-derivation:

$$\frac{\dfrac{\Delta_3 \vdash c_2 : [\![S_2]\!] \quad \Delta_3 \vdash m_1' : U_1 \quad \Delta_3 \vdash m_3 : U_3 \quad \Delta_3 \vdash c_1.l_2 : U_2}{\Delta_3 \vdash [\![c_1, l_1 = m_1', l_3 = m_3]\!] : c_2}\ \text{T-Class}}{\Delta_1 \vdash c_2[\![c_1, l_1 = m_1', l_3 = m_3]\!] : (c_2{:}[\![S_2]\!])}\ \text{T-NClass} \quad (8)$$

The type-check of the second premise of Equation (6) works similarly. □

4. Typed operational semantics for open systems

The operational semantics is given in two stages, component *internal* steps and *external* ones, where the latter describe the interaction at the interface. Section 4.1 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 semantics, presented afterwards in Section 4.2, define the interaction between component and environment. They are defined in reference to assumption and commitment contexts.

4.1. *Internal semantics*

The internal steps rewrite components as given in the abstract grammar from 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

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

In the semantics later, we will distinguish confluent steps $\rightsquigarrow$ and non-confluent ones $\xrightarrow{\tau}$; when being unspecific we simply write $\to$ 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 2 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.

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. In the rule Let dealing with nested let-constructs, the variable x_1 is assumed not to occur free in t. 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.[‖] 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 $p\langle call\ o.l(\vec{v}){:}T\rangle!$ describes the message for the method call. The expression run-time syntax and additional to the grammar of Table 1), as part of the productions for C.

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). In the configuration after the reduction step, the meta-mathematical notation $M.l(o)(\vec{v})$ stands for $t[o/s][\vec{v}/\vec{x}]$, when the method suite M contains a method definition of the form $l = \varsigma(s{:}T).\lambda(\vec{x}{:}\vec{T}).t$, which in this case is unique.

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 determine which fields and methods are meant in a call is formalized in the function *members* from Definition 2. 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).[††] Fields are considered private and thus subject to the same naming convention as the one for private methods. 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 method to implement the embedding in Definition 2, Rule M-TOP deals with a class without super-class (which corresponds to `Object` in Java), in which case the

[‖] 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.

[††] We furthermore do not consider overloading here.

fields and methods 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 of c_2 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 not by the *members*-function, but statically by the type system. The public availability of methods for instances of a class is determined by the signature of the class and subtyping via subsumption allows to hide methods and make them thus unavailable.

Definition 2 (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

With this definition, 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.

Example 3. Assume two classes

$$\begin{aligned}
\Gamma^c \vdash \mathit{circle1} = [\![\ \bot,\ & \mathit{setCenterX} = \varsigma(s{:}\mathit{circle1}).\lambda(x{:}\mathit{float}).\ s.\mathit{centerX}() := x,\\
& \mathit{setCenterY} = \varsigma(s{:}\mathit{circle1}).\lambda(x{:}\mathit{float}).\ s.\mathit{centerY}() := x,\\
& \mathit{setRadius} = \varsigma(s{:}\mathit{circle1}).\lambda(x{:}\mathit{float}).\ s.\mathit{radius1}() := x,\\
& \mathit{centerX} = 0.0,\ \mathit{centerY} = 0.0,\ \mathit{radius1} = 0.0\ ]\!]\\
\Gamma^c \vdash \mathit{circle2} = [\![\ \mathit{circle1},\ & \mathit{setRadius} = \varsigma(s{:}\mathit{circle2}).\lambda(x{:}\mathit{float}).\ s.\mathit{radius2}() := x; s.A() := x{*}x{*}\mathit{pi},\\
& \mathit{radius2} = 0.0,\ A = 0.0,\ \mathit{pi} = 3.14\ ]\!]\ .
\end{aligned}$$

Then

$$\begin{aligned}
\Gamma^c \vdash \mathit{members(circle2)} = \\
& \mathit{setRadius} = \varsigma(s{:}\mathit{circle2}).\lambda(x{:}\mathit{float}).\ s.\mathit{radius2}() := x; s.A() := x{*}x{*}\mathit{pi},\\
& \mathit{setCenterX} = \varsigma(s{:}\mathit{circle1}).\lambda(x{:}\mathit{float}).\ s.\mathit{centerX}() := x,\\
& \mathit{setCenterY} = \varsigma(s{:}\mathit{circle1}).\lambda(x{:}\mathit{float}).\ s.\mathit{centerY}() := x,\\
& \mathit{radius2} = 0.0,\ A = 0.0,\ \mathit{pi} = 3.14,\ \mathit{centerX} = 0.0,\ \mathit{centerY} = 0.0,\ \mathit{radius1} = 0.0\,.
\end{aligned}$$

Note that though the field *radius1* is contained in instances of *circle2*, this field is not accessible.

Claiming as well as executing the get-expression fetches the value of a future reference.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Table 6: Internal steps

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 grab are dual and set the lock to free resp. set it to the state $\top$ of "taken". Both operations are *not* user syntax. The expression suspend, finally, introduces a scheduling point by temporarily releasing and then trying to re-acquire the lock.

Example 4 (Internal semantics). Assume that the class *circle2* from Example 3 is defined in the component. Then calling the method *setCenterX* of an instance o of *circle2* is an internal call. Its execution creates a fresh thead (rule NewT) that grabs the object's lock and executes the method call (rule Call). The method sets the value of a field (rule FSet) and, after a reduction (rule Let), releases the lock (rule Release).

$$
\begin{aligned}
& o[circle2, centerX = 0.0, \ldots, \bot] \parallel p'\langle \mathsf{let}\ x{:}\langle circle2\rangle = o@setCenterX(5.0)\ \ \mathsf{in}\ t\rangle \\
\rightsquigarrow\ & o[circle2, centerX = 0.0, \ldots, \bot] \parallel \nu(p{:}\langle circle2\rangle).(p'\langle \mathsf{let}\ x{:}\langle circle2\rangle = p\ \ \mathsf{in}\ t\rangle \parallel \\
& \qquad\qquad p\langle call\ o.setCenterX(5.0){:}circle2\rangle!)) \\
\xrightarrow{\tau}\ & o[circle2, centerX = 0.0, \ldots, \top] \parallel \nu(p{:}\langle circle2\rangle).(p'\langle \mathsf{let}\ x{:}\langle circle2\rangle = p\ \ \mathsf{in}\ t\rangle \parallel \\
& \qquad\qquad p\langle \mathsf{let}\ x{:}circle2 = o.centerX() := 5.0\ \mathsf{in}\ \mathsf{release}(o); x\rangle) \\
\xrightarrow{\tau}\ & o[circle2, centerX = 5.0, \ldots, \top] \parallel \nu(p{:}\langle circle2\rangle).(p'\langle \mathsf{let}\ x{:}\langle circle2\rangle = p\ \ \mathsf{in}\ t\rangle \parallel \\
& \qquad\qquad p\langle \mathsf{let}\ x{:}circle2 = o\ \mathsf{in}\ \mathsf{release}(o); x\rangle) \\
\rightsquigarrow\ & o[circle2, centerX = 5.0, \ldots, \top] \parallel \nu(p{:}\langle circle2\rangle).(p'\langle \mathsf{let}\ x{:}\langle circle2\rangle = p\ \ \mathsf{in}\ t\rangle \parallel \\
& \qquad\qquad p\langle \mathsf{release}(o); o\rangle) \\
\xrightarrow{\tau}\ & o[circle2, centerX = 5.0, \ldots, \bot] \parallel \nu(p{:}\langle circle2\rangle).(p'\langle \mathsf{let}\ x{:}\langle circle2\rangle = p\ \ \mathsf{in}\ t\rangle \parallel \\
& \qquad\qquad p\langle o\rangle)
\end{aligned}
$$

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

4.2. *External semantics*

In the external semantics, 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

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

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

Table 8: Reduction modulo congruence

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

Table 9: Labels

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.

4.2.1. *Connectivity contexts and cliques* An important condition for the open semantics concerns which *combinations* of names can occur in communications. A well-typed component thus takes into account the *relation* of objects from the assumption context Δ among 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. See the discussion related to Figure 2 for an illustration of connectivity as heap abstraction.

Definition 5 (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 satisfies 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 $\Theta \vdash n : T$.

To facilitate the following notationally, we use the following conventions.

Notation 6 (Contexts). We abbreviate the pair $\Delta; E_\Delta$ and $\Theta; E_\Theta$ of both assumption and commitment context by Ξ, i.e., we write for instance $\Xi \vdash C$ 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 7 (Connectivity contexts). The semantics 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 5). 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.:

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

and dually $E_\Theta \subseteq (\Theta_o \times \Theta_o) + (\Theta_o \times \Xi_p) + (\Theta_p \times \Theta_o)$. 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, connectivity contexts E_Δ express assumptions about the environment, and E_Θ commitments of the component.

Remark 8 (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 9 (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.,

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

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 $o \hookrightarrow p \in E_\Delta$, and analogously $\Delta; E_\Delta \vdash p \hookrightarrow o$. Note that we use the transitive and reflexive closure for the connectivity among object identities, only.

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

Definition 10 (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.

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

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

Table 10: Binding

The rules for the find function of Table 10 work straightforwardly, determining the class a member is defined in. Unlike the members function from Definition 2, the functions here uses the *interface* information to find the class. The members 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 labeled 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 11 (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: The find-function determines the class c' where the method is implemented and $\Gamma^{\mathrm{c}}_{\Theta} \vdash c'$ assures that the class is part of the component, as in the open semantics, only the class table $\Gamma^{\mathrm{c}}_{\Theta}$ 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 the 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 requirement is that the name is indeed fresh, and that the type mentioned in the label is actually a type (stipulated by $\Xi \vdash T$). Besides that, the name creation must come from an execution in an object on the environment side, stipulated by the second premise.

The interface interaction provides also information that updates the contexts.

Definition 12 (Name context update). Let Ξ be a context and a an *incoming* label, with $\Xi \vdash a$ (cf. Definition 11) with incoming label a. The updated context $\acute{\Xi} = \Xi + a$ is defined as follows (dually for outgoing communication):

1 $a = \nu(n{:}T)_{o'}?$, then $\acute{\Delta} = \Delta, n{:}T$ and $\acute{\Theta} = \Theta$.
2 If $a = p\langle call\ o.l(\vec{v}){:}T\rangle?$, 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 $\vec{T}$ resp. c of the arguments $\vec{v}$ resp. of o are given by $\Delta \vdash v_i : T_i$, resp. $\Delta \vdash o : c$.
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 from (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

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

$$\frac{\Delta \vdash p : \langle T\rangle \qquad \Xi \vdash v : T}{\Xi \vdash p\langle get(v)\rangle?}\ \text{LT-GetI} \qquad \frac{\Xi \nvdash n \qquad \Delta \vdash o{:}c \qquad \Xi \vdash T}{\Xi \vdash \nu(n{:}T)_o?}\ \text{LT-NewI}$$

Table 11: Checking static assumptions

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

Table 12: Connectivity check

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 5 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 5, 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.

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

Definition 13 (Connectivity context check). Let Ξ be a context and a be an incoming communication label. Overloading the notation from Definition 11, 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 $E_\Delta \vdash o' \leftrightharpoons o, \vec{v}$. 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 $E_\Delta \vdash p \hookrightarrow o \leftrightharpoons v$, 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. 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 14 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 14 (Connectivity context update). Assume $\Xi \vdash a$. The update of the connectivity contexts $\acute{\Xi} = \Xi + a$ is defined as follows. (The definition for outgoing communication is dual.)

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

The situation of the connectivity context update is also illustrated in Figure 2 in Section 2.4. In Figures 2a and 2b, the component on the left-hand side creates two objects o_1 and o_2. Assume that component thread executing in object o issues the two object-creation statements The two interface communications are thus labeled by $\nu(o_1{:}C_C)_o!$ and $\nu(o_2{:}C_C)_o!$. According to the dual situation of part 1 of Definition 14, the assumption context $\acute{E}_\Theta$ after the two steps contains $o \hookrightarrow o_1$ and $o \hookrightarrow o_2$ and, via transitivity, reflexivity, and symmetry, also $\acute{E}_\Theta \vdash o_1 \leftrightharpoons o_2$. This reflects that fact, that the sender o_1 and o_2 *may* know each other without further interface interaction. That is illustrated in

a step to Figure 2c, when an call to the component-*internal* set-method sets a component field of o_1 to point to o_2 (depicted by the corresponding bold arrow in the figure). However, the assumption context $\acute{E}_\Delta$ after the steps is unchanged compared to the situation E_Δ before the steps, which means that $\acute{E}_\Delta \not\vdash o_1 \leftrightharpoons o_2$, since o_1 and o_2 are two (different) fresh identifiers. In that situation, for instance no incoming call of the form $p''\langle call\ o'.l(o_1, o_2){:}T\rangle?$ would be possible, since the connectivity check from Definition 13 would fail for incoming communication. Using a set-method which is implemented by the environment instead, the situation changes, which is illustrated in the step from Figure 2b to 2d: In this situation, the method call to the set method is an external communication, labeled $p\langle call\ o_1.set(o_2){:}\mathsf{Unit}\rangle!$. According to the dual situation of Definition 14(2), $\acute{E}_\Theta$ is further updated to contain $o_1 \hookrightarrow o_2$, which is shown by the bold arrow in Figure 2d (we ignore the role of the thread identifier in the example).

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

$$\Delta; E_\Delta \vdash C : \Gamma^c_\Theta, \Theta; E_\Theta \ . \tag{15}$$

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 is given implicitly. As before, we abbreviate the judgment of Equation (15) as $\Xi \vdash C$ (cf. Notation 6). 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 2 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 15. 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:

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

The definition for Γ^c_Δ works dually.

Using the *find*-function from Definition 10, 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 $R.l$ selects the intended one. For instance, in the situation of Listing 2 in the informal exposition of Section 2, invoking *members* on the subclass in the component will give back *only* method m_C, since only this method, unlike m_E of the super-class is actually implemented by the component itself. Assuming classes as given in Listing 3, where here C_E is assumed to be an *component* class (in contrast to the discussion in Section 2), invoking *members* on the subclass C_E gives back implementations C_E's implementation for m_2 independent of the fact whether C_C is a component or an environment class, since the method is overridden in C_E. Whether *members* also contains the method m_1 depends on whether the super-class is a component class, as well, or not.

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 labeled 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 is 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 12(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 is possible. Note that unlike in rule NewOO, no object half is actually instantiated in step. 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 $p\langle call\ o.l(\vec{v}){:}T\rangle!$. 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 a 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 is 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 is 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.

The 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 $C(\vec{v})$ to denote the corresponding newly instantiated object-halves. 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 labeled 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.

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

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

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

$$\frac{a = p\langle call\ o.l(\vec{v}){:}T\rangle! \qquad C = C' \parallel o[c, M, F, \bot] \qquad M.l = \bot \qquad \acute{\Xi} = \Xi + a}{\Xi \vdash C \parallel a \xrightarrow{a} \acute{\Xi} \vdash C}\ \textsc{CallO}$$

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

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

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

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

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

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

Table 13: External steps

The following example illustrates the interface behavior and the role of the assumption and commitment context on a small example involving inheritance.

Example 16 (Observable trace). Assume class c' extends c, inheriting a method m from c and containing a method m' which is sketched in Listing 5. Assume further, that c' is a component class and c is a class from the environment.

Listing 5: Observable trace and cross-border inheritance

```
T m' () {
    let x2 : c = new c()
    in let y1:c'' = this.m()
    in let y2:c'' = x2.m()
    in t
}
```

In the above code, the method-call notation $x.m()$ abbreviates an asynchronous call to m followed immediately by the corresponding get-operation on the corresponding future, i.e., $\mathsf{let}\, y{:}T = x.m()\,\mathsf{in}\, t$ abbreviates $\mathsf{let}\, y{:}T = (\mathsf{let}\, y'{:}\langle T\rangle = x@m()\,\mathsf{in}\,\mathsf{get}@y')\,\mathsf{in}\, t$. Furthermore, we use the more conventional this instead of the ς-bound self-parameter of the formal calculus. Assuming that method m' is being invoked on an object o_1 (meaning that this is substituted by o_1 in the running code), one possible trace, using the operational rules of Table 6 and 13, looks as follows:

$$\begin{aligned} &p\langle call\ o_1.m'():T\rangle?.\nu(o_2{:}c)_{o_1}!. \\ &\quad \nu(p_1{:}\langle c''\rangle)_{o_1}!.p_1\langle call\ o_1.m():c''\rangle!.p_1\langle get(o_1')\rangle?. \\ &\quad \nu(p_2{:}\langle c''\rangle)_{o_1}!.p_2\langle call\ o_2.m():c''\rangle!.p_2\langle get(o_2')\rangle? \end{aligned} \tag{18}$$

After invocation of m', the method creates a new object o_2, indicated by the outgoing ν-label. The remaining 6 labels represent the two method calls with o_1, resp. o_2 as callee (the call to o_1 is a direct self-call), and the corresponding communication of the results back to the caller, represented by the two get-labels. Due to the asynchronous nature of communication, that trace is only one possible behavior, e.g., alternatively the order of the outgoing calls may be swapped.

The change of the assumption and commitment contexts during the execution is shown in equation (19). The second column contains the respective communication labels, the last 4 columns contain the corresponding contexts *after* executing the label; in case the step leaves the context unchanged, we leave the corresponding entry empty. For the contexts Θ and Δ, we elide the typing information. Furthermore, the super-scripts refer to the line in the table and the contexts without grave-accent to the state *before* the step. For example, in the first line, the entry $\Delta^0 \setminus p$ represents the assumption context $\acute{\Delta}^0$ after the incoming call, relating it to the assumption context Δ^0 before that initial call.

	label	$\acute{E}_\Theta$	$\acute{\Theta}$	$\acute{\Delta}$	$\acute{E}_\Delta$
0	$p\langle call\ o_1.m'():T\rangle?$	$E^0_\Theta, p{\hookrightarrow}o_1$	Θ^0, p, o_1	$\Delta^0 \setminus p$	
1	$\nu(o_2{:}c)_{o_1}!.$	$E^1_\Theta, o_1{\hookrightarrow}o_2$	Θ^1, o_2		
2	$\nu(p_1{:}\langle c''\rangle)_{o_1}!$	$E^2_\Theta, o_1{\hookrightarrow}p_1$	Θ^2, p_1		
3	$p_1\langle call\ o_1.m():c''\rangle!$		$\Theta^3 \setminus p_1$	Δ^3, p_1, o_1	$E^3_\Delta, p_1{\hookrightarrow}o_1$
4	$p_1\langle get(o'_1)\rangle?$	$(E^4_\Theta \setminus o_1{\hookrightarrow}p_1), o_1{\hookrightarrow}o'_1$		$\Delta^4 \setminus p_1$	$E^4_\Delta \setminus p_1{\hookrightarrow}o_1$
5	$\nu(p_2{:}\langle c''\rangle)_{o_1}!$	$E^5_\Theta, o_1{\hookrightarrow}p_2$	Θ^5, p_2		
6	$p_2\langle call\ o_2.m():c''\rangle!$		$\Theta^6 \setminus p_2$	Δ^6, p_2, o_2	$E^6_\Delta, p_2{\hookrightarrow}o_2$
7	$p_2\langle get(o'_2)\rangle?$	$(E^7_\Theta \setminus o_1{\hookrightarrow}p_2), o_1{\hookrightarrow}o'_2$		$\Delta^7 \setminus p_2$	$E^7_\Delta \setminus p_2{\hookrightarrow}o_2$

(19)

The table represents the context *updates* for the various steps from Definition 12 and 14. The *checks* for well-typedness and connectivity from Definitions 11 and 13 are given in (20). As mentioned, for *outgoing* communication, well-typedness and connectivity of the interaction label are *not* checked by the premises of the rules of Table 13, as their satisfaction is maintained by the steps of the semantics. The table from (20) list the checks (which are the exact duals of their counter-parts for incoming communication) nonetheless. When considering the trace of (18) in isolation, i.e., not as observable behavior of the concrete program from Listing 5, then the checks for incoming and outgoing communications would validate that the trace is the behavior of a arbitrary program with the statically given classes c and c' and their inheritance structure. In the table of (20), the inheritance structure is used in the call-steps, where $\Xi \vdash \mathit{find}(c', m') = c'$, and $\Xi \vdash \mathit{find}(c', m) = c$ determine c' resp. c implementing the corresponding method body, and $\Gamma^c_\Theta \vdash c'$ resp. $\Gamma^c_\Delta \vdash c$ determine that c' is a component class and c a class of the environment.

	label	*typing*	*connectivity*
0	$p\langle call\ o_1.m'():T\rangle?$	$\Xi \vdash o_1{:}c'$, $\Xi \vdash \mathit{find}(c', m') = c'$, $\Gamma^c_\Theta \vdash c'$ $\Xi \vdash c'{:}[\![m'{:}\mathsf{Unit} \to T, \ldots]\!]$, $\Delta \vdash p{:}\langle T\rangle$	$\Delta \vdash o{:}\tilde{c}, E_\Delta \vdash o{\hookrightarrow}p, E_\Delta \vdash o{\leftrightharpoons}o_1$
1	$\nu(o_2{:}c)_{o_1}!.$	$\Xi \not\vdash o_2, \Theta \vdash o_1{:}c', \Xi \vdash c$	
2	$\nu(p_1{:}\langle c''\rangle)_{o_1}!$	$\Xi \not\vdash p_1, \Theta \vdash o_1{:}c', \Xi \vdash \langle c''\rangle$	
3	$p_1\langle call\ o_1.m():c''\rangle!$	$\Xi \vdash o_1{:}c'$, $\Xi \vdash \mathit{find}(c', m) = c$, $\Gamma^c_\Delta \vdash c'$, $\Xi \vdash c : [\![m{:}\mathsf{Unit} \to c'', \ldots]\!]$, $\Theta \vdash p_1{:}\langle c''\rangle$	$\Theta \vdash o_1{:}c', E_\Theta \vdash o_1{\hookrightarrow}p_1, E_\Theta \vdash o_1{\leftrightharpoons}o_1$
4	$p_1\langle get(o'_1)\rangle?$	$\Delta \vdash p_1{:}\langle c''\rangle, \Xi \vdash o'_1{:}c''$	$E_\Delta \vdash p_1{\hookrightarrow}o_1{\leftrightharpoons}o'_1, E_\Theta \vdash o_1{\hookrightarrow}p_1$
5	$\nu(p_2{:}\langle c''\rangle)_{o_1}!$	$\Xi \not\vdash p_2, \Theta \vdash o_1{:}c', \Xi \vdash \langle c''\rangle$	
6	$p_2\langle call\ o_2.m():c''\rangle!$	$\Xi \vdash o_2{:}c$, $\Xi \vdash \mathit{find}(c, m) = c$ $\Gamma^c_\Delta \vdash c$, $\Xi \vdash c : [\![m{:}\mathsf{Unit} \to c'', \ldots]\!]$, $\Theta \vdash p_2{:}\langle c''\rangle$	$\Theta \vdash o_1{:}c', E_\Theta \vdash o_1{\hookrightarrow}p_2, E_\Theta \vdash o_1{\leftrightharpoons}o_2$
7	$p_2\langle get(o'_2)\rangle?$	$\Delta \vdash p_2{:}\langle c''\rangle, \Xi \vdash o'_2{:}c''$	$E_\Delta \vdash p_2{\hookrightarrow}o_2{\leftrightharpoons}o'_2, E_\Theta \vdash o_1{\hookrightarrow}p_2$

(20)

Two points are worth noting: the requirement $\Xi \not\vdash o_2$ expressing that o_2 is *fresh* in line 1 of (20) ultimately entails that after line 6, object o_2 cannot be connected to o'_1,

i.e. $E^7_\Delta \not\vdash o_2 \leftrightharpoons o'_1$. In other words, due to the connectivity check in line 7 of (20), the last incoming communication $p_2\langle get(o'_2)\rangle?$ of the program is impossible if $o'_2 = o'_1$, and the corresponding trace would be illegal, and the assertion $y_1 \neq y_2$ as pre-condition to t in Listing 5 would be provable. The information that y_1 and y_2 are no aliases (at that point) can be used to show that when replacing the rest of the method t by an alternative t' where subsequent sequential accesses to y_1 and y_2 are executed in parallel instead is an *observably equivalent* variation of t. Without the open semantics keeping track of the potential connectivity of objects, such an optimization would not be possible.

Furthermore note that for the calls, the type information is used to determine where the code of the method resides. That is done with the help of the *find*-function (cf. Definition 10). E.g., in line 0 of (20), method m' is implemented in class c', and in lines 3 resp. 6, *find* establishes the environment class c to contain m. E.g., for the outgoing call of the self-call, the condition $\Gamma^c_\Delta \vdash c'$ asserts that c' (which implements m) is an environment class, and therefore the self-call is an interface interaction; if $\Gamma^c_\Theta \vdash c'$ instead, the call would be a component-internal step.

Finally consider a restricted set-up, where, unlike as in the example here, component code can neither instantiate an environment class nor where a component class can inherit methods from environment classes. In that situation, *outgoing* communication labels of the form $\nu(o{:}c)_o!$ (as in line 1) would not occur, and as a consequence, for *incoming* communication, negative connectivity assertions such as $E^7_\Delta \not\vdash o_2 \leftrightharpoons o'_1$ from above would never be derived. The restricted set-up corresponds roughly to the use of class *libraries*: the client code using the library of course can instantiate library classes or extend them via inheritance, but the converse direction is not possible. In that setting, the semantics simplifies considerably as the connectivity can be ignored to obtain a precise open semantics. Furthermore, self-calls of a library component are always internal. □

Remark 17 (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 is 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 is included in our representation, namely: whether or not a class overrides a method or inherits it is *observable* from the outside. □

Remark 18 (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 represent information about the status of the lock as far

as the environment is concerned. A lock status $\perp$ 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 the absence of re-entrant threading, a lock status of $\perp$ 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 is unobservable from the perspective of the component, whether or not another thread has taken the lock again in the meantime. 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. Ábrahám et al. [2008], which deals with re-entrant monitor behavior). □

Remark 19 (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♯. 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 for a precise characterization, one would need to characterize the *re-entrant* behavior of threads: the future references here would be interpreted as thread identifiers and for each thread identifier, the trace must be a (prefix of) a context-free language of matching calls and returns. That corresponds to *well-bracketed* strategies in game theory (cf. e.g. Abramsky and McCusker [1997]).

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

$$\Xi \vdash \epsilon : \mathit{trace} \qquad \text{L-Empty}$$

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

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

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

Table 14: Legal traces (dual rules omitted)

re-entrant locks are explored in Ábrahám et al. [2006], 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. □

5. Interface behavior and legal traces

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 careful 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. The rules of Table 14 specify legality of traces. We use the same conventions and notations as for the operational semantics (cf. Notation 6). The judgments in the derivation system are of the form

$$\Xi \vdash s : \mathit{trace}\ . \tag{21}$$

We write $\Xi \vdash s : \mathit{trace}$, if there exists a derivation according to the rules of Table 14. 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 $\Xi \vdash C$ 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 20. 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 21 (Subject reduction). Assume $\Xi \vdash C$.

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

Proof. All parts by induction on the derivation for different reduction relations resp. equivalence relation. For all parts, we are given $\Xi \vdash C$, (see Definition 20). The assertion $\Xi \vdash C$ consists of a typing part and a part asserting that the actual connectivity of C is over-approximated by the commitments E_Θ of Ξ.

We start by proving preservation of the well-typedness part. For part 1a, assume $C \equiv \acute{C}$. The equivalence relation is given in Table 7 as the reflexive, transitive, and symmetric closure of the rules shown. Reflexivity is trivial, transitivity follows by induction in the number of $\equiv$-steps. For $C = \mathbf{0} \parallel C'$, assume $\Xi \vdash \mathbf{0} \parallel C'$. For the typing part, it means $\Delta_1, \Delta_1 \vdash \mathbf{0} \parallel C' : \Theta_1, \Theta_2$ with $\Delta_1, \Theta_2 \vdash \mathbf{0} : \Theta_1$ and $\Delta_2, \Theta_1 \vdash C' : \Theta_2$ by sub-derivation (as premises of T-Par). Since $\mathbf{0}$ contains no objects, Θ_1 must be empty, too. Note that the subsumption rule T-Sub does not allow to remove or add objects identities. Hence, $\Theta_2 = \Theta$ which means $\Delta_1 \vdash C' : \Theta$. Weakening the environment assumptions Δ_1 to Δ_1, Δ_2 gives $\Delta \vdash C' : \Theta$, as required. The inverse direction, given $\Xi \vdash C'$, is simpler since $\Xi \vdash C' \parallel \mathbf{0}$ follows by parallel composition and the fact that $\mathbf{0}$ is well-typed under any assumptions (and with empty commitments). Symmetry in the second rule of Table 7 is straightforward, as the treatment of typing is defined symmetrically wrt. $\parallel$. For associativity, the typing part follows by inversion/application of the rule T-Par. The

cases for ν-binders are straightforward (observing that for the forth equation, n does not occur free in C_1, as mentioned in the text).

The preservation for reduction modulo congruence from Table 8 (for $C \xrightarrow{\tau} \acute{C}$ and $C \rightsquigarrow \acute{C}$ follows from part 1a for $\equiv$, induction, and preservation for basic $\xrightarrow{\tau}$-steps resp. $\rightsquigarrow$-steps. For steps of the form $\xrightarrow{\tau}$, the basic steps are defined in Table 6 and "embedded" into a larger context by the 5th and 6th rule of Table 8. Those basic steps are proven by case distinction on the respective rules of Table 6:

Case: RED: $p\langle \mathsf{let}\, x{:}T = v\, \mathsf{in}\, t\rangle \rightsquigarrow p\langle t[v/x]\rangle$
The well-typedness assumption $\Xi \vdash C$ implies $\Delta' \vdash p\langle \mathsf{let}\, x{:}T = v\, \mathsf{in}\, t\rangle : (p{:}\langle T'\rangle)$ for some name context Δ' and some type T' and furthermore, by inverting rules T-NTHREAD (from Table 2), T-LET (from Table 3), $\bullet; \Delta', p{:}\langle T'\rangle \vdash v : T$ and $x{:}T; \Delta', p{:}\langle T'\rangle \vdash t : T'$. Hence, by a (standard) substitution lemma, i.e., preservation of typing under substitution, $\bullet; \Delta', p{:}\langle T'\rangle \vdash t[v/x] : T'$, and the result follows by T-NTHREAD.

Case: NEWO: $p\langle \mathsf{let}\, x{:}T = \mathsf{new}\, c\, \mathsf{in}\, t\rangle \rightsquigarrow \nu(o{:}c).(o[c, M, F, \bot] \parallel p\langle \mathsf{let}\, x{:}T = o\, \mathsf{in}\, t\rangle)$
relative to the class table Γ^c and where $\Gamma^c \vdash \mathit{members}(c) = M, F$. Remember, that in the operational semantics, we group all class definitions together into Γ^c as class table. The well-typedness assumption $\Xi \vdash p\langle \mathsf{let}\, x{:}T = \mathsf{new}\, c\, \mathsf{in}\, t\rangle$ means $\Delta \vdash p\langle \mathsf{let}\, x{:}T = \mathsf{new}\, c\, \mathsf{in}\, t\rangle : \Theta$. Inverting T-NTHREAD and the rule for the let-construct gives

$$\dfrac{\dfrac{\bullet; \Delta, p{:}\langle T'\rangle \vdash \mathsf{new}\, c : T \quad x{:}T; \Delta, p{:}\langle T'\rangle \vdash t : T'}{\bullet; \Delta, p{:}\langle T'\rangle \vdash \mathsf{let}\, x{:}T = \mathsf{new}\, c\, \mathsf{in}\, t : T'}\,\text{T-LET}}{\Delta \vdash p\langle \mathsf{let}\, x{:}T = \mathsf{new}\, c\, \mathsf{in}\, t\rangle : \Theta}\,\text{T-NTHREAD} \tag{22}$$

where $\Theta = p{:}\langle T'\rangle$. The left-most premise $\bullet; \Delta, p{:}\langle T'\rangle \vdash \mathsf{new}\, c : T$ is justified by a sequence of instances of the subsumption rule T-SUB and one instance of T-NEW, which implies that type $T = c'$ for some class name c' and $\Delta, p{:}\langle T'\rangle \vdash c \le c'$.

The configuration after the step can be derived as follows (abbreviating $\Delta, p{:}\langle T'\rangle$ as Δ'):

$$\dfrac{\Delta, \Theta \vdash o[c, M, F, \bot] : o{:}c \qquad \dfrac{\dfrac{\dfrac{\bullet; \Delta', o{:}c \vdash o : c \quad \Delta', o{:}c \vdash c \le c'}{\bullet; \Delta', o{:}c \vdash o : c'} \qquad x{:}c'; \Delta', o{:}c \vdash t : T'}{\bullet; \Delta', o{:}c \vdash \mathsf{let}\, x{:}c' = o\, \mathsf{in}\, t : T'}}{\Delta, o{:}c \vdash p\langle \mathsf{let}\, x{:}c' = o\, \mathsf{in}\, t\rangle : \Theta}}{\Delta \vdash o[c, M, F, \bot] \parallel p\langle \mathsf{let}\, x{:}c' = o\, \mathsf{in}\, t\rangle : \Theta, o{:}c}}{\Delta \vdash \nu(o{:}c).(o[c, M, F, \bot] \parallel p\langle \mathsf{let}\, x{:}c' = o\, \mathsf{in}\, t\rangle) : \Theta}$$

The premise $\Delta', o{:}c \vdash c \le c'$ follows from $\Delta, o{:}c \vdash c \le c'$ by weakening $\Delta' = \Delta, p{:}\langle T'\rangle$ to $\Delta', o{:}c$. Likewise by weakening, the premise $x{:}c'; \Delta', o{:}c \vdash t : T'$ follows from the corresponding premise in the derivation in (22). The left-most premise follows by T-NOBJ. The remaining rules for $\rightsquigarrow$ as well as for $\xrightarrow{\tau}$ work similarly.

Case: CALLI$_1$ with $a = p\langle \mathit{call}\ o.l(\vec{v}){:}T\rangle$?

We are given

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

The well-typedness assumption $\Xi \vdash C$ before the step can be written as $\Delta \vdash C : \Theta$. To be well-typed after the reduction step requires that all four components are well-typed. The update of the context from Ξ to $\acute{\Xi}$, as far as the typing is concerned, is given in Definition 12, for this case, in part 2 of the definition. The commitment part Θ is updated to $\acute{\Theta} = \Theta, p{:}\langle T\rangle, o{:}c, \vec{v}{:}\vec{T}$, the assumption context Δ remains unchanged in the step. Let's abbreviate $p{:}\langle T\rangle$ as Θ_p and the bindings $o{:}c, \vec{v}{:}\vec{T}$ as Θ_o. The (unchanged) part C is well-typed in the context $\Delta, \Theta_p, \Theta_o \vdash C : \Theta$ (by weakening). For $\Delta, \Theta, \Theta_o \vdash p\langle \mathsf{let}\, x{:}T = M.l(o)(\vec{v})\, \mathsf{in}\, \mathsf{release}(o); x\rangle : \Theta_p$ follows by T-NThread, two times T-Let, T-Release, and T-Var. The premise $\Xi \vdash a$ checks well-typedness of the incoming label $a = p\langle \mathit{call}\ o.l(\vec{v}){:}T\rangle?$ (cf. Definition 11, resp. rule LT-CalliI from Table 11). In particular, the premise assures that the types of the parameters match the declared ones for the method labeled l and that the return type equally corresponds to the one declared in the signature of the class implementing l. Note also that T is the type as declared for the local variable x which ultimately will contain the future value, and hence the the future reference is typed by $\langle T\rangle$ (as stipulated by Θ_p). The corresponding method body, say $M.l = m = \varsigma(s{:}c').\lambda(\vec{x}{:}\vec{T'}).t$ is assured to be well-typed correspondingly (by rule T-Class, which checks member implementations (methods and fields) against their type as declared in the class signature). I.e., the method implementation m is of type $\vec{T} \to T$, in particular also $\Delta, \Theta, \Theta_p \vdash m : \vec{T} \to T$. By preservation of well-typedness under substitution, that implies $\Delta, \Theta, \Theta_p \vdash M.l(o)(\vec{v}) : \vec{T} \to T$, as well; remember that $M.l(o)(\vec{v})$ abbreviates $t[o/s][\vec{v}/\vec{x}]$, i.e., the substitution of the formal parameters by the actual ones in the body t of the method m labeled l of the method suite M). The remaining parts of the post-configuration, the newly instantiated objects $o[c, M, F, \top]$ and $C(\vec{v})$, are well-typed by T-NObj and the fact that the corresponding classes they instantiate are assumed well-typed before the step. The well-typedness of the overall post-configuration follows then by instances of T-Par.

The remaining cases for preserving well-typedness work similarly.

Concerning preservation of the *connectivity* part of $\Xi \vdash C$, i.e., part 2 of Definition 20: we need to prove that the connectivity commitment context correctly over-approximates the *actual* connectivity as reflected in the fields of the objects and as captured by the notation $C \vdash o_1 \hookrightarrow o_2$ in Definition 20. For the proof of subject reduction, we weaken the formulation of the correctness invariant of Definition 2 slightly. We write $C \Vdash o_1 \hookrightarrow o_2$ if

1 $C \vdash o_1 \hookrightarrow o_2$ or
2 $C = C' \parallel p\langle t\rangle$ where t is a method body of object o_1 and contains o_2, or
3 $C = C' \parallel p\langle \mathit{call}\ o_1.l(\vec{v}){:}T\rangle!$ and where o_2 is contained in $\vec{v}$.

So instead of proving part 2 of Definition 20 directly we use the weaker $C \Vdash o_1 \hookrightarrow o_2$ as invariant instead. The result for the original, stronger formulation follows directly since object fields, being instance private, can be changed only by threads executing inside that object.

Now preservation of this relaxed invariant is straightforward: for $C \equiv \acute{C}$, preservation is trivial, since $\equiv$ only rearranges the representation of the component without changing the contents of the fields nor changing the code of the thread. For the confluent steps of the form $C \rightsquigarrow \acute{C}$, note that no fields of objects are read or updated, those steps work solely in a thread-local manner. Since E_θ does not change in the step (and threads may only "forget" references), the preservation is likewise straightforward. In case of NewT, issuing a method call, the step looks as follows: $p'\langle \mathsf{let}\, x{:}\langle T\rangle = o_2@l(\vec{v})\ \ \mathsf{in}\, t\rangle \rightsquigarrow \nu(p{:}\langle T\rangle).(p'\langle \mathsf{let}\, x{:}\langle T\rangle = p\ \ \mathsf{in}\, t\rangle \parallel p\langle call\ o_2.l(\vec{v}){:}T\rangle!)$. Assuming that the thread p' issuing the call executes a method body inside object o_1, we have by assumption $C \Vdash o_1 \hookrightarrow o_2$ and $C \Vdash o_1 \hookrightarrow v_i$ for all arguments (by clause 2 of the definition of the weakened invariant). After the step, $C \Vdash o_1 \hookrightarrow o_2$ and $C \Vdash o_1 \hookrightarrow v_i$ by clause 3 and potentially less actually connectivity by thread p'. The case for receiving a method call for role Call works similarly, the remaining cases are simpler. For non-confluent steps, one interesting case is FSet, which updates a field of an object:

$$o[c, M, F, L] \parallel p\langle \mathsf{let}\, x{:}T = o.l() := v\, \mathsf{in}\, t\rangle \xrightarrow{\tau} o[c, M, F.l := v, L] \parallel p\langle \mathsf{let}\, x{:}T = o\, \mathsf{in}\, t\rangle\ . \quad (23)$$

Since fields are instance-local, thread p executes "inside" o, i.e., $C \Vdash o \hookrightarrow v$ before the step (due to clause 2 of the weakened invariant) and likewise for the configuration after the step, with the updated field, due to clause 1.

The external steps $\Xi \vdash C \xrightarrow{a} \acute{\Xi} \vdash$ for part 2 of the lemma are immediate. For incoming calls where $ap\langle call\ o.l(\vec{v}){:}T\rangle?$ (by rule CalliI$_1$ resp. CalliI$_2$), $C \Vdash o \hookrightarrow v_i$ for all arguments by clause 2. By part 2 of Definition 14 E_Θ is updated by all pairs $o \hookrightarrow \vec{v}$, i.e., $\acute{E}_\Theta \vdash o \leftrightharpoons v_i$ for all arguments v_i, by reflexivity of $\leftrightharpoons$. The case for incoming get-labels works similarly. For incoming ν-labels in case 1 of Definition 14, corresponding to an instance of rule NewOI for object references resp. of rule NewTI for thread-identifiers/future references, the case is immediate, as only the assumption context E_Δ changes. Note that in both cases, the component C itself is unchanged, i.e., $\acute{C} = C$. The cases for outgoing communication work similarly: we show the case for GetO where the step is of the form $\Xi \vdash C \parallel p\langle v\rangle \xrightarrow{a} \acute{\Xi} \vdash C$ and where $a = p\langle get(v)\rangle!$. Before the step, we are given $E_\Delta \vdash o \hookrightarrow p$ (and $E_\Delta \Vdash o \hookrightarrow p$) indicating that object o references the future p, which indicates that o is the caller of the component-side method which has resulted in the evaluated future $p\langle v\rangle$. Furthermore we have $E_\Theta \vdash p \hookrightarrow o'$ which represents that the future has been evaluated in callee o'. After the step GetO, the future $p\langle v\rangle$ is removed from the component; analogously, in (the dual formulation of) part 1 of Definition 14, that is reflected by $\acute{E}_\Theta = E_\Theta \setminus p \hookrightarrow o'$. Since all other parts of the component are unchanged, $\acute{E}_\Delta \vdash C : \acute{E}_\Theta$, as required. □

Lemma 22 (Subject reduction). $\Xi \vdash C$ and $\Xi \vdash C \stackrel{s}{\Longrightarrow} \acute{\Xi} \vdash \acute{C}$ imply $\acute{\Xi} \vdash \acute{C}$.

Proof. By induction on the number of steps, using preservation under single steps from Lemma 21. □

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 learned about already known objects ("no surprise") in the assumptions can be phrased using the notion of conservative extension.

Definition 23 (Conservative extension). Given two contexts Ξ_Δ and $\acute{\Xi}_{\acute{\Delta}}$ where $\acute{\Delta}$ is an extension of Δ. Then we write $\Xi_\Delta \vdash \acute{\Xi}_{\acute{\Delta}}$ if $\acute{\Xi}_{\acute{\Delta}} \vdash n_1 \leftrightharpoons n_2$ implies $\Xi_\Delta \vdash n_1 \leftrightharpoons n_2$, for all n_1, n_2 with $\Delta \vdash n_1, n_2$.

Lemma 24 (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 12 and 14. □

Finally to the proof of soundness, that the open semantics is over-approximated by the legal traces. The proof is rather straightforward and the result may seem unsurprising as such, because the premises that governs the steps of the open semantics are partly re-used in the formalization of the legal traces. The reason why the legal traces and the open semantics fit together well and lead thus to a clean proof, however, rests on the careful "assumption-commitment" design of the steps of semantics: incoming steps depends *only* on assumptions about the environment and dually outgoing steps depend *only* on the given component (resp. on the commitment context for the legal traces). This clear dualism and separation of concern allows now a clean proof. Another important aspect in that context is that the legal traces (as a "symmetric abstraction" of the open semantics) are not just a sound over-approximation. Even without taking the connectivity (or well-typedness or all the other conditions into account) they would over-approximate the semantics. Important is, that when considering these conditions, in particular, restricting the traces by considering connectivity, the legal traces are still sound.

Theorem 25 (Soundness). If $\Xi_0 \vdash C$ and $\Xi_0 \vdash C \stackrel{s}{\Longrightarrow}$, then $\Xi_0 \vdash s : \mathit{trace}$.

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 22; 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: CalliI$_1$ and CalliI$_2$
Both cases are covered immediately by L-CalliI. 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 $a = p\langle call\ o.l(\vec{v}){:}T\rangle!$

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. □

6. Conclusion

This paper formally investigates the interface behavior of a typed, object-oriented language with *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 Denotational semantics are inherently defined in a compositional manner. For class-based, object-oriented languages for instance Cook [1989] develops such a semantics for a calculus with inheritance, based on fix-points, closures, and semantic domains; a similar approach is presented in Reddy [1988]. Another early denotational semantics of inheritance for a subsect of Smalltalk-80 is presented in Kamin [1988]. The semantics makes use of a *global* fixpoint with all classes present, which makes the semantics non-compositional.

Banerjee and Naumann [2005] are concerned with observable equivalence of classes resp. objects and substitutability in a setting of a class-based, object-oriented language with inheritance. Different from 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* Haynes [1984] (cf. also Mitchell [1986] Donahue [1979] Reynolds [1974] Reynolds [1983]). 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 Banerjee and Naumann [2005], 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 Banerjee and Naumann [2005], 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 Banerjee and Naumann [2005] by ownership restrictions. In contrast, our behavioral interface description takes a "black-box" view and considers two systems to be equivalent, if they exhibit the same traces at the interface. Also Poetzsch-Heffter and Schäfer [2007] 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 do not 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 Poetzsch-Heffter and Schäfer [2007]: to form a component, the corresponding set of classes must be "closed" in that all classes, methods, etc. *used* in the code of the component are actually *defined* in the component itself (which is "declaration complete" in the terminology of Poetzsch-Heffter and Schäfer [2007]; 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, Poetzsch-Heffter and Schäfer [2007] need 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 when given a trace (called history in Poetzsch-Heffter and Schäfer [2007]), 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 more recent Welsch and Poetzsch-Heffter [2013], which develops a full-abstract trace semantics for a class-based object-oriented calculus, again under the restriction of definition-completeness. In that work, the trace semantics is also used as formal basis for a verification method, using appropriate simulation relations.

Similarly, in the context of observable equivalence and a fully abstract semantics based on interface traces, Jeffrey and Rathke [2005] and Jeffrey and Rathke [2002] do not need to consider connectivity: in Jeffrey and Rathke [2002], because the language is object-*based*, i.e., without classes at all. Jeffrey and Rathke [2005], in contrast, avoids considering connectivity by introducing "packages" as units of composition, which, in the terminology of Poetzsch-Heffter and Schäfer [2007] are definition complete. Also Viswanathan [1998] 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. Koutavas and Wand [2007] develop a proof technique based on bisimulations to capture contextual equivalence for a class-based language (without and with inheritance). Besides observational equivalences based on traces, also *bisimulation* has been used, e.g., in Gordon and Rees [1996] for a functional first-order variant of Abadi and Cardelli's object caluli Abadi and Cardelli [1996] with subtyping. Similarly in Gordon et al. [1997], considering *imperative* objects; in the proto-type based setting object cloning is used instead of class inheritance. Breazu-Tannen et al. [1990] present a denotational semantics in the presence of subtyping ("coercions") achieving computational adequacy wrt. a given opertational semantics.

In the context of Java and JML, Ruby and Leavens [2000] are concerned with which interface information is needed to allow safe inheritance of methods (which they call the semantic fragile *subclassing* problem). In particular *downcalls* are problematic, i.e., the situation when a inherited method calls via a self-call the method of the sub-class overriding the corresponding method from the super-class.

The results here extends our previous work namely with *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., Ábrahám et al. [2009] 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. Ábrahám et al. [2006] 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. The general common-sense observation that loosely coupled systems entail a more compositional system description and especially simplify reasoning in a modular fashion is also expoited in Ahrendt and Dylla [2012], Din et al. [2012], Kurnia and Poetzsch-Heffter [2013] in the context of Hoare-style proof systems.

References

Martín Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag, 1996.

Erika Ábrahám, Frank S. de Boer, Marcello M. Bonsangue, Andreas Grüner, and Martin Steffen. Observability, connectivity, and replay in a sequential calculus of classes. In Marcello Bonsangue, Frank S. de Boer, Willem-Paul de Roever, and Susanne Graf, editors, *Proceedings of the Third International Symposium on Formal Methods for Components and Objects (FMCO 2004)*, volume 3657 of *Lecture Notes in Computer Science*, pages 296–316 (21 pages). Springer-Verlag, 2005. URL `http://www.ifi.uio.no/~msteffen/download/fa-fmco.pdf`.

Erika Ábrahám, Andreas Grüner, and Martin Steffen. Abstract interface behavior of object-oriented languages with monitors. In Roberto Gorrieri and Heike Wehrheim, editors, *FMOODS '06*, volume 4037 of *Lecture Notes in Computer Science*, pages 218–232 (15 pages). Springer-Verlag, 2006. . URL `http://www.springerlink.com/content/3365g26781740807`.

Erika Ábrahám, Andreas Grüner, and Martin Steffen. Abstract interface behavior of object-oriented languages with monitors. *Theory of Computing Systems*, 43(3-4): 322–361 (40 pages), December 2008. . URL `http://www.ifi.uio.no/~msteffen/download/07/monitors-tocs.pdf`.

Erika Ábrahám, Immo Grabe, Andreas Grüner, and Martin Steffen. Behavioral interface description of an object-oriented language with futures and promises. *Journal of Logic and Algebraic Programming*, 78(7):491–518 (28 pages), 2009. . URL `http://www.ifi.uio.no/~msteffen/download/09/futures.pdf`. Special issue with

selected contributions of NWPT'07. The paper is a reworked version of an earlier UiO Technical Report TR-364, Oct. 2007.

Samson Abramsky and Guy McCusker. Linearity, sharing, and state: a fully abstract game semantics for Idealized Algol with active expressions (extended abstract). volume 2. Birkhäuser, two volumes, 1997. Reprint of the paper which appeared in the Proceedings of the 1996 Workshop on Linear Logic, Vol. 3 of Electronic Notes in Theoretical Computer Science, 1996.

G. Agha and C. Hewitt. Concurrent programming using actors. In *Object-Oriented Concurrent Programming*, pages 37–53. MIT Press, 1987.

Wolfgang Ahrendt and Maximilian Dylla. A system for compositional verification of asynchronous objects. *Science of Computer Programming*, 77(12):1289 – 1309, 2012. ISSN 0167-6423. . URL `http://www.sciencedirect.com/science/article/pii/S0167642310001553`.

J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall Europe, Hemel Hempstead, Hertfordshire, 2nd edition, 1996.

Anindya Banerjee and David A. Naumann. Ownership confinement ensures representation independence for object-oriented programs. *Journal of the ACM*, 52(6):894–960, 2005. ISSN 0004-5411. .

Val Breazu-Tannen, , Carl A. Gunter, and André Ščedrov. Computing with coercions. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 44–61, 1990.

William Cook. *A Denotational Model of Inheritance*. PhD thesis, Brown University, 1989.

Pierre-Louis Curien. Definability and full abstraction. *Electronic Notes in Theoretical Computer Science*, 172:301–310, April 2007. ISSN 1571-0661. . URL `http://dx.doi.org/10.1016/j.entcs.2007.02.011`.

Frank S. de Boer, Dave Clarke, and Einar Broch Johnsen. A complete guide to the future. In R. de Nicola, editor, *Proceedings of Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Vienna, Austria.*, volume 4421 of *Lecture Notes in Computer Science*, pages 316–330. Springer-Verlag, 2007.

Crystal Chang Din, Johan Dovland, and Olaf Owe. Compositional reasoning about shared futures. In *Proceedings of SEFM'12*, volume 7504 of *Lecture Notes in Computer Science*, pages 94–108. Springer-Verlag, 2012. ISBN 978-3-642-33825-0.

J. Donahue. On the semantics of data type. *SIAM J. Computing*, 8:546–560, 1979.

Andrew D. Gordon and Gareth D. Rees. Bisimilarity for a first-order calculus of objects with subtyping. In *Proceedings of POPL '96*, pages 386–395. ACM, January 1996. Full version available as Technical Report 386, Computer Laboratory, University of Cambridge, January 1996.

Andrew D. Gordon, Paul D. Hankin, and Søren B. Lassen. Compilation and equivalence of imperative objects. In S. Ramesh and G. Sivakumar, editors, *Proceedings of FSTTCS '97*, volume 1346 of *Lecture Notes in Computer Science*, pages 74–87. Springer-Verlag, December 1997. Full version available as Technical Report 429, University of Cambridge Computer Laboratory, June 1997.

C.T̃. Haynes. A theory of data type representation independence. In Gilles Kahn, David

MacQueen, and Gordon D. Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 157–176. Springer-Verlag, 1984.

Alan Jeffrey and Julian Rathke. A fully abstract may testing semantics for concurrent objects. In *Proceedings of LICS '02*, pages 101–112. IEEE, Computer Society Press, July 2002.

Alan Jeffrey and Julian Rathke. Java Jr.: A fully abstract trace semantics for a core Java language. In Mooly Sagiv, editor, *Proceedings of ESOP 2005*, volume 3444 of *Lecture Notes in Computer Science*, pages 423–438. Springer-Verlag, 2005.

Samuel Kamin. Inheritance in Smalltalk-80: A denotational definition. In *ACM Conference on Programming Language Design and Implementation (PLDI) (Atlanta, GA)*. ACM, June 1988. In *SIGPLAN Notices* 23(7).

Vasileios Koutavas and Mitchell Wand. Reasoning about class behavior. In *Informal Workshop Record of FOOL 2007*, January 2007.

Ilham W. Kurnia and Arnd Poetzsch-Heffter. Verification of open concurrent object systems. In Frank S. de Boer, Marcello M. Bonsangue, Elena Giachino, and Reiner Hähnle, editors, *Proceedings of the 11th International Symposium on Formal Methods for Components and Objects, FMCO 2012*, volume 7866 of *Lecture Notes in Computer Science*. Springer-Verlag, 2013.

Leonid Mikhajlov and Emil Sekerinski. A study of the fragile base class problem. In *Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP'98), Brussels, Belgium*, volume 1445 of *Lecture Notes in Computer Science*, pages 355–354. Springer-Verlag, 1998.

John C. Mitchell. Representation independence and data abstraction. In *Thirteenth Annual Symposium on Principles of Programming Languages (POPL) (St. Peterburg Beach, FL)*, pages 263–276. ACM, January 1986.

M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima Series. Artima Press, 2011. ISBN 9780981531649. URL `http://books.google.no/books?id=ZNo8cAAACAAJ`.

Arnd Poetzsch-Heffter and Jan Schäfer. A representation-independent behavioral semantics for object-oriented components. In Marcello M. Bonsangue and Einar Broch Johnsen, editors, *FMOODS '07*, volume 4468 of *Lecture Notes in Computer Science*, pages 157–173. Springer-Verlag, June 2007.

Uday S. Reddy. Objects as closures: Abstract semantics of object-oriented languages. In *Symposium on* LISP *and Functional Programming (Snowbird, UT)*, pages 289–297. ACM, July 1988.

John Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Colloque sur la programmation (Paris, France)*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer-Verlag, 1974.

John Reynolds. Types, abstraction, and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523. IFIP, North-Holland, 1983.

Clyde Ruby and Gary T. Leavens. Safely creating correct subclasses without seeing superclass code. In *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) '00*, pages 208–228. ACM, 2000. In *SIGPLAN Notices*.

Alan Snyder. Encapsulation and inheritance in object-oriented programming languages.

In *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) '86 (Portland, Oregon)*, pages 38–45. ACM, 1986. In *SIGPLAN Notices* 21(11).

Raymie Stata and John. V. Guttag. Modular reasoning in the presence of subclassing. In *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) '95*, pages 200–214. ACM, 1995. In *SIGPLAN Notices*) 30(10).

Ramesh Viswanathan. Full abstraction for first-order objects with recursive types and subtyping. In *Proceedings of LICS '98*. IEEE, Computer Society Press, July 1998.

Yannik Welsch and Arnd Poetzsch-Heffter. A fully abstract trace-based semantics for reasoning about backward compatibility of class libraries. *Science of Computer Programming*, October 2013. ISSN 0167-6423. . URL `http://www.sciencedirect.com/science/article/pii/S0167642313002529`. Article in press. Available online 25. October 2013.