Behavioral interface description of an object-oriented language with futures and promises [⋆]

Erika Ábrahám [a] Immo Grabe [b] Andreas Grüner [b] Martin Steffen [c,*]

[a] *RWTH Aachen, Germany*

[b] *Christian-Albrechts-University Kiel, Germany*

[c] *University of Oslo, Norway*

Abstract

This paper formalizes the observable interface behavior of a concurrent, object-oriented language with futures and promises. The calculus captures the core of *Creol*, a language, featuring in particular asynchronous method calls and, since recently, first-class futures.

The focus of the paper are *open* systems and we formally characterize their behavior in terms of interactions at the interface between the program and its environment. The behavior is given by transitions between typing judgments, where the absent environment is represented abstractly by an assumption context. A particular challenge is the safe treatment of promises: The erroneous situation that a promise is fulfilled twice, i.e., bound to code twice, is prevented by a resource aware type system, enforcing linear use of the write-permission to a promise. We show subject reduction and the soundness of the abstract interface description.

Keywords: concurrent object-oriented languages, Creol, formal semantics, concurrency, futures and promises, open systems, observable behavior

1 Introduction

How to marry concurrency and object-orientation has been a long-standing issue; see e.g., [11] for an early discussion of different design choices. The thread-based model of concurrency, prominently represented by languages like *Java* and C$^\sharp$, has

[⋆] Part of this work has been supported by the NWO/DFG project Mobi-J (RO 1122/9-4), the DAAD-NFR exchange project Avabi, and by the EU-project IST-33826 *Credo: Modeling and analysis of evolutionary structures for distributed services.* For more information, see `http://credo.cwi.nl`.

[*] Corresponding author.
Email address: `msteffen@ifi.uio.no` (Martin Steffen).

been recently criticized, especially in the context of *component-based* software development. As the word indicates, components are (software) artifacts intended for composition, i.e., open systems, interacting with a surrounding environment. To compare different concurrency models for open systems on a solid mathematical basis, a semantic description of the interface behavior is needed, and this is what we provide in this work. We present an *open semantics* for the core of the *Creol* language [25,45], an object-oriented, concurrent language, featuring in particular asynchronous method calls and, since recently [27], first-class futures. An open semantics means, it describes the interface behavior of a program or a part of a program, interacting with its environment.

Futures and promises

A *future,* very generally, represents a result yet to be computed. It acts as a proxy for, or reference to, the delayed result from some piece of code (e.g., a method or a function body in an object-oriented, resp. a functional setting). As the consumer of the result can proceed its own execution until it actually needs the result, futures provide a natural, lightweight, and (in a functional setting) transparent mechanism to introduce parallelism into a language. Since their introduction in *Multilisp* [38,13], futures have been used in various languages like Alice ML [47,9,61], E [28], the ASP-calculus [18], Creol, and others. A *promise* is a generalization [1] insofar as the reference to the result on the one hand, and the code to calculate the result on the other, are not created at the same time; instead, a promise can be created and only later, after possibly passing it around, be bound to the code (the promise is *fulfilled*).

The notion of futures goes back to functional programming languages. In that setting, futures are annotations to side-effect-free expressions [2], that can be computed in parallel to the rest of the program. If some program code needs the result of a future, its execution blocks until the future's evaluation is completed and the result value is automatically fetched back (*implicit* futures). An important property of future-based functional programs is, that future annotations do not change the functionality: the observable behavior of an annotated program equals the observable behavior of its non-annotated counterpart. This property no longer is assured in the object-oriented setting.

[1] The terminology concerning futures, promises, and related constructs is not too consistent in the literature. Sometimes, the two words are used as synonyms. Interested in the observable differences between futures and promises, we distinguish the concepts and thus follow the terminology as used e.g., in λ_{fut}, Alice ML, and the definition given in Wikipedia.
[2] Though in e.g. *Multilisp* also expressions with side-effects can be computed in parallel, but still under the restriction that the observable behavior equals that of the sequential counterpart.

Interface behavior

An open program, in this setting, interacts with its environment via message exchange. Besides message passing, of course, different communication and synchronization mechanisms exists (shared variable concurrency, multi-cast, black-board communication, publish-and-subscribe and many more). We concentrate here, however, on basic message passing using asynchronous method calls. In that setting, the interface behavior of an open program C can be characterized by the set of all those message sequences (traces) t, for which there *exists* an environment E such that C and E exchange the messages recorded in t. Thereby we abstract away from any concrete environment, but consider only environments that are compliant to the language restrictions (syntax, type system, etc.). Consequently, interactions are not arbitrary traces $C \overset{t}{\Longrightarrow}$; instead we consider behaviors $C \parallel E \underset{\bar{t}}{\overset{t}{\Longrightarrow}} \acute{C} \parallel \acute{E}$ where E is a *realizable* environment and trace $\bar{t}$ is complementary to t, i.e., each input is replaced by a matching output and vice versa. The notation $C \parallel E$ indicates that the component C runs in parallel with its environment or observer E. To account for the abstract environment ("there exists an E s.t. ..."), the open semantics is given in an *assumption-commitment* way:

$$\Delta \vdash C : \Theta \overset{t}{\Longrightarrow} \acute{\Delta} \vdash \acute{C} : \acute{\Theta} \ ,$$

where Δ (as an abstract version of E) contains the *assumptions* about the environment, and dually Θ the *commitments* of the component. Abstracting away also from C gives a language characterization by the set of all possible traces between any component and any environment.

Such a behavioral interface description is relevant and useful for the following reasons. 1) The set of possible traces given this way is more restricted (and realistic) than the one obtained when ignoring the environments. When reasoning about the trace-based behavior of a component, e.g., in compositional verification, with a more precise characterization one can carry out stronger arguments. 2) When using the trace description for *black-box testing*, one can describe test cases in terms of the interface traces and then synthesize appropriate test drivers from it. Clearly, it makes no sense to specify impossible interface behavior, as in this case one cannot generate a corresponding tester. 3) A representation-independent behavior of open programs paves the way for a compositional semantics, a two-level semantics for the nested composition of program components. It allows furthermore optimization of components: only if two components show the same external, observable behavior, one can replace one for the other without changing the interaction with any environment. 4) The formulation gives insight into the semantic nature of the language, here, the externally observable consequences of futures and promises. This helps to compare alternatives, e.g., the Creol concurrency model with *Java*-like threading.

Results

The paper formalizes the abstract interface behavior for concurrent object-oriented languages with futures and promises. The contributions are the following.

Concurrent object calculus with futures and promises We formalize a class-based concurrent language featuring futures and promises. The formalization is given as a typed, imperative object calculus in the style of [1] resp. one of its concurrent extensions. The operational semantics for components distinguishes unobservable component-internal steps from external steps which represent observable component-environment interactions. We present the semantics in a way that facilitates comparison with *Java*'s multi-threading concurrency model, i.e., the operational semantics is formulated so that the multi-threaded concurrency as (for instance) in *Java* and the one here based on futures are represented similarly.

Linear type system for promises The calculus extends the semantic basis of *Creol* as given for example in [27] with promises. Promises can refer to a computation with code bound to it later, where the binding is done at most once. To guarantee such a *write-once* policy when passing around promises, we refine the type system introducing two type constructors

$$[T]^{+-} \quad \text{and} \quad [T]^{+} \ .$$

The first one represents a reference to a promise that can still be written (and read) with result type T, the second one where only a *read*-permission is available. The write permission constitutes a resource which is consumed when the promise is fulfilled. The resource-aware type system is therefore formulated in a *linear* manner wrt. the write permissions. Linear type sytems [67] or linear logics [35] are, roughly speaking, instances of so-called sub-structural logics resp. type systems. In constrast to ordinary such derivation systems, where a hypothesis can be used as many times as needed for carrying out a proof, derivation systems built upon linear use of assumptions work differently: using an assumption in a proof step "*consumes*" it. That feature allows in a natural way to reason about "resources": In our setting, the write-permission is such a resource, and using the corresponding type to derive well-typedness of one part of the program consumes that right such that it is not longer available for type-checking the rest of the program, assuring the intended write-once discipline. The type system resembles in intention the one in [55] for a functional calculus with references. Our work is more general, in that it tackles the problem in an object-oriented setting (which, however, conceptually does not pose much complications), and, more importantly, in that we do not consider closed systems, but open components. Also this aspect of openness is not dealt with in [27]. Additionally, the type system presented here is simpler than in [55], as it avoids the representation of the promise-concept by so-called *handled futures*.

Soundness of the abstractions We show soundness of the abstractions, which includes

- *subject reduction,* i.e., preservation of well-typedness under reduction. Subject reduction is not just proven for a closed system (as usual), but for an open system interacting with its environment. Subject reduction implies furthermore:
- *absence of run-time errors* like "message-not-understood", also for open systems.
- *soundness* of the interface behavior characterization, i.e., all possible interaction behavior is included in the abstract interface behavior description.
- for promises: absence of *write-errors,* i.e. the attempt to fulfill a promise twice.

The paper is organized as follows. Section 2 defines the syntax, the type system, and the operational semantics, split into an internal one, and one for the interface behaviour of open systems. Section 3 describes the interface behavior. Section 4 concludes with related and future work. For more details see [2]. There is a notation index at the end of the paper.

2 Calculus

This section presents the calculus, based on a version of the *Creol*-language with first-class futures [27] and extended with promises. It is a concurrent variant of an imperative, object-calculus in the style of the calculi from [1]. Our calculus covers first-class futures, which can be seen as a generalization of asynchronous method calls and promises.

We start with the abstract syntax in Section 2.1. After discussing the type system in Section 2.2, we present the operational semantics in Section 2.3.

2.1 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 latter are not found in a program written by the user, but generated at run-time by the rules of the operational semantics.

The basic syntactic category of names n, which count among the values v, represents references to classes, to objects, and to futures/promises. To facilitate reading, we allow ourselves to write o and its syntactic variants for names referring to objects, c for classes, and n when being unspecific. Technically, the disambiguation

C	$::=$	$\mathbf{0} \mid C \parallel C \mid \underline{\nu(n{:}T).C} \mid n[(O)] \mid \underline{n[n,F,L]} \mid \underline{n\langle t\rangle} \mid \underline{n\langle\bullet\rangle}$	component
O	$::=$	F,M	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	$::=$	$\varsigma(n{:}T).\lambda().v \mid \varsigma(n{:}T).\lambda().\bot_{n'}$	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.
	$\mid$	$\mathsf{promise}\ T \mid \mathsf{bind}\,n.l(\vec{v}) : T \hookrightarrow n \mid v.l() \mid v.l := \varsigma(s{:}n).\lambda().v$	
	$\mid$	$\mathsf{new}\,n \mid \mathsf{claim@}(n,n) \mid \underline{\mathsf{get@}n} \mid \mathsf{suspend}(n) \mid \underline{\mathsf{grab}(n)} \mid \underline{\mathsf{release}(n)}$	
v	$::=$	$x \mid n \mid ()$	values
L	$::=$	$\bot \mid \top$	lock status

Table 1
Abstract syntax

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 $n\langle t\rangle$, where t is the code being executed and n the name of the thread. In the given setting, threads are always promises (with the exception of initial threads, see Section 2.3), with their name being the reference under which the result value of t, if any[3], will be available. A class $c[(O)]$ carries a name c and defines its methods and fields in O. An object $o[c,F,L]$ with identity o keeps a reference to the class c it instantiates, 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) or not (in which case the lock is free). The symbols $\top$, resp., $\bot$, indicate that the lock is taken, resp., free. From the three kinds of entities at component level —threads $n\langle t\rangle$, classes $c[(O)]$, and objects $o[c,F,L]$— only the threads are *active*, executing entities, being the target of the reduction rules. The objects, in contrast, store the state in their fields or instance variables, whereas the classes are constant entities specifying the methods.

[3] There will be no result value in case of non-terminating methods.

The named threads $n\langle t\rangle$ 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.[4] During execution, $n\langle t\rangle$ contains in t the currently running code of a method body. When evaluated, the thread is of the form $n\langle v\rangle$ and the value can be accessed via n, 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 crucial difference between Java's multi-threading concurrency model and Creol's active objects model used here is the way method calls are issued at the caller site. In Java and similar languages, method calls are *synchronous* in the sense that the calling activity blocks to wait for the return of the result and thus the control is transferred to the callee. Here, method calls are issued *asynchronously,* i.e., the calling thread continues executing and the code of the method being called is computed concurrently in a new thread located in the callee object. In that way, a method call never transfers control from one object, the caller, to another one, the callee. In other words, no thread ever crosses the boundaries of an object, which means, the boundaries of an object are at the same time boundaries of the threads and thus, the objects are at the same time units of concurrency. Thus, the objects are harnessing the activities and can be considered as bearers of the activities. This is typical for object-oriented languages based on *active objects*.

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 promises. The scope is dynamic, i.e., when the name is communicated by message passing, it is enlarged.

Besides components, the grammar specifies the lower level syntactic constructs, in particular, methods, expressions, and (unnamed) threads, which are basically sequences of expressions. A method $\varsigma(s{:}T).\lambda(\vec{x}{:}\vec{T}).t$ provides the method body t abstracted over the ς-bound "self" parameter s and the formal parameters $\vec{x}$. For uniformity, fields are represented as methods without parameters (with the exception of the standard self-parameter). The "body" of a field is either a value or yet

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

undefined. Note that the methods are stored in the classes but the fields are kept in the objects, of course. In freshly created objects, the lock is free, and all fields carry the undefined reference $\perp_c$, where class name c is the (return) type of the field.

We use f for instance variables or fields and $l = \varsigma(s{:}T).\lambda().v$, resp. $l = \varsigma(s{:}T).\lambda().\perp_c$ for field variable definition. Field access is written as $v.l()$ and field update as $v'.l := \varsigma(s{:}T).\lambda().v$. By convention, we abbreviate the latter constructs by $l = v$, $l = \perp_c$, $v.l$, and $v'.l := v$. Note that the construct $v.l()$ is used for field access only, but not for method invocation. We will also use $v_\perp$ to denote either a value v or a symbol $\perp_c$ for being undefined. Note that the syntax does not allow to set a field back to undefined. Direct access (read or write) to fields across object boundaries is forbidden by convention, and we do not allow method update. Instantiation of a new object from class c is denoted by $\mathsf{new}\, c$.

Expressions especially include syntax to deal with promises and futures. The expression $\mathsf{promise}\, T$ creates a new promise, i.e., a reference or name for a result yet to come. At the point of creation, only the name exists, but no code has been determined and attached to the reference to calculate the result. Binding code to the promise is done by $\mathsf{bind}\, o.l(\vec{v}) : T \hookrightarrow n$, stipulating that the eventual result is calculated using the method l of object o with actual parameters $\vec{v}$. Executing the binding operation is also known as *fulfilling* the promise. Some languages do not allow to *independently* create a name for the eventual result, i.e., creation and binding are done inseparately by one single command. In that situation, one does not speak of promises, but (just) of futures, even if in the literature, sometimes no distinction is drawn between futures and promises. In a certain way, futures and promises can be seen as two different roles of a reference n: the promise-role means, a client can *write* to the name using the bind-operation, and the future-role represents the possibility to *read* back an eventual result using the reference. In this way, we will use both the term future and promise when referring to the same reference, depending on the role it is playing when used.

The expression $\mathsf{bind}\, o.l(\vec{v}) : T \hookrightarrow n$ binds a *method body* to the promise n. Thus, there is a close connection to asynchronous method calls, central to Creol's concurrency model. Indeed, in comparison with [27], which introduces the concept of futures (but not promises) into Creol, asynchronous calls are syntactic sugar for creating a new promise and immediately binding $o.l(\vec{v})$ to it. This behaves as an asynchronous method call, as the one creating the promise and executing the bind can continue without being blocked waiting for 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 assuring 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 ($\perp$), 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, when succeeding to grab the lock via $\mathsf{grab}(o)$. The user is not allowed to directly manipulate the object locks. Thus, both expressions belong to the run-time syntax, underlined in Table 1, and are only generated and handled by the operational semantics as auxiliary expression at run-time. Instead of using directly grab and release, the lock-handling is done automatically when executing a method body: before starting to execute, 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 $\mathsf{claim}@(n,o)$ is the attempt to obtain the result of a method call from the future n while in possession of the lock of object o. There are two possibilities in that situation: either the value of the future has already been determined, i.e., the method calculating the result has terminated, in which case the client just obtains the value *without* loosing its own lock. In the alternative case, where the value is not yet determined, the client trying to read the value gives up its lock via release and continues executing only after the requested value has been determined (using get to read it) and after it has re-acquired the lock. Unlike claim, the get-operation is not part of the user-syntax. Both expressions are used to read back the value from a future and 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. This behavior is sketched in Figure 1. Note the order in which get and grab are executed after releasing the lock: the value is read in via get *before* the lock has actually been re-acquired! That this order is acceptable rests on the fact that a future, once evaluated, does not change the value later and reading the value in by itself has no side-effect. Reversing the order —first re-aquiring the lock and afterwards checking for availability of the future's value— would result in equivalent behavior but amount to busy waiting. 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-command only refer to the self-parameter *self*, i.e., they are written $\mathsf{claim}@(n,self)$ and $\mathsf{suspend}(self)$.

Before continuing with the type system, let us explain how and why we exclude a specific potential deadlock situation in the semantics of the claim statement (though the language does not generally exclude the presence of deadlocks, i.e., it is possible to write a deadlocking program in the language). Remember that if a thread is about to execute a claim statement in an object, it always owns the object's lock. If the claimed result is not yet available, then the claiming thread blocks. During blocking, if we would not release the lock previously, no other thread could execute in the object, since it would require the object's lock. Consequently, if the computation of the claimed result needs execution in the object, the threads would deadlock. Such deadlocks could not be easily excluded syntactically, since release and grab are only auxiliary syntax, i.e., they cannot be used to write programs, and we do not support checking if a thread already finished its computation. Thus we release the lock before blocking, i.e., waiting for the claimed result, and re-grab the

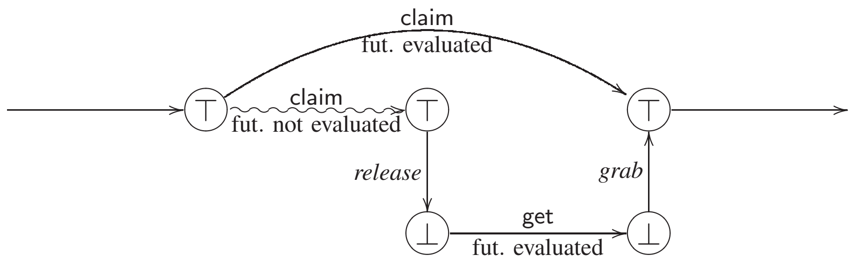

Fig. 1. Claiming a future

lock after the thread got the result.

2.2 *Type system*

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

$$\begin{aligned} T &::= B \mid \mathsf{Unit} \mid [T]^{+-} \mid [T]^{+} \mid [l{:}U,\ldots,l{:}U] \mid [(l{:}U,\ldots,l{:}U)] \mid n \\ U &::= T \times \ldots \times T \to T \end{aligned}$$

Besides base types B (left unspecified; typical examples are booleans, integers, etc.), Unit is the type of the unit value $()$. Types $[T]^{+-}$ and $[T]^{+}$ represent the reference to a future which will return a value of type T, in case it eventually terminates. $[T]^{+-}$ indicates that the promise has not yet been fulfilled, i.e., it represents the write-permission to a promise, which implies read-permission at the same time. $[T]^{+}$ represents read-only permission to a future. The read/write capability is more specific than read-only, which is expressed by the (rather trivial) subtyping relation generated by $[T]^{+-} \leq [T]^{+}$, accompanied by the usual subsumption rule. Furthermore, $[\_]^{+}$ acts monotonely, and $[\_]^{+-}$ invariantly wrt. subtyping. When not interested in the access permission, we just write $[T]$.

The name of a class serves as the type for its instances. We need as auxiliary type constructions (i.e., not as part of the user syntax, but to formulate the type system) the type or interface of unnamed objects, written $[l_1{:}U_1,\ldots,l_k{:}U_k]$ and the interface type for classes, written $[(l_1{:}U_1,\ldots,l_k{:}U_k)]$. We allow ourselves to write $\vec{T}$ for $T_1 \times \ldots \times T_k$ etc. where we assume that the number of arguments match in the rules, and write $\mathsf{Unit} \to T$ 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 on the level of components, i.e., global configurations, and secondly one for their syntactic subconstituents (cf. also the two different levels in the abstract syntax of Table 1).

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

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

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. Both contexts constitute the static interface information. A pair Δ and Θ of assumption and commitment context with disjoint domains is called *well-formed*.

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). Also the treatment of the assumption context requires care wrt. the write permissions. 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 $C_1 \parallel C_2$. This, however, does not apply to the write-permissions: if $C_1 \parallel C_2$ do have write permission on a promise n, which resides in the environment of $C_1 \parallel C_2$, this is represented as $n{:}[T]^{+-}$ in the assumptions of that parallel

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

$$\frac{;\Delta, c{:}T \vdash [\![O]\!] : T}{\Delta \vdash c[\![O]\!] : (c{:}T)}\ \text{T-NCLASS} \qquad \frac{;\Delta \vdash c : [\![(T_F, T_M)]\!] \quad ;\Delta, o{:}c \vdash [F] : [T_F]}{\Delta \vdash o[c, F, L] : (o{:}c)}\ \text{T-NOBJ}$$

$$\frac{;\Delta, n{:}[T]^{+} \vdash t : T}{\Delta \vdash n\langle t\rangle : (n{:}[T]^{+})}\ \text{T-NTHREAD} \qquad \frac{}{\Delta \vdash n\langle\bullet\rangle : (n{:}[T]^{+-})}\ \text{T-NTHREAD}'$$

$$\frac{\Delta' \leq \Delta \quad \Theta \leq \Theta' \quad \Delta \vdash C : \Theta}{\Delta' \vdash C : \Theta'}\ \text{T-SUB}$$

Table 2
Typing (component level)

composition. Due to the linear nature of the write permission, however, the binding $n{:}[T]^{+-}$ can occur only in the assumptions of either C_1 or of C_2 in the two premises of T-PAR. In other words, the assumption context of $C_1 \parallel C_2$ must be split as far as the write permissions to promises are concerned. To capture this intuition, we define:

Definition 2.1 *Let the symmetric operation $\oplus$ on well-formed name contexts be defined as follows:*

$$\begin{array}{lll}
\mathbf{0} \oplus \Delta & = \Delta & \\
n{:}[T]^{+}, \Delta_1 \oplus n{:}[T]^{+-}, \Delta_2 & = n{:}[T]^{+-}, (\Delta_1 \oplus \Delta_2) & \\
n{:}T, \Delta_1 \oplus n{:}T, \Delta_2 & = n{:}T, (\Delta_1 \oplus \Delta_2) & T \neq [T']^{+-} \textit{ for some } T' \\
\Delta_1 \oplus \Delta_2 & = \text{undefined} & \textit{else}
\end{array}$$

We omit symmetric rules (e.g. for $\Delta \oplus \mathbf{0}$). The contexts are considered as unordered, i.e., $n{:}T, \Delta$ does not mean, the binding $n{:}T$ occurs leftmost in a "list".

In combination with the rest of the rules (in particular, rule T-BIND below in Table 4), this assures that a promise cannot be fulfilled by the component and the environment at the same time.

The ν-binder hides the bound object or the name of the future 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. Let-bound variables are *stack* allocated and checked in a stack-organized variable context Γ (see Tables 3 and 4 below). Object names created by new and future/promise names created by promise are *heap* allocated and thus checked in a "parallel" context (cf. again the assumption-commitment rule T-PAR). The rules for named classes introduce the name of the class and its type into the commitment (cf. T-NCLASS). The code $[(O)]$ of the class $c[(O)]$ is checked in an assumption context where the name of the class is available. Note also that the premise of T-NCLASS (like those of T-NOBJ and T-NTHREAD) is not covered by the rules for type checking at the component level, but by the rules for the lower level entities (in this particular case, by rule T-OBJ from Table 3). The judgments there use as assumption not just a name context, but additionally a stack-organized, variable 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 $;\Delta, c{:}T \vdash [(O)] : T$; similar for the premises of T-NOBJ and T-NTHREAD. An instantiated object will be available in the exported context Θ by rule T-NOBJ.

As we will see in the following section, promises, that are not yet fulfilled, are present in the configuration as thread entities $n\langle\bullet\rangle$; their type $[T]^{+-}$ can be derived by rule T-NTHREAD$'$. Fulfilled promises $n\langle t\rangle$ are treated by rule T-NTHREAD, where the type $[T]^+$ of the future reference n is matched against the result type T of thread t. As n is already fulfilled, its type exports read-permission, only. As t may refer to n, it is checked in the premise by Δ extended by the appropriate binding $n{:}[T]^+$. 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. Note, however, that each named object has exactly one type, namely its class.

Definition 2.2 (Subtyping) *The relation $\leq$ on types is defined as identity for all types except for $[T]^{+-} \leq [T]^+$ (mentioned above) and object interfaces, where we have:*

$$[(l_1{:}U_1,\ldots,l_k{:}U_k,l_{k+1}{:}U_{k+1},\ldots)] \leq [(l_1{:}U_1,\ldots l_k{:}U_k)]\ .$$

For well-formed name contexts Δ_1 and Δ_2 , we define in abuse of notation $\Delta_1 \leq \Delta_2$, if Δ_1 and Δ_2 have the same domain and additionally $\Delta_1(n) \leq \Delta_2(n)$ for all names n.

The definition is applied, of course, also to name contexts Θ, used for the commitments. The relations $\leq$ are obviously reflexive, transitive, and antisymmetric.

Next we formalize the typing for objects and threads and their syntactic sub-constituents. Again, the treatment of the write-permissions requires care: The capability to write to a promise is consumed by the bind-operation as it should be done only once. This is captured by a *linear* type system where the execution of a thread or an expression may change the involved types. The judgments are of the form

$$\Gamma;\Delta \vdash e : T :: \acute{\Gamma},\acute{\Delta}, \tag{2}$$

where the change from Γ and Δ to $\acute{\Gamma}$ and $\acute{\Delta}$ reflects the potential consumption of write permissions when executing e. The consumption is only potential, as the type system statically overapproximates the run-time behavior, of course. The typing is given in Tables 3 and 4. For brevity, we write $\Delta;\Gamma \vdash e : T$ for $\Delta;\Gamma \vdash e : T :: \acute{\Gamma},\acute{\Delta}$, when $\acute{\Gamma} = \Gamma$ and $\acute{\Delta} = \Delta$. 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. Apart from the technicalities, treating the write capabilities in a linear fashion is straightforward: one must assure that the corresponding capability is available at most once in the program and is not duplicated when passed around. A promise is no longer available for writing when bound to a variable using the let-construct, or when handed over as argument to a method call or a return.

Classes, objects, and methods resp. fields have no effect on Δ (see rules T-CLASS, T-OBJ, T-MEMB, and T-UNDEF). Note that especially in T-MEMB, the name context Δ does not change. This does *not* mean, that a method cannot have a side-effect by fulfilling promises, but they are not part of the check of the method *declaration* here. Rule T-CLASS is the introduction rule for class types, the rule of instantiation of a class T-NEWC requires reference to a class-typed name. In the rules T-MEMB

$$\frac{\Gamma;\Delta \vdash c : [\!(l_1{:}U_1,\ldots,l_k{:}U_k)\!] \quad \Gamma;\Delta \vdash m_i : U_i \quad m_i = \varsigma(s_i{:}c).\lambda(\vec{x}_i{:}\vec{T}_i).t_i}{\Gamma;\Delta \vdash [\!(l_1 = m_1,\ldots,l_k = m_k)\!] : c}\ \text{T-CLASS}$$

$$\frac{\Gamma;\Delta \vdash c : [\!(l_1{:}U_1,\ldots,l_k{:}U_k)\!] \qquad \Gamma;\Delta \vdash f_i : U_i \qquad f_i = \varsigma(s_i{:}c).\lambda().v_\perp}{\Gamma;\Delta \vdash [l_1 = f_1,\ldots,l_k = f_k] : c}\ \text{T-OBJ}$$

$$\frac{\Gamma,\vec{x}{:}\vec{T};\Delta,s{:}c \vdash t : T' :: \acute{\Gamma};\acute{\Delta} \quad \Gamma;\Delta \vdash c : T \quad T = [\!(\ldots,l{:}\vec{T} \to T',\ldots)\!]}{\Gamma;\Delta \vdash \varsigma(s{:}c).\lambda(\vec{x}{:}\vec{T}).t : T.l}\ \text{T-MEMB}$$

$$\frac{\Gamma;\Delta,s{:}c \vdash c : [\!(\ldots,l : \mathsf{Unit} \to c',\ldots)\!]}{\Gamma;\Delta \vdash \varsigma(s{:}c).\lambda().\perp_{c'} : c'}\ \text{T-UNDEF}$$

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

$$\frac{\Gamma_1;\Delta_1 \vdash e : T_1 :: \Gamma_2;\Delta_2 \qquad \Gamma_2,x{:}T_1;\Delta_2 \vdash t : T_2 :: \Gamma_3;\Delta_3}{\Gamma_1;\Delta_1 \vdash \mathsf{let}\, x{:}T_1 = e\, \mathsf{in}\, t : T_2 :: \Gamma_3;\Delta_3}\ \text{T-LET}$$

$$\frac{\Gamma_1;\Delta_1 \vdash v_1 : T_1 \quad \Gamma_1;\Delta_1 \vdash v_2 : T_1 \quad \Gamma_1;\Delta_1 \vdash e_1 : T_2 :: \Gamma_2;\Delta_2 \quad \Gamma_1;\Delta_1 \vdash e_2 : T_2 :: \Gamma_2;\Delta_2}{\Gamma_1;\Delta_1 \vdash \mathsf{if}\, v_1 = v_2\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 : T_2 :: \Gamma_2;\Delta_2}\ \text{T-COND}$$

$$\frac{\begin{array}{l}\Gamma_1;\Delta_1 \vdash v : c \quad \Gamma_1;\Delta_1 \vdash c : [\!(\ldots,l{:}\mathsf{Unit} \to T,\ldots)\!] \\ \Gamma_1;\Delta_1 \vdash e_1 : T_2 :: \Gamma_2;\Delta_2 \quad \Gamma_1;\Delta_1 \vdash e_2 : T_2 :: \Gamma_2;\Delta_2\end{array}}{\Gamma_1;\Delta_1 \vdash \mathsf{if}\, \mathsf{undef}(v.l())\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 : T_2 :: \Gamma_2;\Delta_2}\ \text{T-COND}_\perp$$

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

Table 3
Typing (objects and threads)

and T-FUPDATE we use the meta-mathematical notation $T.l$ to pick the type in T associated with label l, i.e., $T.l$ denotes U, when $T = [\ldots,l{:}U,\ldots]$ and analogously for $T = [\!(\ldots,l{:}U,\ldots)\!]$. Rules T-CLASS and T-OBJ check the definition of classes resp., of objects against the respective interface type $[\!(l_1{:}U_1,\ldots,l_k{:}U_k)\!]$. Note that the type of the self-parameter must be identical to the name of the class, the method resides in. The premises of rule T-MEMB checks the method body in the context Γ appropriately extended with the formal parameters x_i, resp. the context Δ extended by the ς-bound self-parameter (s in the rule). T-UNDEF works similarly, treating the case of an uninitialized field. The terminated expression stop and the unit value do not change the capabilities (cf. rules T-STOP and T-UNIT). Note that stop has any type (cf. rule T-STOP) reflecting the fact that control never reaches the point *after* stop. Further constructs without side effects are the three expressions to manipulate the monitor locks (suspension, lock grabbing, and lock release), object instantiation (T-NEWC), and field update. Wrt. field update in rule T-FUPDATE, the reason why the update has no effect on the contexts is that we do not allow fields to carry

$$\frac{}{\Gamma;\Delta \vdash \mathsf{promise}\ T : [T]^{+-}}\ \textrm{T-P}\textsc{rom}$$

$$\frac{\Gamma;\Delta \vdash n : [T]^{+} \qquad \Gamma;\Delta \vdash o{:}c}{\Gamma;\Delta \vdash \mathsf{claim@}(n,o) : T}\ \textrm{T-C}\textsc{laim} \qquad \frac{\Gamma;\Delta \vdash n : [T]^{+}}{\Gamma;\Delta \vdash \mathsf{get@}n : T}\ \textrm{T-G}\textsc{et}$$

$$\frac{\Gamma(x) = T \qquad \acute{\Gamma} = \Gamma\backslash x : T}{\Gamma;\Delta \vdash x : T :: \acute{\Gamma};\Delta}\ \textrm{T-V}\textsc{ar} \qquad \frac{\Delta(x) = T \qquad \acute{\Delta} = \Delta\backslash n : T}{\Gamma;\Delta \vdash n : T :: \Gamma;\Delta'}\ \textrm{T-N}\textsc{ame}$$

$$\frac{\Gamma;\Delta,n{:}[T]^{+} \vdash o : c \quad \Gamma;\Delta,n{:}[T]^{+} \vdash c : [\!(\ldots,l{:}\vec{T} \to T,\ldots)\!] \quad \Gamma;\Delta,n{:}[T]^{+} \vdash \vec{v} : \vec{T} \quad \acute{\Gamma};\acute{\Delta} = \Gamma;\Delta\backslash(\vec{v} : \vec{T})}{\Gamma;\Delta,n : [T]^{+-} \vdash \mathsf{bind}\, o.l(\vec{v}) : T \hookrightarrow n : [T]^{+} :: \acute{\Gamma};\acute{\Delta},n{:}[T]^{+}}\ \textrm{T-B}\textsc{ind}$$

$$\frac{\Delta \vdash o : c}{\Gamma;\Delta \vdash \mathsf{suspend}(o) : \mathsf{Unit}}\ \textrm{T-S}\textsc{uspend} \qquad \frac{\Delta \vdash o : c}{\Gamma;\Delta \vdash \mathsf{grab}(o) : \mathsf{Unit}}\ \textrm{T-G}\textsc{rab} \qquad \frac{\Delta \vdash o : c}{\Gamma;\Delta \vdash \mathsf{release}(o) : \mathsf{Unit}}\ \textrm{T-R}\textsc{elease}$$

$$\frac{\Gamma_1;\Delta_1 \vdash t : T :: \Gamma_2;\Delta_2 \qquad T \leq T'}{\Gamma_1;\Delta_1 \vdash t : T' :: \Gamma_2;\Delta_2}\ \textrm{T-S}\textsc{ub}$$

Table 4
Typing (objects and threads)

a type of the form $[T]^{+-}$. This effectively prevents the passing around of write-permissions via fields. The rule T-LET for let-bindings introduces a local scope. The change from Δ_1 to Δ_2 and further from Δ_2 to Δ_3 (and analogously for the Γs) reflects the sequential evaluation strategy: first e is evaluated and afterwards t. For conditionals, both branches must agree on their pre- and post Δ-contexts, which typically means, over-approximating the effect by taking the upper bound on both as combined effect. Note that the comparison of the values in T-COND resp. the check for definedness in T-COND$_\perp$ has no side-effect on the contexts. The rule for testing for definedness using undef (not shown) works analogously.

Table 4 deals with futures, promises, and especially the linear aspect of consuming and transmitting the write-permissions. The claim-command fetches the result value from a future; hence, if the reference n is of type $[T]^{+}$, the value itself carries type T (cf. rule T-CLAIM). The rule T-GET for get works analogously.

The expression promise$T$ creates a new promise, which can be read or written and is therefore of type $[T]^{+-}$. Note, however, that the context Δ does *not* change. The reason is that the new name created by promise is hidden by a ν-binder immediately after creation and thus does not immediately extend the Δ-context (see the reduction rule PROM below). The binding of a thread t to a promise n is well-typed if the type of n still allows the promise to be fulfilled, i.e., n is typed by $[T]^{+-}$ and not just $[T]^{+}$. The expression claim dereferences a future, i.e., it fetches a value of type T from the reference of type $[T]^{+}$. Otherwise, the expression has no effect on Δ, as reading can be done arbitrarily many times. As an aside: in rule T-CLAIM, the type of o is not checked, as by convention, the claim-statement must be used in the form claim@(n, self) in the user syntax, where *self* is the self-parameter of the surrounding methods. Reduction then preserves well-typedness so a re-check here is not needed. Similar remarks apply to the remaining rules. The treatment of get is analogous (cf. rules T-CLAIM and T-GET). For T-BIND, handing over a promise with read/write permissions as an actual parameter of a method call, the caller loses the right to fulfill the promise. Of course, the caller can only pass the promise to a method which assumes read/write permissions, if itself has the write permission. The loss of the write-permission is specified by setting $\acute{\Delta}$ and $\acute{\Gamma}$ to $\Delta \backslash \vec{v} : \vec{T}$ resp. to $\Gamma \backslash \vec{v} : \vec{T}$. The *difference*-operator $\Delta \backslash n : [T]^{+-}$ removes the *write*-permission for n from the context Δ. In T-BIND, the premise $\Gamma; \Delta, n{:}[T]^{+} \vdash \vec{v} : \vec{T}$ abbreviates the following: assume $\vec{v} = v_1, \ldots v_n$ and $\vec{T} = T_1 \ldots T_n$ and let Ξ_1 abbreviate $\Gamma; \Delta, n{:}[T]^{+}$. Then $\Xi \vdash \vec{v} : \vec{T}$ means: $\Xi_i \vdash v_i : T_i$ and $\Xi_{i+1} = \Xi_i \backslash T_i$, for all $1 \leq i \leq n$. Note that checking the type of the callee has no side-effect on the bindings. Mentioning a variable or a name removes the write permission (if present) from the respective binding context (cf. T-VAR and T-NAME). The next three rules T-SUSPEND, T-GRAB, and T-RELEASE deal with the expressions for coordination and lock handling; they are typed by Unit. The last rule T-SUB is the standard rule of subsumption.

2.3 Operational semantics

The operational semantics is given in two stages, component internal steps and external ones, where the latter describe the interaction at the interface. Section 2.3.1 starts with component-internal steps, i.e., those definable without reference to the environment. In particular, the steps have no externally observable effect. The external steps, presented afterwards in Section 2.3.2, define the interaction between component and environment. They are defined in reference to assumption and commitment contexts. The static part of the contexts corresponds to the static type system from Section 2.2 on component level and takes care that, e.g., only well-typed values are received from the environment.

2.3.1 Internal steps

The internal semantics describes the operational behavior of a *closed* system, not interacting with its environment. The corresponding reduction steps are shown in Table 5, 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 imperative side effects and thus confluent. The $\xrightarrow{\tau}$-steps, in contrast, access the instance state, either by reading or by writing it, and may thus lead to race conditions. In other words, this part of the reduction relation is in general not confluent.

The first seven rules deal with the basic sequential constructs, all as $\rightsquigarrow$-steps. The basic evaluation mechanism is substitution (cf. rule RED). Note that the rule requires that the leading let-bound variable is replaced only by *values* v. The operational behavior of the two forms of conditionals are axiomatized by the four COND-rules. Depending on the result of the comparison in the first pair of rules, resp., the result of checking for definedness in the second pair, either the then- or the else-branch is taken. In COND_2, we assume that v_1 does not equal v_2, as side condition. Evaluating stop terminates the future for good, i.e., the rest of the thread will never be executed as there is no reduction rule for the future $n\langle \mathsf{stop}\rangle$ (cf. rule STOP). The rule FLOOKUP deals with field look-up, where $F'.l(o)()$ stands for $\bot_c[o/s] = \bot_c$, resp., for $v[o/s]$, where $[c, F', L] = [c, \ldots, l = \varsigma(s{:}c).\lambda().\bot_c, \ldots, L]$, if the field is yet undefined, resp., $[c, F', L] = [c, \ldots, l = \varsigma(s{:}c).\lambda().v, \ldots, L]$. In FUPDATE, the meta-mathematical notation $F.l := v$ stands for $(\ldots, l = v, \ldots)$, when $F = (\ldots, l = v', \ldots)$. There will be no external variant of the rule for field look-up later in the semantics of open systems, as we do not allow field access across component boundaries. The same restriction holds for field update in rule FUPDATE. A new object as instance of a given class is created by rule NEWO_i. Note that initially, the lock is free and there is no activity associated with the object, i.e., the object is initially passive.

The expression promise T creates a fresh promise n'. A new thread $n'\langle \bullet \rangle$ is allocated with an "undefined" body, as so far nothing more than the name is known. The rule PROM mentions the types T and T'. The typing system assures that the type T is of the form $[S]^{+-}$ for some type S. A promise is fulfilled by the bind-command (cf. rule BIND_i), in that the new thread n' is put together with the code to be executed and run in parallel with the rest. 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 $[M]$ equals $[\ldots, l = \varsigma(s{:}T).\lambda(\vec{x}{:}\vec{T}).t, \ldots]$.

Upon termination, the result is available via the claim- and the get-syntax (cf. the CLAIM-rules and rule GET_i), but not before the lock of the object is given back again using release(o) (cf. rule RELEASE). If the thread is not yet terminated, the requesting thread suspends itself, thereby giving up the lock. The behavior of claim is sketched in Figure 1. Note the types of the involved let-bound variables: the future reference is typed by $[T]$, indicating that the value for x will not directly be available, but must be dereferenced first via claim.

The two operations grab and release take, resp., give back an object's lock. They are not part of the user syntax, i.e., the programmer cannot directly manipulate the monitor lock. The user can release the lock using the suspend-command or by trying to get back the result from a call using claim.

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 6 where in the fourth axiom, n does not occur free in C_1. The congruence relation is imported into the reduction relations in Table 7. Note that all syntactic entities are always tacitly understood modulo α-conversion.

For illustration of the operational semantics, we show the combination of creating a promise and binding a method body to it. The steps in the reduction sequence below are justified by PROM, LET, and BIND, in that order. In the sequence, we did not write the definition of the object plus the class, needed to do the last reduction step. I.e., the reduction sequence below runs in parallel with $c[(F',M)] \parallel o[c,F,L]$, where in particular the method suite M, stored in the class c of the object o, contains the definition of the method body. That definition is needed for binding operation

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

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

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

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

$$n\langle \mathsf{let}\,x{:}T = (\mathsf{if}\,\mathsf{undef}(\perp_{c'})\,\mathsf{then}\,e_1\,\mathsf{else}\,e_2)\,\mathsf{in}\,t\rangle \rightsquigarrow n\langle \mathsf{let}\,x{:}T = e_1\,\mathsf{in}\,t\rangle \qquad \textsc{Cond}_1^{\perp}$$

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

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

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

$$o[c,F,L] \parallel n\langle \mathsf{let}\,x{:}T = o.l := v\,\mathsf{in}\,t\rangle \xrightarrow{\tau} o[c,F.l := v,L] \parallel n\langle \mathsf{let}\,x{:}T = o\,\mathsf{in}\,t\rangle \qquad \text{F}\textsc{Update}$$

$$\begin{aligned} &c[(F,M)] \parallel n\langle \mathsf{let}\,x{:}c = \mathsf{new}\,c\,\mathsf{in}\,t\rangle \rightsquigarrow \\ &\qquad c[(F,M)] \parallel \nu(o{:}c).(o[c,F,\perp] \parallel n\langle \mathsf{let}\,x{:}c = o\,\mathsf{in}\,t\rangle) \qquad \textsc{NewO}_i \end{aligned}$$

$$n\langle \mathsf{let}\,x{:}T' = \mathsf{promise}\,T\,\mathsf{in}\,t\rangle \rightsquigarrow \nu(n'{:}T').(n\langle \mathsf{let}\,x{:}T' = n'\,\mathsf{in}\,t\rangle \parallel n'\langle\bullet\rangle) \qquad \textsc{Prom}$$

$$\begin{aligned} &c[(F',M)] \parallel o[c,F,L] \parallel n_1\langle \mathsf{let}\,x{:}T = \mathsf{bind}\,o.l(\vec{v}) : T_2 \hookrightarrow n_2\,\mathsf{in}\,t_1\rangle \parallel n_2\langle\bullet\rangle \xrightarrow{\tau} \\ &\qquad c[(F',M)] \parallel o[c,F,L] \parallel n_1\langle \mathsf{let}\,x{:}T = n_2\,\mathsf{in}\,t_1\rangle \qquad \textsc{Bind}_i \\ &\qquad\qquad \parallel n_2\langle \mathsf{let}\,x{:}T_2 = \mathsf{grab}(o); M.l(o)(\vec{v})\,\mathsf{in}\,\mathsf{release}(o); x\rangle \end{aligned}$$

$$n_1\langle v\rangle \parallel n_2\langle \mathsf{let}\,x : T = \mathsf{claim}@(n_1,o)\,\mathsf{in}\,t\rangle \rightsquigarrow n_1\langle v\rangle \parallel n_2\langle \mathsf{let}\,x : T = v\,\mathsf{in}\,t\rangle \qquad \textsc{Claim}_i^1$$

$$\frac{t_2 \neq v}{\begin{aligned} &n_2\langle t_2\rangle \parallel n_1\langle \mathsf{let}\,x : T = \mathsf{claim}@(n_2,o)\,\mathsf{in}\,t_1'\rangle \rightsquigarrow \\ &\qquad n_2\langle t_2\rangle \parallel n_1\langle \mathsf{let}\,x : T = \mathsf{release}(o); \mathsf{get}@n_2\,\mathsf{in}\,\mathsf{grab}(o); t_1'\rangle \end{aligned}} \ \textsc{Claim}_i^2$$

$$n_1\langle v\rangle \parallel n_2\langle \mathsf{let}\,x : T = \mathsf{get}@n_1\,\mathsf{in}\,t\rangle \rightsquigarrow n_1\langle v\rangle \parallel n_2\langle \mathsf{let}\,x : T = v\,\mathsf{in}\,t\rangle \qquad \textsc{Get}_i$$

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

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

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

Table 5
Internal steps

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

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

Table 6
Structural congruence

in the last reduction step. In the rule corresponding rule $\text{B}\small{\text{IND}}_i$, this is written as $M.l(o)(\vec{v})$. In the final configuration, t' contains the result of looking up the method body and is of the form $\mathsf{grab}(o);M.l(o)(\vec{v})$. This, the overall behavior of the fulfilled promise n_2, i.e., after the binding step, is: first acquire the lock of the object, afterwards executed the method body with the formal parameters including the self-parameter appropriately substituted. With the return value computed and remembered in z, the lock is released and the result made available under the future reference n_2:

$$\begin{array}{ll}
n_1\langle \mathsf{let}\, x{:}[T]^{+-} = \mathsf{promise}\, T\, \mathsf{in}\ (\mathsf{let}\, y : T_2 = \mathsf{bind}\, o.l(\vec{v}) : T \hookrightarrow x\, \mathsf{in}\, t)\rangle & \rightsquigarrow \\
\nu(n_2{:}[T]^{+-}).(n_1\langle \mathsf{let}\, x{:}[T]^{+-} = n_2\, \mathsf{in}\ (\mathsf{let}\, y{:}[T]^{+} = \mathsf{bind}\, o.l(\vec{v}) : T \hookrightarrow x\, \mathsf{in}\, t)\rangle \parallel n_2\langle\bullet\rangle) & \rightsquigarrow \\
\nu(n_2{:}[T]^{+-}).(n_1\langle \mathsf{let}\, y{:}[T]^{+} = \mathsf{bind}\, o.l(\vec{v}) : T \hookrightarrow n_2\, \mathsf{in}\, t[n_2/x]\rangle\rangle \parallel n_2\langle\bullet\rangle) & \xrightarrow{\tau} \\
\nu(n_2{:}[T]^{+-}).(n_1\langle \mathsf{let}\, y{:}[T]^{+} = n_2\, \mathsf{in}\, t[n_2/x]\rangle\rangle \parallel n_2\langle \mathsf{let}\, z{:}T = t'\, \mathsf{in}\, \mathsf{release}(o); z\rangle) &
\end{array}$$

Note that the overall behavior of first creating a promise and, in a next step, binding a method body to it, corresponds exactly to the working of an asynchronous method call. Asynchronous method calls can therefore be seen as syntactic sugar. The introduction of promises as a separate datatype and binding as corresponding, separate operation on promises therefore generalizes the setting with futures and asynchronous method calls, only.

In the following, we show that the type system indeed assures what it is supposed to, most importantly that a promise is indeed fulfilled only once. An important part of it is a standard property, namely preservation of well-typedness under internal

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

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

Table 7
Reduction modulo congruence

reduction (subject reduction). First we characterize as erroneous situations where a promise is about to be written a second time: A configuration C contains a *write error* if it is of the form $C \equiv \nu(\Theta').(C' \parallel n'\langle \mathsf{let}\, x : T = \mathsf{bind}\, t_1 : T_1 \hookrightarrow n\, \mathsf{in}\, t_2\rangle \parallel n\langle t\rangle)$. Configurations without such write-errors are called *write-error free*, denoted $\vdash C : ok$. In [55], an analogous condition is called *handle error*.

The necessary ancillary lemmas will in general proceed by induction on the typing derivations for judgments of the form $\Delta \vdash C : \Theta$. From a proof-theoretical (and algorithmic) point of view, the type system as formalized in Tables 2, 3, and 4 has an unwelcome property: it is too "non-deterministic" in that it allows the non-structural subsumption rules T-SUB on the level of threads t and on the level of components C at any point in the derivation. This liberality is unwelcome for proofs by induction on the typing derivation as one loses knowledge about the structure of the premises of an applied rule in the derivation. We write $\Delta \vdash_m C : \Theta$ for derivations where subsumption at the level of components (by rule T-SUB from Table 2) is *not* used, and subsumption from Table 4 is only used "when needed", i.e., for adaptation. Taking for instance T-BIND and concentrating on the premises relevant for the illustration: Given as the interface type of the class $\Gamma;\Delta, n{:}[T]^+ \vdash_m c : [(\ldots, l{:}\vec{T} \to T, \ldots)]$ and furthermore $\Gamma;\Delta, n{:}[T]^+ \vdash_m \vec{v} : \vec{S}$, the minimal types $\vec{S}$ of the $\vec{v}$ may not directly match the expected argument type $\vec{T}$ of the method labeled l (as is required in the premise of the rule T-BIND). Restricting now the use of subsumption to "*adapt*" the $\vec{S}$ to $\vec{T}$ gives the type system for minimal types (denoted by using $\vdash_m$ instead of $\vdash$). This could be explicitly done my removing the freely applicable T-SUB and distributing its effect into the premises of structural rules, where such adaptation is needed. In the discussed rule T-BIND, by stipulating

$$\frac{\ldots \qquad \Gamma;\Delta, n{:}[T]^+ \vdash_m c : [(\ldots, l{:}\vec{T} \to T, \ldots)] \quad \Gamma;\Delta, n{:}[T]^+ \vdash_m \vec{v} : \vec{S} \quad \vec{S} \leq \vec{T}}{\Gamma;\Delta, n : [T]^{+-} \vdash_m \mathsf{bind}\, o.l(\vec{v}) : T \hookrightarrow n : [T]^+ :: \acute{\Gamma};\acute{\Delta}, n{:}[T]^+} \text{ T-BIND}_m$$

where $\vec{S} \leq \vec{T}$ is interpreted pointwise $S_i \leq T_i$, for all i. As the formulation of that type system is rather standard and straightforward, we omit its definition.

Lemma 2.3 (Minimal typing) *(1) If $\Delta \vdash_m C : \Theta$ and $\Delta' \vdash C : \Theta'$, then $\Delta \leq \Delta'$ and $\Theta' \leq \Theta$.*
(2) If $\Delta \vdash_m C : \Theta$ then $\Delta \vdash C : \Theta$.
(3) If $\Delta' \vdash C : \Theta'$, then $\Delta \vdash_m C : \Theta$ with $\Delta \leq \Delta'$ and $\Theta' \leq \Theta$.

Proof: Straightforward. □

First we show that a well-typed component does not contain a manifest write-error.

Lemma 2.4 *If $\Delta \vdash_m C : \Theta$, then $\vdash C : ok$.*

Proof: By induction on the typing derivations for judgments on the level of components, i.e., for judgments of the form $\Delta \vdash C : \Theta$; the subordinate typing rules from Tables 3 and 4 on the level of threads and expressions do not play a role for the proof. The empty component in the base case of T-EMPTY is clearly write-error free. The cases for the T-NU-rules by straightforward induction. The cases for T-NCLASS, T-NOBJ, and T-NFUTURE are trivially satisfied, as they mention a single, basic component, only.

Case: T-PAR
We are given $\Delta_1, \Theta_2 \vdash C_1 : \Theta_1$ and $\Delta_2, \Theta_1 \vdash C_2 : \Theta_2$ with $\Delta = \Delta_1 \oplus \Delta_2$. By induction, both C_1 and C_2 are write-error free. The non-trivial case (which we will lead to a contradiction) is when one of the components attempts to write to a promise and the partner already has fulfilled it. So, without loss of generality assume that $C_1 = \nu(\Theta_1').(C_1' \parallel n_1\langle \mathsf{let}\, x : T = \mathsf{bind}\, x : T \hookrightarrow n_2 \,\mathsf{in}\, t''\rangle$ and $C_2 = \nu(\Theta_2').(C_2' \parallel n_2\langle t_2\rangle)$. Assume that n_2 occurs in neither Θ_1' nor Θ_2', otherwise no write error is present (since in that case, the name n_2 mentioned on both sides of the parallel refer to different entities). For C_1 to be well-typed, we have $\Delta_1, \Theta_2 \vdash n_2 : [T_2]^{+-}$ for some type T_2. For C_2 to be well-typed, we have $\Theta_2 \vdash n : [T_2]^{+}$ for some type T_2. Thus, $\Delta \vdash C_1 \parallel C_2 : \Theta_1, \Theta_2$ cannot be derived, which contradicts the assumption. □

Lemma 2.5 (Subject reduction: $\equiv$) *If $\Delta \vdash_m C_1 : \Theta$ and $C_1 \equiv C_2$, then $\Delta \vdash_m C_2 : \Theta$.*

Proof: We show preservation of typing by the axioms of Table 6. Proceed by induction on the derivation of $\Delta \vdash_m C_1 : \Theta$.

Case: $C \parallel \mathbf{0} \equiv C$ (idempotence)
We are given $\Delta \vdash C \parallel \mathbf{0} : \Theta$. Inverting T-PAR and by T-EMPTY we get as sub-goals $\Delta, \Theta \vdash_m \mathbf{0} : ()$ and $\Delta \vdash_m C : \Theta$, which concludes the case.

Case: $C \equiv C \parallel \mathbf{0}$ (idempotence)
Immediate using T-PAR and T-EMPTY.

Case: $C_1 \parallel C_2 \equiv C_2 \parallel C_1$ (commutativity)
Immediate.

Case: $C_1 \parallel (C_2 \parallel C_3) \equiv (C_1 \parallel C_2) \parallel C_2$ and vice versa (associativity)
By straightforward induction.

Case: $C_1 \parallel \nu(n{:}T).C_2 \equiv \nu(n{:}T).(C_1 \parallel C_2)$
where n does not occur free in C_1. We are given $\Delta \vdash C_1 \parallel \nu(n{:}T).C_2 : \Theta_1, \Theta_2$, where n occurs in neither Θ_1 nor Θ_2. Inverting T-PAR and T-NU, we obtain as two sub-goals $\Delta, \Theta_2 \vdash C_1 : \Theta_1$ and $\Delta, \Theta_1 \vdash C_2 : \Theta_1, \Theta_2, n{:}T$, and the result follows by T-PAR and the T-NU-rule.

Case: $\nu(n_1{:}T_1).\nu(n_2{:}T_2).C \equiv \nu(n_2{:}T_2).\nu(n_1{:}T_1).C$
Analogously. □

The next lemma is another step towards subject reduction. Note that minimal types are *not* preserved by reduction. Especially executing a bind-operation with rule $\textsc{Bind}_i$ changes the type of the corresponding name from $[T]^{+-}$ to $[T]^{+}$.

Lemma 2.6 (Subject reduction: $\xrightarrow{\tau}$ and $\rightsquigarrow$) *Assume $\Delta \vdash_m C : \Theta$.*

(1) If $C \xrightarrow{\tau} \acute{C}$, then $\Delta \vdash \acute{C} : \Theta$.
(2) If $C \rightsquigarrow \acute{C}$, then $\Delta \vdash \acute{C} : \Theta$.

Proof: The reduction rules of Table 5 are all of the form $C_1 \parallel n\langle t_1\rangle \xrightarrow{\tau} C_2 \parallel n\langle t_2\rangle$, where often $C_1 = C_2$ or C_1 and C_2 missing. In the latter case, it suffices to show that $;\Delta, n{:}[T]^{+} \vdash_m t_1 : T$ implies $;\Delta, n{:}[T]^{+} \vdash t_2 : T$.

Case: $\textsc{Red}$: $n\langle \mathsf{let}\, x : T = v\, \mathsf{in}\, t\rangle \rightsquigarrow n\langle t[v/x]\rangle$
By preservation of typing under substitution.

The 5 rules for let and for conditionals are straightforward. The case for stop follows from the fact that stop has every type (cf. rule T-$\textsc{Stop}$).

Case: $\textsc{Prom}$: $n\langle \mathsf{let}\, x{:}T' = \mathsf{promise}\, T\, \mathsf{in}\, t\rangle \rightsquigarrow \nu(n'{:}T').(n\langle \mathsf{let}\, x : T' = n'\, \mathsf{in}\, t\rangle \parallel n'\langle\bullet\rangle)$
The type system (for minimal types) assures that $T' = [T]^{+-}$, i.e., for the left-hand side of the reduction step, we obtain as one subgoal (inverting T-N$\textsc{Thread}'$, T-$\textsc{Let}$, and T-$\textsc{Prom}$) $x{:}[T]^{+-};\Delta, n{:}[S]^{+} \vdash t : S$. The result follows from T-$\textsc{Nu}$, T-$\textsc{Par}$, T-$\textsc{Let}$, and T-N$\textsc{Thread}'$ (and weakening):

$$\dfrac{\dfrac{\dfrac{\dfrac{\ldots \qquad x{:}[T]^{+-};\Delta, n'{:}[T]^{+-}, n{:}[S]^{+} \vdash t : S}{;\Delta, n'{:}[T]^{+-}, n{:}[S]^{+} \vdash \mathsf{let}\, x : [T]^{+-} = n'\, \mathsf{in}\, t : S}}{\Delta, n'{:}[T]^{+-} \vdash n\langle \mathsf{let}\, x : [T]^{+-} = n'\, \mathsf{in}\, t\rangle : n{:}[S]^{+}} \quad \dfrac{}{\Delta, n{:}[S]^{+}, n'{:}[T]^{+} \vdash n'\langle\bullet\rangle : n'{:}[T]^{+-}}}{\Delta \vdash n\langle \mathsf{let}\, x : [T]^{+-} = n'\, \mathsf{in}\, t\rangle \parallel n'\langle\bullet\rangle : (n{:}[S]^{+}, n'{:}[T]^{+-})}\ \textsc{T-Par}}{\Delta \vdash \nu(n'{:}[T]^{+-}).(n\langle \mathsf{let}\, x : T' = n'\, \mathsf{in}\, t\rangle \parallel n'\langle\bullet\rangle) : (n{:}[S]^{+})}\ \textsc{T-Nu}$$

Case: $\textsc{Bind}_i$ $n_1\langle t\rangle \parallel n_2\langle\bullet\rangle = n_1\langle \mathsf{let}\, x{:}T = \mathsf{bind}\, o.l(\vec{v}) : T_2 \hookrightarrow n_2\, \mathsf{in}\, t_1\rangle \parallel n_2\langle\bullet\rangle \xrightarrow{\tau}$ $n_1\langle \mathsf{let}\, x{:}T = n_2\, \mathsf{in}\, t_1\rangle \parallel n_2\langle \mathsf{let}\, x{:}T_2 = \mathsf{grab}(o); M.l(o)(\vec{v})\, \mathsf{in}\, \mathsf{release}(o); x\rangle$
The type system assures (cf. T-$\textsc{Bind}$) that $T = [T_2]^{+}$. By assumption, we are given $\Delta \vdash n_1\langle t\rangle : \Theta$, which implies $\Theta = n_1{:}[T_1]^{+}$ for some type T_1. Inverting rules T-$\textsc{Par}$, T-N$\textsc{Thread}$, T-$\textsc{Let}$, and T-$\textsc{Bind}$ gives for the named thread n_1:

$$\dfrac{\dfrac{\dfrac{;\Delta_1, n_2{:}[T_2]^{+}, n_1{:}[T_1]^{+} \vdash \vec{v} : \vec{T} \quad \Delta'' = \Delta' \backslash (\vec{v}{:}\vec{T}) \quad \ldots}{;\Delta_1, n_2{:}[T_2]^{+-} \vdash \mathsf{bind}\, o.l(\vec{v}) : T_2 \hookrightarrow n_2 : T :: \ ;\Delta''_1, n_2{:}[T_2]^{+}}\ \textsc{T-Bind} \qquad x{:}[T_2]^{+};\Delta''_1, n_2{:}[T_2]^{+} \vdash t_1 : T_1 :: x{:}[T_2]^{+};\acute{\Delta}_2, n_2{:}[T_2]^{+}}{;\Delta_1, n_2{:}[T_2]^{+-}, n_1{:}[T_1]^{+} \vdash \mathsf{let}\, x{:}[T_2]^{+} = \mathsf{bind}\, o.l(\vec{v}) : T_2 \hookrightarrow n_2\, \mathsf{in}\, t_1 : T_1 :: \ ;\acute{\Delta}_1, n_2{:}[T_2]^{+}, n_1{:}[T_1]^{+}}}{\Delta_1, n_2{:}[T_2]^{+-} \vdash n_1\langle \mathsf{let}\, x{:}[T_2]^{+} = \mathsf{bind}\, o.l(\vec{v}) : T_2 \hookrightarrow n_2\, \mathsf{in}\, t_1\rangle : n_1{:}[T_1]^{+}}$$

Rule T-BIND (and T-NTHREAD) implies that the assumption context Δ contains especially the binding $n_2{:}[T]^{+-}$, i.e., the assumption Δ in the last conclusion is of the form $\Delta', n_2{:}[T_2]^{+-}$.

Now to the post-configuration after the $\xrightarrow{\tau}$-step. With T-PAR we obtain the following two sub-goals:

$$\dfrac{\Delta, n_2{:}[T_2]^{+-} \vdash n_1\langle \mathsf{let}\, x{:}T = n_2\, \mathsf{in}\, t_1\rangle : n_1{:}[T_1]^{+} \qquad \Delta, n_1{:}[T_1]^{+} \vdash n_2\langle \mathsf{let}\, x{:}T_2 = \mathsf{grab}(o); M.l(o)(\vec{v})\, \mathsf{in}\, \mathsf{release}(o); x\rangle : n_2{:}[T_2]^{+-}}{\Delta \vdash n_1\langle \mathsf{let}\, x{:}T = n_2\, \mathsf{in}\, t_1\rangle \parallel n_2\langle \mathsf{let}\, x{:}T_2 = \mathsf{grab}(o); M.l(o)(\vec{v})\, \mathsf{in}\, \mathsf{release}(o); x\rangle : n_1{:}[T_1]^{+}, n_2{:}[T_2]^{+-}}$$

The left one can be derived using T-NTHREAD, T-LET, and T-NAME, where there second premise of T-LET is discharged by the corresponding assumption from above and weakening.

$$\dfrac{\dfrac{\dfrac{}{\Delta, n_2{:}[T_2]^{+}, n_1{:}[T_2]^{+} \vdash n_2 : [T_2]^{+}}\,\textsc{T-Name} \qquad x{:}[T_2]^{+}, \Delta; n_2{:}[T_2]^{+}, n_1{:}[T_2]^{+} \vdash t_1 : T_1}{\Delta, n_2{:}[T_2]^{+}, n_1{:}[T_2]^{+} \vdash \mathsf{let}\, x{:}T = n_2\, \mathsf{in}\, t_1 : T_1}\,\textsc{T-Let}}{\Delta, n_2{:}[T_2]^{+} \vdash n_1\langle \mathsf{let}\, x{:}[T_2]^{+} = n_2\, \mathsf{in}\, t_1\rangle : n_1{:}[T_1]^{+}}$$

The second premise can be derived as follows:

$$\dfrac{\dfrac{\dfrac{\dots \qquad \dfrac{;\Delta, n_1{:}[T_1]^{+}, n_2{:}[T_2]^{+} \vdash M.l(o)(\vec{v}) : T_2 \qquad \dfrac{\dfrac{}{y{:}T_2; \Delta, n_1{:}[T_1]^{+}, n_2{:}[T_2]^{+} \vdash y : T_2}\,\textsc{T-Var}}{y{:}T_2; \Delta, n_1{:}[T_1]^{+}, n_2{:}[T_2]^{+} \vdash \mathsf{release}(o); y : T_2}}{;\Delta, n_1{:}[T_1]^{+}, n_2{:}[T_2]^{+} \vdash \mathsf{let}\, y{:}T_2 = M.l(o)(\vec{v})\, \mathsf{in}\, \mathsf{release}(o); y : T_2}\,\textsc{T-Let}}{;\Delta, n_1{:}[T_1]^{+}, n_2{:}[T_2]^{+} \vdash \mathsf{let}\, y{:}T_2 = \mathsf{grab}(o); M.l(o)(\vec{v})\, \mathsf{in}\, \mathsf{release}(o); y : T_2}}{\Delta, n_1{:}[T_1]^{+} \vdash n_2\langle \mathsf{let}\, y{:}T_2 = \mathsf{grab}(o); M.l(o)(\vec{v})\, \mathsf{in}\, \mathsf{release}(o); y\rangle : n_2{:}[T_2]^{+}}\,\textsc{T-NThread}}{\Delta, n_1{:}[T_1]^{+} \vdash n_2\langle \mathsf{let}\, y{:}T_2 = \mathsf{grab}(o); M.l(o)(\vec{v})\, \mathsf{in}\, \mathsf{release}(o); y\rangle : n_2{:}[T_2]^{+-}}\,\textsc{T-Sub}$$

The premise $;\Delta, n_1{:}[T_1]^{+}, n_2{:}[T_2]^{+} \vdash M.l(o)(\vec{v}) : T_2$ follows by preservation of typing by substitution. Note the use of subsumption in the last step. $\square$

Lemma 2.7 (Subject reduction: $\equiv$) *If $\Delta \vdash C_1 : \Theta$ and $C_1 \equiv C_2$, then $\Delta \vdash C_2 : \Theta$.*

Proof: Assume $\Delta \vdash C_1 : \Theta$ and $C_1 \equiv C_2$. By Lemma 2.3(3), $\Delta' \vdash_m C_1 : \Theta'$ s.t. $\Delta \leq \Delta'$ and $\Theta' \leq \Theta$. By Lemma 2.5, $\Delta' \vdash_m C_2 : \Theta'$, and hence by Lemma 2.3(2), also $\Delta' \vdash C_2 : \Theta'$, and the result follows by subsumption (rule T-SUB). $\square$

Lemma 2.8 (Subject reduction: $\xrightarrow{\tau}$ and $\rightsquigarrow$) *Assume $\Delta \vdash C : \Theta$.*

(1) If $C \xrightarrow{\tau} \acute{C}$, then $\Delta \vdash \acute{C} : \Theta$.
(2) If $C \rightsquigarrow \acute{C}$, then $\Delta \vdash \acute{C} : \Theta$.

Proof: As consequence of the corresponding property for minimal typing from Lemma 2.6, Lemma 2.3, and subsumption. □

Lemma 2.9 (Subject reduction) *If* $\Delta \vdash C : \Theta$ *and* $C \Longrightarrow \acute{C}$*, then* $\Delta \vdash \acute{C} : \Theta$.

Proof: A consequence of Lemma 2.7 and 2.8. □

A direct consequence is that all reachable configurations are write-error free:

Corollary 2.10 *If* $\Delta \vdash C : \Theta$ *and* $C \Longrightarrow \acute{C}$*, then* $\vdash \acute{C} : ok$.

Proof: A consequence of Lemma 2.4 and subject reduction from Lemma 2.9. □

2.3.2 External semantics

In this section we introduce the external semantics that defines the interaction between component and environment. We start by formalizing typing judgments and transitions between typing judgments, being the basic form of the external steps. We continue with static typing assumptions for well-formed and well-typed labels. Context updates, given next, express the dynamic change of typing judgments for incoming and outgoing communications. Making use of the above formalisms, we give the steps of the external semantics.

The external semantics formalizes the interaction of an open component with its environment. The semantics is given as labeled transitions between typing judgments on the level of components (cf. Table 2), i.e., judgments of the form

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

where, as before, Δ represents the assumptions about the environment of the component C and Θ the commitments. The assumptions require the existence of named entities in the environment (plus giving static typing information), and dually, the commitment promises the existence of such entities in C. It is an invariant of the semantics, that the assumption and commitment contexts are disjoint concerning their name bindings. In addition, the interface keeps information about whether the value of a future n is already known at the interface (this is a bit of information not needed in the static type system of Table 2). If it is, we write $n{:}T = v$ as binding of the context. We write furthermore $\Delta \vdash n = v$, if Δ contains the corresponding value information (and if not interested in the type) and write $\Delta \vdash n = \bot$, if that is not the case. This extension makes the value of a future (once successfully claimed) available at the interface. With these judgments, the external transitions are of the form:

$$\Delta \vdash C : \Theta \quad \xrightarrow{a} \quad \acute{\Delta} \vdash \acute{C} : \acute{\Theta}\ . \tag{4}$$

Notation 2.11 *We abbreviate the tuple of name contexts* Δ, Θ *as* Ξ*. Furthermore we understand* $\acute{\Delta}, \acute{\Theta}$ *as* $\acute{\Xi}$*, etc.*

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

Table 8
Labels

The labels of the external transitions represent single steps of the interface interactions (cf. Table 8). A component exchanges information with the environment via *call* and *get* labels (by convention, referred to as γ_c and γ_g, for short). Interaction is either incoming or outgoing, indicated by ?, resp., !. In the labels, n is the identifier of the thread (i.e., also future/promise) carrying out the call resp. of being queried via claim or get. Besides that, object and future names (but no class names) may appear as arguments in the communication. Scope extrusion of names across the interface is indicated by the ν-binder. Given a basic label $\gamma = \nu(\Xi).\gamma'$ where Ξ is a name context such that $\nu(\Xi)$ abbreviates a sequence of single $n{:}T$ bindings (whose names are assumed all disjoint, as usual) and where γ' does not contain any binders, we call γ' the *core* of the label and refer to it by $\lfloor\gamma\rfloor$. We define the core analogously for receive and send labels. The free names $\mathit{fn}(a)$ and the bound names $\mathit{bn}(a)$ of a label a are defined as usual, whereas $\mathit{names}(a)$ refer to all names of a. In addition, we distinguish between names occurring as arguments of a label, in *passive* position, and the name occurring as carrier of the activity, in *active* position. Name n, for illustration, occurs actively and free in $n\langle \mathit{call}\ o.l.(\vec{v})\rangle$ and in $n\langle \mathit{get}(v)\rangle$. We write $\mathit{fn}_a(a)$ for the free names occurring in active position, $\mathit{fn}_p(a)$ for the free names in passive position, etc. All notations are used analogously for basic labels γ. Note that for incoming labels, Ξ contains only bindings to environment objects (besides future names), as the environment cannot create component objects; dually for outgoing communication.

The steps of the operational semantics for open systems check the *static* assumptions, i.e., whether at most the names actually occurring in the core of the label are mentioned in the ν-binders of the label, and whether the transmitted values are of the correct types. This is covered in the following definition.

Definition 2.12 (Well-formedness and well-typedness) *A label $a = \nu(\Xi).\lfloor a\rfloor$ is* well-formed, *written $\vdash a$, if $\mathit{dom}(\Xi) \subseteq \mathit{names}(\lfloor a\rfloor)$ and if Ξ is a well-formed name-context for object and future names, i.e., no name bound in Ξ occurs twice. The assertion*

$$\acute{\Xi} \vdash o.l? : \vec{T} \to T \tag{5}$$

("an incoming call of the method labeled l in object o expects arguments of type $\vec{T}$ and results in a value of type T") is given by the following rule, i.e., implication:

$$\frac{;\acute{\Theta} \vdash o : c \qquad ;\acute{\Xi} \vdash c : [(\ldots, l{:}\vec{T} \to T, \ldots)]}{\acute{\Xi} \vdash o.l? : \vec{T} \to T} \tag{6}$$

For outgoing calls, $\acute{\Xi} \vdash o.l! : \vec{T} \to T$ is defined dually. In particular, in the first

$$\frac{\acute{\Xi} = \acute{\Xi}_1, n{:}[T]^+, \acute{\Xi}_2 \quad ;\acute{\Xi} \vdash \vec{v} : \vec{T} \quad a = n\langle call\ o_r.l(\vec{v})\rangle?}{\acute{\Xi} \vdash a : \vec{T} \to \_}\ \text{LT-CALLI} \qquad \frac{;\acute{\Xi} \vdash v : T \qquad a = n\langle get(v)\rangle?}{\acute{\Xi} \vdash a : \_ \to T}\ \text{LT-GETI}$$

Table 9
Typechecking labels

premise, $\acute{\Theta}$ *is replaced by* $\acute{\Delta}$. Well-typedness *of an incoming core label a with expected type* $\vec{T}$*, resp.,* T*, and relative to the name context* $\acute{\Xi}$ *is asserted by*

$$\acute{\Xi} \vdash a : \vec{T} \to \_ \quad \textit{resp.,} \quad \acute{\Xi} \vdash a : \_ \to T \ , \tag{7}$$

as given by Table 9. Finally, let $\acute{\Xi}_0$ *abbreviate* $;\acute{\Xi}$*. Then* $;\acute{\Xi} \vdash \vec{v} : \vec{T}$ *means:* $\acute{\Xi}_i \vdash v_i : T_i$ *and* $\acute{\Xi}_{i+1} = \acute{\Xi}_i \setminus T_i$*, for all* $0 \leq i \leq n-1$.

Note that the receiver o of the call is checked using only the commitment context $\acute{\Theta}$, to assure that o is a component object. Note further that to check the interface type of the class c, the full $\acute{\Xi}$ is consulted, since the argument types $\vec{T}$ or the result type T may refer to both component and environment classes.

The premise $;\acute{\Xi} \vdash \vec{v} : \vec{T}$ in LT-CALLI is interpreted in such a way that checking for write-permission *consumes* that permission (analogous to the corresponding premise of T-BIND in Table 4). This is formalized in the definition of $;\Xi \vdash \vec{v} : \vec{T}$ for well-typedness of a sequence of values, given at the end of Definition 2.12, which iterates through the sequence, potentially removing write-permission for a v_i s.t. the permission is no longer available for type cheking the rest of the sequence.

In a similar spirit: requiring that $\acute{\Xi}$ is of the form $\acute{\Xi}_1, n{:}[T]^+, \acute{\Xi}_2$ assures that it is not possible to transmit n with write-permissions if n is the active thread of the label.

Besides *checking* whether the assumptions are met before a transition, the contexts are *updated* by a transition step, especially extended by the new names, whose scope extrudes. For the binding part Ξ' of a label $\nu(\Xi').\gamma$, the scope of the references to existing objects and thread names Δ' extrudes across the border. In the step, Δ' extends the assumption context Δ and Θ' the commitment context Θ. Besides information about new names, the context information is potentially updated wrt. the availability of a future *value.* This is done when a get-label is exchanged at the interface for the first time, i.e., when a future value is claimed successfully for the first time. For outgoing communication, the situation is dual.

Before we come to the corresponding Definition 2.13 below, we make clear (again) the interpretation of judgments $\Delta \vdash C : \Theta$. Interesting is in particular the information $n{:}[T]^{+-}$, stipulating that name n is available with write-permission (and result type T). In case of $\Delta \vdash n : [T]^{+-}$, the name n is assumed to be available in the environment as writable, and conversely $\Theta \vdash n : [T]^{+-}$ asserts write permission for the component. Since read permissions, captured by types $[T]^+$, are not treated linearly —one is allowed to read from a future reference as many times as wished— the treatment of bindings $n{:}[T]^+$ is simpler. Hence, we concentrate here on $n{:}[T]^{+-}$ and the write permissions.

As the domains of Δ and Θ are disjoint, bindings $n{:}T'$ cannot be available in the assumption context Δ and the commitments Θ at the same time. The information $T' = [T]^{+-}$ indicates which side, component or environment, has the write permission. If, e.g., $\Delta \vdash n : [T]^{+-}$, then the component is not allowed to execute a bind on reference n. In the mentioned situation, the component can execute a claim-operation on n. The same applies if $\Delta \vdash n : [T]^+$. In other words, a name n can be accessed by reading by both the environment and the component once known at the interface, independent from whether it is part of Δ or of Θ. A difference between bindings of the form $n{:}[T]^{+-}$ and $n{:}[T]^+$ (and likewise $n{:}[T]^+ = v$) is, that communication can *change* $\Delta \vdash n : [T]^{+-}$ to $\Theta \vdash n : [T]^{+-}$ and vice versa. For names n of type $[T]^+$, this change of side is impossible. The latter kind of information, for instance $\Theta \vdash n : [T]^+$, implies that the code has been bound to n and it is placed in the component. Once fixed there, the reference to n may, of course, be passed around, but the thread named n itself cannot change to the environment since the language does not support *mobile* code.

Now, how do interface interactions update the contexts? We distinguish two ways, the name n can be transmitted in a label: *passively,* when transported as the argument of a call or a get-interaction, and *actively,* when mentioned as the carrier of the activity, as the n in $n\langle call\ o.l(\vec{v})\rangle$ and $n\langle get(v)\rangle$. As usual, such references (actively or passively) can be transmitted as fresh names, i.e., under a ν-binder, or alternatively as an already known name. When transmitted *passively* and typed with $[T]^{+-}$ for some type T, the write-permission to n is handed over to the receiving side and at the same time, that permission is removed from the sender side.

Now, what about transmitting *n actively?* An incoming call $n\langle call\ o.l(\vec{v})\rangle?$, e.g., reveals at the interface that the promise indeed has been fulfilled. As, in that situation of an incoming call, the thread, executing the call, is located at the component, the commitment context is updated to satisfy $\Theta \vdash n : [T]^+ = \bot$ (for an appropriate type T) after the communication. Indeed, before the step it is checked, that the environment actually has write permission for n, i.e., that $\Delta \vdash n : [T]^{+-}$, or that the name n is new. See the incoming call in Figure 2(a), where the n is fresh, resp. in 2(c), where the n has been transmitted passively and with write-permissions to the environment before the call (in the dotted arrow).

Whereas call-labels make public, at which side the thread in question resides, get-labels, on the other hand, reveal that the computation has terminated and fix the result value (if the information about the result value had not been public interface information before). There are two situations, where a, say, outgoing get-communication is possible. In both cases, the named thread, representing the future, resides in the component and after the get-communication, the value is determined, i.e., $\Theta \vdash n : [T]^{+} = v$ (if not already before the step). One scenario is that $\Delta \vdash n : [T]^{+} = \bot$ before the step still. If, in that situation, the get is executed by the environment, it is required that the component must have had write permission before, i.e., $\Theta \vdash n : [T]^{+-}$ (cf. Figure 2(b)). The only way, the value for n is available for the environment now is that the promise had been fulfilled and the corresponding thread already has terminated, and this could have been done by the component, only. In that situation, the contexts are updated from $\Theta \vdash n : [T]^{+-}$ to $\Theta \vdash n : [T]^{+} = v$ by the get-interaction. Alternatively, the thread may be known to be part of the component with the promise already fulfilled ($\Theta \vdash n : [T]^{+} = \bot$, as shown in Figure 2(a) and 2(c)). Finally, the value for n might already been known at the interface, i.e., already before the step, $\Theta \vdash n : [T]^{+} = v$ holds. In that situation, v has been added as interface information previously by a prior get-interaction, and the situation corresponds to the very last get in Figure 2(b) and 2(c).

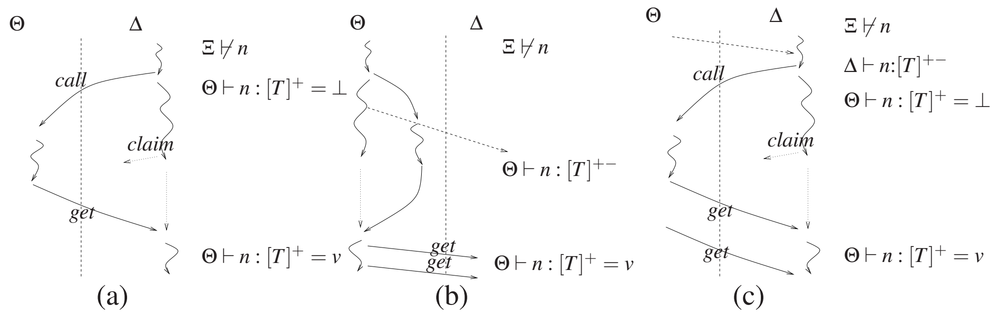

Fig. 2. Scenarios

Definition 2.13 (Context update) *Let Ξ be a name context and $a = \nu(\Xi').\lfloor a \rfloor$ an* incoming *label. Let $\acute{\Xi} = \Xi + a$ be defined as follows.*

We define the (intermediate) contexts $\Theta'' = \Theta$ and $\Delta'' = \Delta, \Xi'$. Let furthermore Σ'' be the set of bindings defined as follows. In case of a call label, i.e., $\lfloor a \rfloor = n\langle call\ o.l(\vec{v})\rangle?$, let the vector of types $\vec{T}$ be defined by $\Xi \vdash o.l? : \vec{T} \to T$ according to Equation (5) of Definition 2.12. Then Σ'' consists of bindings of the form $v_i{:}[T_i']^{+-}$ for values v_i from $\vec{v}$ such that $T_i = [T_i']^{+-}$. In case of a get label, i.e., $\lfloor a \rfloor = n\langle get(v)\rangle?$, the context Σ'' is $v{:}[T]^{+-}$ if $\Delta'' \vdash n : [[T]^{+-}]^{+}$, and empty otherwise.

With Σ'' given this way, the definitions of the post-contexts $\acute{\Delta}$ and $\acute{\Theta}$ distinguish between calls and get-interaction: If a is a call label and $n \in names_a(a)$, we define

$$\acute{\Delta} = (\Delta'' \setminus \Sigma'') \setminus n{:}[T]^{+-} \quad and \quad \acute{\Theta} = \Theta'', \Sigma'', n{:}[T]^{+} \,. \tag{8}$$

If a is a get label $a = \nu(\Xi').n\langle get(v)\rangle?$ and $n \in names_a(a)$, $\acute{\Delta}$ and $\acute{\Theta}$ are given by:

$$\acute{\Delta} = (\Delta'' \setminus \Sigma''), n{:}[T]^{+} = v \quad and \quad \acute{\Theta} = \Theta'', \Sigma'' \,. \tag{9}$$

For outgoing *communication, the definition is applied dually.*

The definition proceeds in two stages. In a first step, the assumption context Δ is extended with the bindings Ξ' carried with the incoming label a. The second step deals with the write permissions, i.e., it transfers the write permission transmitted from the sender to the receiver. The binding context Σ'' deals with the permissions carried by thread names transmitted passively, i.e., as arguments of the communication. It remains to take care also of the information carried by the active thread. There, we distinguish calls and get-labels. An incoming call (Equation (8)) with n as active thread is the sign that the thread is now located at the component side and that the write permission has been consumed by the environment. Hence, in Equation (8), the environment loses the write-permission and the component is extended by the binding $n{:}[T]^{+}$. In case of an incoming get, the transmitted value v is remembered as part of Δ (cf. Equation (8)).

The previous definition deals with the change of context information by communication. Apart from that, unfulfilled promises of the form $n\langle\bullet\rangle$ also change side, if their name is exchanged together with write-permission. As notation, we will use $C(\Xi)$ to denote the component $n_1\langle\bullet\rangle \parallel \ldots \parallel n_k\langle\bullet\rangle$, where the names n_i correspond to all names of the context Ξ mentioned as $n_i : [T_i]^{+-}$ for some type T_i.

Now to the interface behavior, given by the external steps of Table 10. Most rules have some premises in common. In all cases of a labeled transition, the context Ξ is updated to $\acute{\Xi} = \Xi + a$ using Definition 2.13. The rules for incoming communication differ from the corresponding ones for outgoing communication in that well-typedness and well-formedness of the label is checked by the premises $\acute{\Xi} \vdash \lfloor a \rfloor : \vec{T} \to \_$, resp. $\acute{\Xi} \vdash \lfloor a \rfloor : \_ \to \vec{T}$ (for calls) resp., $\acute{\Xi} \vdash \lfloor a \rfloor : \_ \to T$ (for get-labels), using Definition 2.12. For outgoing communication, the check is unnecessary as starting with a well-typed component, there is no need in re-checking now, as the operational steps preserve well-typedness (subject reduction).

When the component claims the value of a future, we distinguish two situations: the future value is accessed for the first time across the interface or not. In the first case (rules CLAIMI$_1$ and CLAIMI$_2$), the interface does not contain the value of the future yet, stipulated by the premise $\Delta \vdash n' = \bot$. In that situation it is unclear from the perspective of the component, whether or not the value has already been computed. Hence, it is possible that executing claim is immediately successful (cf. rule CLAIM$_1$) or that the thread n trying to obtain the value has to suspend itself and try later (cf. rule CLAIM$_2$). Rule CLAIM$_2$ works exactly like the corresponding internal rule CLAIM$_i^2$ from Table 5, except that here it is required that the queried future n' is part of the environment. If the future value is already known at the interface (cf. rule CLAIM$_3$ and especially premise $\Delta \vdash n' = v$), executing claim is always successful and the value v is (re-)transmitted. get works analogously to claim, except that get insists on obtaining the value, i.e., the alternative of relinquishing the lock and trying again as in rule CLAIM$_2$, is not available for get. The last two rules deal with the situation that the environment fetches the value.

$$\frac{a = \nu(\Xi').\, n\langle \mathit{call}\ o.l(\vec{v})\rangle? \quad \acute{\Xi} = \Xi + a \quad (\Xi' \vdash n \vee \Delta \vdash n : [\_]^{+-}) \quad \acute{\Xi} \vdash o.l? : \vec{T} \to T \quad \acute{\Xi} \vdash \lfloor a \rfloor : \vec{T} \to \_}{\Xi \vdash C \xrightarrow{a} \acute{\Xi} \vdash C \parallel C(\Xi') \parallel n\langle \mathsf{let}\, x{:}T = \mathsf{grab}(o); M.l(o)(\vec{v})\,\mathsf{in}\,\mathsf{release}(o); x\rangle} \ \textsc{CallI}$$

$$\frac{a = \nu(\Xi').\, n\langle \mathit{call}\ o.l(\vec{v})\rangle! \quad \Xi' = \mathit{fn}(\lfloor a \rfloor) \cap \Xi_1 \quad \acute{\Xi}_1 = \Xi_1 \setminus \Xi' \quad \Delta \vdash o \quad \acute{\Xi} = \Xi + a}{\Xi \vdash \nu(\Xi_1).(C \parallel C(\Xi') \parallel n\langle \bullet \rangle \parallel n'\langle \mathsf{let}\, x{:}T = \mathsf{bind}\, o.l(\vec{v}) : T \hookrightarrow n\ \mathsf{in}\, t\rangle) \xrightarrow{a} \acute{\Xi} \vdash \nu(\acute{\Xi}_1).(C \parallel n'\langle \mathsf{let}\, x : T = n\ \mathsf{in}\, t\rangle)} \ \textsc{CallO}$$

$$\frac{a = \nu(\Xi').\, n'\langle \mathit{get}(v)\rangle? \qquad \acute{\Xi} = \Xi + a \qquad \Delta \vdash n' = \bot \qquad \acute{\Xi} \vdash \lfloor a \rfloor : \_ \to T}{\Xi \vdash \nu(\Xi_1).(C \parallel n\langle \mathsf{let}\, x{:}T = \mathsf{claim}@(n', \_)\,\mathsf{in}\, t\rangle) \xrightarrow{a} \acute{\Xi} \vdash \nu(\Xi_1).(C \parallel C(\Xi') \parallel n\langle \mathsf{let}\, x{:}T = v\,\mathsf{in}\, t\rangle)} \ \textsc{ClaimI}_1$$

$$\frac{\Delta \vdash n' = \bot}{\Xi \vdash \nu(\Xi_1).(C \parallel n\langle \mathsf{let}\, x{:}T = \mathsf{claim}@(n', o)\,\mathsf{in}\, t\rangle) \rightsquigarrow \Xi \vdash \nu(\Xi_1).(C \parallel n\langle \mathsf{let}\, x : T = \mathsf{release}(o); \mathsf{get}@n'\,\mathsf{in}\,\mathsf{grab}(o); t\rangle)} \ \textsc{ClaimI}_2$$

$$\frac{a = n'\langle \mathit{get}(v)\rangle? \qquad \Delta \vdash n' = v \qquad \Xi \vdash \lfloor a \rfloor : \_ \to T}{\Xi \vdash \nu(\Xi_1).(C \parallel n\langle \mathsf{let}\, x{:}T = \mathsf{claim}@(n', \_)\,\mathsf{in}\, t\rangle) \xrightarrow{a} \Xi \vdash \nu(\Xi_1).(C \parallel C(\Xi') \parallel n\langle \mathsf{let}\, x{:}T = v\,\mathsf{in}\, t\rangle)} \ \textsc{ClaimI}_3$$

$$\frac{a = \nu(\Xi').\, n'\langle \mathit{get}(v)\rangle? \qquad \acute{\Xi} = \Xi + a \qquad \Delta \vdash n' = \bot \qquad \acute{\Xi} \vdash \lfloor a \rfloor : \_ \to T}{\Xi \vdash \nu(\Xi_1).(C \parallel n\langle \mathsf{let}\, x{:}T = \mathsf{get}@n'\,\mathsf{in}\, t\rangle) \xrightarrow{a} \acute{\Xi} \vdash \nu(\Xi_1).(C \parallel C(\Xi') \parallel n\langle \mathsf{let}\, x{:}T = v\,\mathsf{in}\, t\rangle)} \ \textsc{GetI}_1$$

$$\frac{a = n'\langle \mathit{get}(v)\rangle? \qquad \Delta \vdash n' = v \qquad \Xi \vdash \lfloor a \rfloor : \_ \to T}{\Xi \vdash \nu(\Xi_1).(C \parallel n\langle \mathsf{let}\, x{:}T = \mathsf{get}@n'\,\mathsf{in}\, t\rangle) \xrightarrow{a} \Xi \vdash \nu(\Xi_1).(C \parallel C(\Xi') \parallel n\langle \mathsf{let}\, x{:}T = v\,\mathsf{in}\, t\rangle)} \ \textsc{GetI}_2$$

$$\frac{a = \nu(\Xi').n\langle \mathit{get}(v)\rangle! \quad \Xi' = \mathit{fn}(\lfloor a \rfloor) \cap \Xi_1 \quad \acute{\Xi}_1 = \Xi_1 \setminus \Xi' \quad \acute{\Xi} = \Xi + a}{\Xi \vdash \nu(\Xi_1).(C \parallel C(\Xi') \parallel n\langle v \rangle) \xrightarrow{a} \acute{\Xi} \vdash \nu(\acute{\Xi}_1).(C \parallel n\langle v \rangle)} \ \textsc{GetO}_1 \qquad \frac{a = n\langle \mathit{get}(v)\rangle! \quad \Theta \vdash n = v}{\Xi \vdash C \xrightarrow{a} \Xi \vdash C} \ \textsc{GetO}_2$$

Table 10
External steps

Finally, we characterize the *initial* configuration. Initially, the component contains at most one initial activity and no objects. More precisely, given that $\Xi_0 \vdash C_0$ is the initial judgment, then C_0 contains no objects. Concerning the threads: initially exactly one thread is executing, either at the component side or at the environment side. The distinction is made at the interface that initially either $\Theta_0 \vdash n$ or $\Delta_0 \vdash n$, where n is the only thread name in the system.

3 Interface behavior

Next we characterize the possible ("legal") *interface behavior* as interaction traces between component and environment. Half of the work has been done already in the definition of the external steps in Table 10: 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 char-

$$\Xi \vdash \varepsilon : \mathit{trace} \qquad \text{L-EMPTY}$$

$$\dfrac{\begin{array}{c} a = \nu(\Xi').\, n\langle \mathit{call}\ o.l(\vec{v})\rangle? \quad \acute{\Xi} = \Xi + a \quad (\Xi' \vdash n \vee \Delta \vdash n : [\,]^{+-}) \\ \acute{\Xi} \vdash o.l? : \vec{T} \to \_ \quad \acute{\Xi} \vdash \lfloor a \rfloor : \vec{T} \to \_ \quad \acute{\Xi} \vdash s : \mathit{trace} \end{array}}{\Xi \vdash a\, s : \mathit{trace}}\ \text{L-CALLI}$$

$$\dfrac{a = \nu(\Xi').n\langle \mathit{get}(v)\rangle? \quad \acute{\Xi} = \Xi + a \quad \Delta \vdash n = \bot \quad \acute{\Xi} \vdash \lfloor a \rfloor : \_ \to T \quad \acute{\Xi} \vdash s : \mathit{trace}}{\Xi \vdash a\, s : \mathit{trace}}\ \text{L-GETI}_1$$

$$\dfrac{a = n\langle \mathit{get}(v)\rangle? \quad \Delta \vdash n = v \quad \Xi \vdash s : \mathit{trace}}{\Xi \vdash a\, s : \mathit{trace}}\ \text{L-GETI}_2$$

Table 11
Legal traces (dual rules omitted)

acterize 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.

3.1 Legal traces system

The rules of Table 11 specify legality of traces. We use the same conventions and notations as for the operational semantics (cf. Notation 2.11). The judgments in the derivation system are of the form

$$\Xi \vdash s : \mathit{trace}\ . \tag{10}$$

We write $\Xi \vdash t : \mathit{trace}$, if there exists a derivation according to the rules of Table 11 with an instance of L-EMPTY as axiom. The empty trace is always legal (cf. rule L-EMPTY), and distinguishing according to the first action a of the trace, the rules check whether a is possible. Furthermore, the contexts are updated appropriately, and the rules recur checking the tail of the trace. The rules are symmetric wrt. incoming and outgoing communication (the dual rules are omitted). Rule L-CALLI for incoming calls works completely analogously to the CALLI-rule in the semantics: the second premise updates the context Ξ appropriately with the information contained in a, premise $\Xi' \vdash n$ of L-CALLI assures that the identity n of the future, carrying out the call, is fresh and the two premises $\acute{\Xi} \vdash o.l? : \vec{T} \to \_$ and $\acute{\Xi} \vdash \lfloor a \rfloor : \vec{T} \to \_$ together assure that the transmitted values are well-typed (cf. Definition 2.12); the latter two checks correspond to the analogous premises for the external semantics in rule CALLI, except that the return type of the method does not play a role here. The L-GETI-rules for claiming a value work similarly. In particular the type checking of the transmitted value is done by the combination of the premises $\Delta \vdash n : [T]$ and $\acute{\Xi} \vdash \lfloor a \rfloor : \_ \to T$. As in the external semantics, we distinguish two cases, namely whether the value of the future has been incorporated in the interface already or not (rules L-GETI$_2$ and L-GETI$_1$). In both cases, the thread must be executing on the side of the environment for an incoming get. This is checked by the premise $\Delta \vdash n = \bot$ resp. by $\Delta \vdash n = v$. In case of L-GETI$_2$, where the value of the future has been incorporated as v into the interface information, the actual parameter of the get-label must, of course, be v. If not (for L-GETI$_1$), the transmitted argument value is arbitrary, apart from the fact that it must be consistent with the static typing requirements.

It remains to show that the behavioral description, as given by Table 11, actually does what it claims to do, to characterize the possible interface behavior of well-typed components. 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. In the formulation of subject reduction, we make sure that the write-permissions of the environment are not available for type-checking the component. We use $\lfloor \Delta \rfloor$ instead of Δ as assumption, were $\lfloor \_ \rfloor$ replaces each binding $n{:}[T]^{+-}$ in Δ by $n{:}[T]^{+}$.

Lemma 3.1 (Subject reduction) *If* $\lfloor \Delta \rfloor \vdash C : \Theta$ *and* $\Delta \vdash C : \Theta \stackrel{s}{\Longrightarrow} \acute{\Delta} \vdash \acute{C} : \acute{\Theta}$*, then* $\lfloor \acute{\Delta} \rfloor \vdash \acute{C} : \acute{\Theta}$.

Proof: By induction on the number of reduction steps. That internal steps preserve well-typedness, i.e., $\lfloor \Delta \rfloor \vdash C : \Theta \Longrightarrow \lfloor \acute{\Delta} \rfloor \vdash C : \Theta$, follows from the corresponding Lemma 2.9 for internal steps. That leaves the external reduction steps of Table 10.

Case: CALLI
We are given $\lfloor \Delta \rfloor \vdash C : \Theta$. The disjunctive premise of the rule distinguishes two sub-cases: 1) $\Xi' \vdash n$ (where Ξ' are the bindings carried along with the call-label, i.e., the thread name is transmitted freshly) or 2) $\Delta \vdash n : [\_]^{+-}$ (the thread is not transmitted freshly and the environment has write-permission before the step). Both are treated uniformly in the following argument. For the right-hand side of the transition, we need to show $\lfloor \acute{\Delta} \rfloor \vdash C' \parallel n\langle \mathsf{let}\, x{:}T = \mathsf{grab}(o); M.l(o)(\vec{v})\, \mathsf{in}\, \mathsf{release}(o); x \rangle : \acute{\Theta}$, where C' corresponds to C extended by new $n'\langle \bullet \rangle$-promises. According to the definition of context update (Definition 2.13), $\acute{\Xi} = \acute{\Delta}, \acute{\Theta}$, where $\acute{\Theta} = \Theta, \Sigma'', n{:}[T]^{+}$ and where Σ'' contains bindings $n'{:}[T']^{+-}$ for those references transmitted with read-write permission as argument of the call (see the right-hand of equation (8)). The assumption context $\acute{\Delta}$ for $\acute{C}$ after the step (by the left-hand of the same equation) is of the form $(\Delta, \Xi') \setminus \Sigma'' \setminus n{:}[T]^{+-}$. So for the new thread n at component side, we need to show that

$$\lfloor (\Delta, \Xi') \setminus \Sigma'' \setminus n{:}[T]^{+-} \rfloor \vdash C \parallel C(\Sigma'') \parallel n\langle t' \rangle : \Theta, \Sigma'', n{:}[T]^{+} \qquad (11)$$

with t' given as $\mathsf{let}\, x{:}T = \mathsf{grab}(o); M.l(o)(\vec{v})\, \mathsf{in}\, \mathsf{release}(o); x$. To derive (11), using a number of instances of T-PAR in the last derivation steps, gives

$$\dfrac{\dfrac{\tilde{\Delta}, \lfloor \Sigma'' \rfloor, n{:}[T]^+ \vdash C : \Theta \qquad \tilde{\Delta}, n{:}[T]^+, \lfloor \Theta \rfloor \vdash C(\Sigma'') : \Sigma'' \qquad \dfrac{\tilde{\Delta}, \lfloor \Theta \rfloor, \Sigma'', n{:}[T]^+ \vdash t' : T}{\tilde{\Delta}, \lfloor \Theta \rfloor, \Sigma'' \vdash n\langle t' \rangle : n{:}[T]^+}}{\vdots}}{\lfloor (\Delta, \Xi') \setminus \Sigma'' \setminus n{:}[T]^{+-} \rfloor \vdash C \parallel C(\Sigma'') \parallel n\langle t' \rangle : \Theta, \Sigma'', n{:}[T]^+} \tag{12}$$

where $\tilde{\Delta}$ abbreviates the assumption context $\lfloor (\Delta, \Xi') \setminus \Sigma'' \setminus n{:}[T]^{+-} \rfloor$ from (11). Note, how the write permissions from Σ'' in commitment of the conclusion at the bottom are split among the three subgoals. All write-permissions are given to the assumptions of $n\langle t' \rangle$, whereas C can assume only read-access (cf. T-PAR and the definition of $\oplus$ from Definition 2.1). The context Θ is split similarly. The left open goal can be rephrased as $\lfloor \Delta \rfloor, \lfloor \Xi' \rfloor, n{:}[T]^+ \vdash C : \Theta$ and can be discharged using the given $\lfloor \Delta \rfloor \vdash C : \Theta$ and weakening. The open goal in the middle follows directly from an appropriate number of instances of T-NTHREAD$'$.

Remains the right-upper subgoal $\lfloor (\Delta, \Xi') \setminus \Sigma'' \setminus n{:}[T]^{+-} \rfloor, \lfloor \Theta \rfloor, \Sigma'', n{:}[T]^+ \vdash \mathsf{let}\, x{:}T = t' : T$ (with $\tilde{\Delta}$ expanded). Note that, apart from the write-permissions, the complicated type context corresponds to Δ, Ξ', Θ, or more formally

$$\lfloor (\lfloor (\Delta, \Xi') \setminus \Sigma'' \setminus n{:}[T]^{+-} \rfloor, \lfloor \Theta \rfloor, \Sigma'', n{:}[T]^+) \rfloor = \lfloor \Delta, \Xi', \Theta \rfloor \tag{13}$$

Intuitively, it means, t' must be checked with all name bindings available from Δ and Θ plus the ones, which scope is exchanged in Ξ' as part of the label. No *write*-permissions, however, can be used to type-check t' *except* those being transmitted by the argument of the call and which are kept in Σ'' (the context Σ'' is the only part of t' typing context *not* being stripped off the write-permissions by $\lfloor \_ \rfloor$).

Note that the meta-mathematical notation $M.l(o)(\vec{v})$ in t' stands for $t_{body}[o/s][\vec{v}/\vec{x}]$, i.e., the method body with the self-parameter s substituted by the callee's identity and with the formal parameter replaced by the actual ones. Now, the well-typedness of the pre-configuration $\lfloor \Delta \rfloor \vdash C : \Theta$ together with the premise $\acute{\Xi} \vdash o.l : ?\vec{T} \to T$ of CALLI (cf. Definition 2.12) implies that C is of the form $C' \parallel c[(\ldots, l = \varsigma(s{:}c).\lambda(\vec{x} : \vec{T}).t_{body}, \ldots)]$, and further that $\lfloor \Delta \rfloor \vdash C : \Theta$ has $\vec{x}{:}\vec{T}; \lfloor \Delta \rfloor, \Theta \vdash t_{body} : T$ as sub-goal. From that, the remaining mentioned subgoal of derivation (12) follows by T-LET, T-GRAB, preservation of typing under substitution, T-RELEASE, and the axiom T-VAR.

Case: CALLO
We are given $\Delta \vdash \nu(\Xi_1).(C \parallel n\langle \bullet \rangle \parallel n'\langle \mathsf{let}\, x{:}T = \mathsf{bind}\, o.l(\vec{v}) : T \hookrightarrow n\ \mathsf{in}\, t \rangle) : \Theta$ before the step and $\acute{\Delta} \vdash \nu(\acute{\Xi}_1).(C \parallel n'\langle \mathsf{let}\, x : T = n\ \mathsf{in}\, t \rangle) : \acute{\Theta}$ afterwards, with $C = C' \parallel n_1\langle \bullet \rangle \parallel \ldots \parallel n_k\langle \bullet \rangle$. By one of the premises of rule CALLO we know $\Delta \vdash o$, i.e., object o is an environment object.[5] That the name o refers to an object is assured

[5] We do not allow cross-border instantiation in this paper, i.e., the component is not al-

by the type system and the assumption that the pre-configuration is well-typed. By inverting the rules T-Nu, T-Par, T-NThread, T-Let, and T-Bind, we get:

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\ldots \quad \Xi' = \lfloor\Delta\rfloor,\Xi_1,\Theta \quad \acute{\Xi}' = \Xi' \setminus (\vec{v}{:}\vec{T}, n{:}[T]^{+-})}{;\lfloor\Delta\rfloor,\Xi_1,\Theta \vdash \mathsf{bind}\, o.l(\vec{v}) : T \hookrightarrow n : T :: \acute{\Delta}'}\ \textsc{T-Bind} \qquad x{:}T;\acute{\Xi}' \vdash t : T' :: \ldots}{;\lfloor\Delta\rfloor,\Xi_1,\tilde{\Theta},n'{:}[T']^{+} \vdash \mathsf{let}\, x{:}T = \mathsf{bind}\, o.l(\vec{v}) : T \hookrightarrow n\, \mathsf{in}\, t : T'}}{\lfloor\Delta\rfloor,\Xi_1,\tilde{\Theta} \vdash n'\langle \mathsf{let}\, x{:}T = \mathsf{bind}\, o.l(\vec{v}) : T \hookrightarrow n\, \mathsf{in}\, t\rangle : n'{:}[T']^{+}}\ \textsc{T-Par}, \textsc{T-Nu}\ldots}{\vdots}}{\lfloor\Delta\rfloor \vdash \nu(\Xi_1).(C \parallel n'\langle \mathsf{let}\, x : T = \mathsf{bind}\, o.l(\vec{v}) : T \hookrightarrow n\ \mathsf{in}\, t\rangle \parallel n\langle\bullet\rangle) : \Theta}
$$

Note that t in the left-upper leaf is type-checked in the context Ξ', which corresponds to $\Xi' = \lfloor\Delta\rfloor,\Xi_1,\tilde{\Theta},n'{:}[T']^{+}$ with those write-permissions *removed* that are transmitted via the arguments of the method l (cf. rule T-Bind).

To derive well-typedness of the post-configuration, we distinguish two sub-cases, namely whether 1) the promise n is known at the interface before the step or 2) it is hidden still. In the first case, we have $\Theta \vdash n : T'$ with $T' = [T]^{+-}$ (as a consequence of the fact that the configuration is well-typed), or more precisely, $\Theta = \tilde{\Theta}',n{:}[T]^{+-},n'{:}[T']^{+}$. The derivation of well-typedness of the post-configuration $\lfloor\acute{\Delta}\rfloor \vdash \nu(\acute{\Xi}_1).(C' \parallel n'\langle \mathsf{let}\, x : T = n\ \mathsf{in}\, t\rangle) : \acute{\Theta}$ works as follows:

$$
\dfrac{\dfrac{\dfrac{\dfrac{;\lfloor\acute{\Delta}\rfloor,\acute{\Xi}_1,\acute{\Theta} \vdash n : T \qquad x{:}T;\lfloor\acute{\Delta}\rfloor,\acute{\Xi}_1,\acute{\Theta} \vdash t : T'}{;\lfloor\acute{\Delta}\rfloor,\acute{\Xi}_1,\acute{\Theta} \vdash \mathsf{let}\, x{:}T = n\, \mathsf{in}\, t : T'}\ \textsc{T-Let}}{\lfloor\acute{\Delta}\rfloor,\acute{\Xi}_1,\tilde{\acute{\Theta}} \vdash n'\langle \mathsf{let}\, x{:}T = n\, \mathsf{in}\, t\rangle : n'{:}[T']^{+}}\ \textsc{T-NThread}}{\vdots}\ \textsc{T-Par}, \textsc{T-Nu}\ldots}{\lfloor\acute{\Delta}\rfloor \vdash \nu(\acute{\Xi}_1).(C' \parallel n'\langle \mathsf{let}\, x : T = n\ \mathsf{in}\, t\rangle) : \acute{\Theta}}
$$

where $\acute{\Theta} = \tilde{\acute{\Theta}},n'{:}[T']^{+}$. Using the premises of the reduction rule CallO, that relates the different binding contexts mentioned in the step (i.e., $\acute{\Xi}_1 = \Xi_1 \setminus \Xi'$, where Ξ' are the bindings mentioned in the call labels), $\Delta' = \Delta,\Xi',n{:}[T]^{+}$ (as stipulated by rule CallO's premise $\acute{\Xi} = \Xi + a$ and given by Definition 2.13, especially equation (8)). Note that (8) is formulated for incoming communication, i.e., used dually here, and that in the considered subcase, we assume that n is known at the interface before the step, i.e., $\Theta \vdash n{:}[T]^{+-}$, as agreed upon earlier. It is straightforward to see that the combined context Δ,Ξ_1,Θ equals $\acute{\Theta},\acute{\Xi}_1,\acute{\Delta}$, with the exception, that the former contains $n{:}[T]^{+-}$ (as part of Θ) and the latter only $n{:}[T]^{+}$ (as part of $\acute{\Delta}$. Cf. especially by (8)). Furthermore, considering $\lfloor\Delta\rfloor$ and $\lfloor\acute{\Delta}\rfloor$ instead of Δ and $\acute{\Delta}$:

$$\lfloor\acute{\Delta}\rfloor,\acute{\Xi}_1,\acute{\Theta} = (\lfloor\Delta\rfloor,\Xi_1,\Theta) \setminus (n{:}[T]^{+-},\vec{v}{:}\vec{T}) \tag{14}$$

lowed to instantiate environment objects and vice versa.

where $\vec{v}{:}\vec{T}$ is given as mentioned in the left-upper leaf if the first derivation tree and as defined by the premise of T-BIND (these bindings correspond to the Σ'' used in equation (8) and represent the write-permissions transmitted by the call-label from the component to the environment). This discharges the top-left subgoal of the derivation. The second sub-case with $\Xi_1 \vdash n{:}[T]^{+-}$ works analogously.

Case: CLAIM$_1$
The core of the type preservation here is to assure that the claim-statement in the pre-configuration and the transmitted value v in the post-configuration are of the same appropriate type T. Well-typedness of the pre-configuration implies with $\mathsf{claim}@(n',o,)$ of type T, that the reference n' is of type $[T]^{+}$. The third premise of CLAIMI$_1$ states $\acute{\Xi} \vdash \lfloor a \rfloor : \_ \to T$, which implies with Definition 2.12, especially rule LT-GETI of Table 9, that also v is of type T, as required.

Case: CLAIM$_2$
By inverting the type rules T-NU, T-PAR, T-LET and T-CLAIM for the pre-configuration of the step, and by using the same typing rules (except T-CLAIM) plus T-GET, T-RELEASE, and T-GRAB.

The remaining rules work similarly. □

Lemma 3.2 (Soundness of abstractions) *If $\Xi_0 \vdash C$ and $\Xi_0 \vdash C \overset{t}{\Longrightarrow}$, then $\Xi_0 \vdash t$: trace.*

Proof: By induction on the number of steps in $\overset{t}{\Longrightarrow}$. The base case of zero steps (which implies $t = \varepsilon$) is immediate, using L-EMPTY. The induction for internal steps of the form $\Xi \vdash C \Longrightarrow \Xi \vdash \acute{C}$ follow by subject reduction for internal steps from Lemma 2.9; in particular, internal steps do not change the context Ξ. Remain the external steps of Table 10. 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 for legality, already, i.e., the premises of the operational rules have their counterparts in the rules for legal traces.

Case: CALLI
Immediate, as the premises of L-CALLI coincide with the ones of CALLI.

Case: CLAIM$_1$ and GET$_1$
The two cases are covered by rule L-GET$_1$, which has the same premises as the operational rules.

Case: CLAIM$_2$
Trivial, as the step is an internal one.

Case: CLAIM$_3$ and GET$_2$
The two cases are covered by L-GET$_2$.

The cases for outgoing communication are slightly more complex, as the label in the operational rule is not type-checked or checked for well-formedness as for incoming communication and as is done in the rules for legality.

Case: CALLO
We need to check whether the premises of L-CALLO, the dual to L-CALLI of Table 11, are satisfied. By assumption, the pre-configuration

$$\Xi \vdash \nu(\Xi_1).(C \parallel n'\langle \mathsf{let}\, x{:}T = \mathsf{bind}\, o.l(\vec{v}) : T \hookrightarrow n \ \mathsf{in}\, t\rangle) \tag{15}$$

is well-typed. For thread name n this implies, it is bound either in Ξ or in Ξ_1, more precisely, either $\Theta \vdash n : [T]^{+-}$ (it is public interface information that the component has write-permission for n) or $\Xi_1 \vdash n : [T]^{+-}$ (the name n is not yet known in the environment before the communication). In the latter situation we obtain $\Xi' \vdash n : [\_]^{+-}$ by the premise $\Xi' = fn(\lfloor a \rfloor) \cap \Xi_1$ of CALLO. Thus, the third premise $\Xi' \vdash n \vee \Theta \vdash n : [\_]^{+-}$ of L-CALLO is satisfied. We furthermore need to check whether the label is type-correct (checked by premises nr. 4 and 5 or L-CALLO). Its easy to check that the label is well-formed (cf. the first part of Definition 2.12). The first premise of the check of equation (6), that the receiving object o is an environment object, is directly given by the premise $\Delta \vdash o$ of CALLO. That the object o supports a method labeled l (of type $\vec{T} \rightarrow T$) follows from the fact that the pre-configuration of the call-step is well-typed. So this gives L-CALLO's premise $\acute{\Xi} \vdash o.l! : \vec{T} \rightarrow T$. Remains the type check $\acute{\Xi} \vdash \lfloor a \rfloor : \vec{T} \rightarrow \_$ (checking that the transmitted values $\vec{v}$ are of the expected type $\vec{T}$), which again follows from well-typedness of equation (15) (especially inverting T-BIND).

The remaining cases work similarly. □

Remark 3.3 (Comparison with reentrant threading) *In a multi-threaded setting with synchronous method calls (see for instance [4] [62]), the definition of legal traces is more complicated. Especially, to judge whether a trace s is possible requires referring to the past. I.e., instead of judgments of the form of equation (10), the check for legality with synchronous calls uses judgments of the form:*

$$\Xi \vdash r \rhd s : \mathit{trace}\ ,$$

reading "after history *r (and in the context Ξ), the trace s is possible". This difference has once more to do with reentrance, resp. with the absence of this phenomenon here. In the threaded case, where, e.g., an outgoing call can be followed by a subsequent incoming call as a "call-back". To check therefore, whether a call or a return is possible as a next step involves checking the proper nesting of the call- and return labels. This nesting requirement (also called the balance condition) degenerates here in the absence of call-backs to the given requirement that* *each call uses a fresh (future) identity and that each get-label (taking the role of the return label in the multithreaded setting) is preceded by exactly one matching preceding call. This can be judged by $\Delta \vdash n : [\_]$ or $\Theta \vdash n : [\_]$ (depending on whether we are dealing with incoming or outgoing get-labels) and especially, no reference to the history of interface interactions is needed.* □

Remark 3.4 (Monitors) *The objects of the calculus act as monitors as they allow only one activity at a time inside the object. For the operational semantics of Section 2.3, the lock-taking is part of the* internal *steps. In other words, the handing-over of the call at the interface and the actual entry into the synchronized method body is* non-atomic*, and at the interface, objects are* input-enabled.

This formalization therefore resembles the one used for the interface description of Java*-like reentrant monitors in [3]. To treat the interface interaction and actual lock-grabbing as non-atomic leads to a clean separation of concerns of the component and of the environment. In [3], this non-atomicity, however, gives rise to quite complex conditions characterizing the legal interface behavior. In short, in the setting of [3], it is non-trivial to characterize exactly those situations, when the lock of the object is* necessarily *taken by one thread which makes certain interactions of other threads impossible. This characterization is non-trivial especially as the interface interaction is non-atomic.*

Note, however, that these complications are not present *in the current setting with active objects, even if the objects act as monitors like in [3]. The reason is simple: there is no* need *to capture situations when the lock is taken. In* Java*, the synchronization behavior of a method is part of the* interface *information. Concretely, the* synchronized*-modifier of* Java*, specifies that the method's body is executed atomically in that object without interference of other*[6] *threads, assuming that all other methods of the callee are synchronized, as well. Here, in contrast, there is no interface information that guarantees that a method body is executed atomically. In particular, the method body can give up the lock temporarily via the* `suspend`*-statement, but this fact is not reflected in the interface information here. This absence of knowledge simplifies the interface description considerably.* □

4 Conclusion

We presented an open semantics describing the interface behavior of components in a concurrent object-oriented language with futures and promises. The calculus corresponds to the core of the *Creol* language, including classes, asynchronous method calls, the synchronization mechanism, and futures, and extended by promises. Con-

[6] Note that a thread can "interfere" in that setting with itself due to recursion and reentrance.

centrating on the black-box interface behavior, however, the interface semantics is, to a certain extent, independent of the concrete language and is characteristic for the mentioned features; for instance, extending *Java* with futures (see also the citations below) would lead to a quite similar formalization (of course, low level details may be different). Concentrating on the concurrency model, certain aspects of *Creol* have been omitted here, most notably inheritance and safe asynchronous class upgrades.

Related work

The general concept of "delayed reference" to a result of a computation to be yet completed is quite old. The notion of futures was introduced by Baker and Hewitt [13], where (`future` e) denotes an expression executed in a separate thread, i.e., concurrently with the rest of the program. As the result of the e is not immediately available, a *future variable* (or future) is introduced as placeholder, which will eventually contain the result of e. In the meantime, the future can be passed around, and when it is accessed for reading ("touched" or "claimed"), the execution suspends until the future value is available, namely when e is evaluated. The principle has also been called *wait-by-necessity* [16][17]. Futures provide, at least in a purely functional setting, an elegant means to introduce concurrency and *transparent* synchronization simply by accessing the futures. They have been employed for the parallel *Multilisp* programming language [38].

Indeed, quite a number of calculi and programming languages have been equipped with concurrency using future-like mechanisms and asynchronous method calls. Flanagan and Felleisen [31] [29] [30] present an operational semantics, based on evaluation contexts, for a λ-calculus with futures. The formalization is used for analysis and optimization to eliminate superfluous dereferencing ("touches") of future variables. The analysis is an application of a set-based analysis and the resulting transformation is known as touch optimization. Moreau [53] presents a semantics of Scheme equipped with futures and control operators. *Promises* is a mechanism quite similar to futures and actually the two notions are sometimes used synonymously. They have been proposed in [50]. A language featuring both futures and promises as separate concepts, is *Alice ML* [9][47][61].

[55] presents a concurrent call-by-value λ-calculus with reference cells (i.e., a non-purely functional calculus with an imperative part and a heap) and with futures (λ_{fut}), which serves as the core of *Alice ML* [9] [60] [47]. Certain aspects of that work are quite close to the material presented here. In particular, we were inspired by using a type system to avoid fulfilling a promise twice (in [55] called handle error). There are some notable differences, as well. The calculus incorporates futures and promises into a λ-calculus, such that functions can be executed in parallel. In contrast, the notion of futures here, in an object-oriented setting, is coupled to the asynchronous execution of methods. Furthermore, the object-oriented setting

here, inspired by *Creol*, is more high-level. In contrast, λ_{fut} relies on an atomic test-and-set operation when accessing the heap to avoid atomicity problems. Besides that, they formalize promises using the notion of *handled* futures, i.e., the two roles of a promise, the writing- and the reading part, are represented by two different references, where the *handle* to the futures represents the writing-end. Apart from that, [55] are not concerned with giving an open semantics as here. On the other hand, the paper investigates the role of the heap and the reference cells, and gives a formal proof that the *only* source of non-determinism by race conditions in their language actually are the reference cells and without those, the language becomes (uniformly) confluent. [7] Recently, an observational semantics for the (untyped) λ_{fut}-calculus has been developed in [54]. The observational equivalence is based on may- and must-program equivalence, i.e., two program fragments are considered equivalent, if, for all observing environments, they exhibit the same necessary and potential convergence behavior.

Futures have also been investigated in the object-oriented paradigm. For instance, the object-oriented language Scala [56] has recently been extended [37] by actor-based concurrency, offering futures and promises as part of the standard library. The futures and promises are inspired by their use in *Alice ML*. In *Java* 5, *futures* have been introduced as part of the `java.util.concurrent` package. As *Java* does not support futures as a core mechanism for parallelism, they are introduced in a library. Dereferencing of a future is done explicitly via a `get`-method (similarly to this paper). A recent paper [68] introduces *safe* futures for *Java*. The safe concept is intended to make futures and the related parallelism *transparent* and in this sense goes back to the origins of the concept: introducing parallelism via futures does not change the program's meaning. While straightforward and natural in a functional setting, safe futures in an object-oriented and thus state-based language such as *Java* require more considerations. The paper introduces a semantics which guarantees safe, i.e., transparent, futures by deriving restrictions on the scheduling of parallel executions and uses object versioning. The futures are introduced as an extension of Featherweight *Java* (*FJ*) [39], a core object calculus, and is implemented on top of *Jikes* RVM [10,15]. Pratikakis et. al. [58] present a constraint-based static analysis for (transparent) futures and proxies in *Java*, based on type qualifiers and qualifier inference [32]. Also this analysis is formulated as an extension of *FJ* by type qualifiers. Similarly, Caromel et. al. [20][19][18] tackle the problem to provide *confluent*, i.e., effectively deterministic system behavior for a concurrent object calculus with futures (asynchronous sequential processes, *ASP*,

[7] Uniform confluence is a strengthening of the more well-known notion of (just ordinary) confluence; it corresponds to the diamond property of the *one-step* reduction property. For standard reduction strategies of a purely functional λ-calculus, only confluence holds, but not uniform confluence. However, the non-trivial "diamonds" in the operational semantics of λ_{fut} are caused not by different redexes within one λ-term (representing one thread), but by redexes from different threads running in parallel, where the reduction strategy per thread is deterministic (as in our setting, as well).

an extension of Abadi and Cardelli's imperative, untyped object calculus impς [1]) and in the presence of imperative features. The *ASP* model is implemented in the *ProActive Java*-library [21]. The fact, that *ASP* is derived from some (sequential, imperative) object-calculus, as in the formalization here, is more a superficial or formal similarity, in particular when being interested in the interface behavior of concurrently running objects, where the inner workings are hidden anyway. Apart from that there are some similarities and a number of differences between the work presented here and *ASP*. First of all, both calculi are centered around the notion of first-class futures, yielding active objects. The treatment, however, of getting the value back, is done differently in [18]. Whereas here, the client must explicitly claim a return value of an asynchronous method, if interested in the result, the treatment of the future references is done *implicitly* in *ASP*, i.e., the client blocks if he performs a strict operation on the future (without explicit syntax to claim the value). Apart from that, the object model is more sophisticated, in that the calculus distinguishes between active and passive objects. Here, we simple have objects, which can behave actively or passively (reactively), depending on the way they are used. In *ASP*, the units of concurrency are the explicitely activated active objects, and each passive one is owned and belongs to exactly one active one. Especially, passive objects do not directly communicate with each other across the boundaries of concurrent activity, all such communication between concurrent activities is mediated and served by the active objects.

Related to that, a core feature of *ASP*, not present here, is the necessity to specify (also) the *receptive behavior* of the active object, i.e., in which order it is willing to process or *serve* incoming messages. The simplest serve strategy would be the willingness to accept all messages and treat them in a first-come, first-serve manner, i.e., a input-enabled FIFO strategy on the input message queue. The so-called *serve*-method is the dedicated activity of an active object to accept and schedule incoming method calls. Typically, as for instance in the FIFO case, it is given an an non-terminating process, but it might also terminate, in which case the active object together with the passive objects it governs, becomes superfluous: an active object which does no service any longer does not become a passive data structure, but can no longer react in any way.

As extension of the core *ASP* calculus, [18, Chapter 10] treats *delegation* that bears some similarities with the promises here. By executing the construct $delegate(o.l(\vec{v}))$ (using our notational conventions), a thread n hands over the permission and obligation to provide eventually a value for the future reference n to method l of object o, thereby losing that permission itself. That corresponds to executing $\mathsf{bind}\, o.l(\vec{v}) : T \hookrightarrow n$. Whereas in our setting, we must use a yet-unfulfilled promise n for that purpose, the delegation operator in *ASP* just (re-)uses the current future for that. Consequently, *ASP* does not allow the creation of promises independently from the implicit creation when asynchronously calling a method, as we do with the $\mathsf{promise}\, T$ construct. In this sense, the promises here are more general, as they allow to profit from delegation and have the promise as first-class entity, i.e., the programmer can pass it around, for instance, as argument of methods. This, on the other hand, requires a more elaborate type system to avoid write errors on promises. This kind of error, fulfilling a promise twice, is avoided in the delegate-construct of *ASP* not by a type system, but by construction, in that the delegate-construct must be used only at the end of a method, so that the delegating activity cannot write to the future/promise after it has delegated the permission to another activity.

Further uses of futures for *Java* are reported in [51] [46] [59] [65] [64]. Futures are also integral part of *Io* [40] and *Scoop* (simple concurrent object-oriented programming) [24] [12] [52], a concurrent extension of *Eiffel*. Both languages are based on the active objects paradigm.

Benton et. al. [14] present polyphonic C$^\sharp$, adding concurrency to C$^\sharp$, featuring asynchronous methods and based on the join calculus [33] [34]. Polyphonic C$^\sharp$allows methods to be declared as being asynchronous using the async keyword for the method type declaration. Besides that, polyphonic C$^\sharp$ supports so-called *chords* as synchronization or join pattern. With similar goals, *Java* has been extended by join patterns in [41] [42].

In the context of *Creol*, de Boer et. al. [27] present a formal, operational semantics for the language and extend it by *futures* (but not promises). Besides the fact, that both operational semantics ultimately formalize a comparable set of features, there are, at a technical level, a number of differences. For once, here, we simplified the language slightly mainly in two respects (apart from making it more expressive in adding promises, of course). We left out the "interleaving" operators $|||$ and $///$ of [27] which allows the user to express interleaving concurrency *within* one method body. Being interested in the observable interface behavior, those operations are a matter of internal, hidden behavior, namely leading to non-deterministic behavior at the interface. Since objects react non-deterministically anyhow, namely due to race conditions present independently of $|||$ and $///$, those operators have no impact on the possible traces at the interface. The operators might be useful as abstractions for the programmer, but without relevance for the interface traces, and so we ignored them here. Another simplification, this time influencing the interface behavior, is how the programmer can claim the value of a future. This influences, as said, the interface behavior, since the component may fetch the value of a future being part of the environment, or vice versa. Now, the design of the *Creol*-calculus in [27] is more liberal wrt. what the user is allowed to do with future references. In this paper, the interaction is rather restricted: if the client requests the value using the claim-operation, there are basically only two reactions. If the future computation has already been completed, the value is fetched and the client continues; otherwise it blocks until, if ever, the value is available. The bottom line is, that the client, being blocked, can never *observe* that the value is yet absent. The calculus of [27], in contrast, permits the user to *poll* the future reference directly, which gives the freedom to decide, *not* to wait for the value if not yet available. Incorporating such a construct into the language makes the *absence* of the value for a future refer-

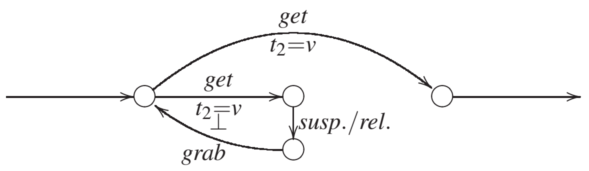

Fig. 3. Claiming a future (busy wait)

ence observable and would complicate the behavioral interface semantics to some extent. This is also corroborated by the circumstance that the expressive power of explicit polling quite complicates the proof theory of [27] (see also the discussion in the conclusion of [27]). This is not a coincidence, since one crux of the complete Hoare-style proof systems such as in [27] is to internalize the (ideally observable) behavior into the program state by so-called auxiliary variable. In particular recording the past interaction behavior in so-called history variables is, of course, an internalization of the interface behavior, making it visible to the Hoare-assertions. As a further indication that allowing to poll a future quite adds expressivity to the language is the observation that adding a poll-operation to *ASP*, destroys a central property of *ASP*, namely confluence, as is discussed in [18, Chapter 11].

Apart from that, the combination of claiming a futures, the possibility of polling a future, and a general await-statement complicates the semantics of claiming a future: in [27], this is done by *busy-waiting*, which in practice one intends to avoid. So instead of the behavior described in Figure 1, the formalization in [27] behaves as sketched in Figure 3.

After an unsuccessful try to obtain a value of future, the requesting thread is suspended and loses the lock. In order to continue executing, the blocked thread needs two resources: the value of the future, once it is there, plus the lock again. The difference of the treatment in Figure 1 and the one of Figure 3 for [27] is the order in which the requesting thread attempts to get hold of these two resources: our formalization first check availability of the future and afterwards re-gains the lock to continue, whereas [27] do it vice versa, leading to busy wait. The reason why it is sound to copy the future value into the local state space without already having the lock again (Figure 1) is , of course, that, once arrive, the future value remains stable and available.

In addition, our work differs also technically in the way, the operational semantics is represented. [27] formulated the (internal) operational semantics using evaluation contexts (as do, e.g., [55] for λ_{fut}), whereas we rely on a "reduction-style" semantics, making use of an appropriate notion of structural congruence. While largely a matter of taste, it seems to us that, especially in the presence of complicated synchronization mechanisms, for instance the ready queue representation of [27], the evaluation contexts do not give rise to an immediately more elegant specification of the reduction behavior. Admittedly, we ignored here the internal interleaving operators $|||$ and $///$, which quite contribute to the complexity of the evaluation contexts. Another technical difference, if you wish, concerns the way, the futures, threads, and objects are *represented* in the operational semantics, i.e., in the run-time syntax of the calculus. Different from our representation, their semantics makes the active-objects paradigm of *Creol* more visible: The activities are modeled as part of the object. More precisely, an object contains, besides the instance state, an explicit representation of the current activity (there called "process") executing "inside" the object plus a representation of the ready-queue containing all the activities, which have been *suspended* during their execution inside the object. The scheduling between the different activities is then done by juggling them in and out of the ready-queue at the processor release points. Here, in contrast, we base our semantics on a separate representation of the involved semantics concepts: 1) classes as *generators* of objects, 2) objects carrying in the instance variables the persistent *state* of the program, thus basically forming the heap, and 3), the *parallel* activities in the form of threads. While this representation makes arguably the active-object paradigm less visible in the semantics, it on the other hand separates the concepts in a clean way. Instead of an explicit local scheduler inside the objects, the access to the shared instance states of the objects is regulated by a simple, binary lock per object. So, instead of having two levels of parallelism —locally inside the objects and inter-object parallelism— the formalization achieves the same with just one conceptual level, namely: parallelism is between threads (and the necessary synchronization is done via object-locks). Additionally, our semantics is rather close to the object-calculi semantics for multi-threading as in *Java* (for instance as in [43] [44] or [62]). This allows to see the differences and similarities between the different models of concurrency, and the largely similar representation could allow are more formal comparison between the interface behaviors in the two settings.

The language *Cool* [22] [23] (concurrent, object-oriented language) is defined as an extension of C^{++} [63] for task-level parallelism on shared memory multi-processors. Concurrent execution in *Cool* is expressed by the invocation of parallel functions executing *asynchronously.* Unlike the work presented here, *Cool* contains future types, which correspond to the types of the form $[T]$ used here. Further languages supporting futures include ACT-1 [48] [49], concurrent *Smalltalk* [69] [73], and of course the influential actor model [8,36,7], *ABCL/1* [70] [71] (in particular the extension *ABCL/f* [66]).

We have characterized the behavioral semantics of open systems, similarly to the one presented here for futures and promises, in earlier papers, especially for object-oriented languages based on *Java*-like multi-threading and synchronous method calls, as in *Java* or $\mathrm{C}^\sharp$. The work [5] deals with thread classes and [4] with re-entrant monitors. In [62] the proofs of full abstraction for the sequential and multi-threaded cases of a class-based object-calculus can be found. Poetzsch-Heffter and Schäfer [57] present a behavioral interface semantics for a class-based object-oriented calculus, however without concurrency. The language, on the other hand, features an ownership-structured heap.

Future work

An obvious way to proceed is to consider more features of the *Creol*-language, in particular inheritance and subtyping. Incorporating inheritance is challenging, as it renders the system open wrt. a new form of interaction, namely the environment inheriting behavior from a set of component classes or vice versa. Also *Creol*'s mechanisms for dynamic class upgrades should be considered from a behavioral point of view (that we expect to be quite more challenging than dealing with inheritance). An observational, black-box description of the system behavior is necessary for the compositional account of the system behavior. Indeed, the legal interface description is only a first, but necessary, step in the direction of a compositional and ultimately fully-abstract semantics, for instance along the lines of [62]. Based on the interaction trace, it will be useful to develop a logic better suited for specifying the desired interface behavior of a component than enumerating allowed traces. Another direction is to use the results in the design of a black-box testing framework, as we started for *Java* in [26]. We expect that, with the theory at hand, it should be straightforward to adapt the implementation to other frameworks featuring futures, for instance, to the future libraries of *Java* 5.

Acknowledgements

We thank Einar Broch Johnsen, Marcel Kyas, Olaf Owe, and Gerardo Schneider for stimulating discussions on the topic. Furthermore we are thankful for the detailed comments and careful suggestions of the reviewers that helped improving the paper.

References

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

[2] E. Ábrahám, I. Grabe, A. Grüner, and M. Steffen. Behavioral Interface Description of an Object-Oriented Language with Futures and Promises. Technical Report 364, University of Oslo, Dept. of Computer Science, Oct. 2007.

[3] E. Ábrahám, A. Grüner, and M. Steffen. Abstract interface behavior of object-oriented languages with monitors. In R. Gorrieri and H. Wehrheim, editors, *FMOODS '06*, volume 4037 of *Lecture Notes in Computer Science*, pages 218–232. Springer-Verlag, 2006.

[4] E. Ábrahám, A. Grüner, and M. Steffen. Abstract interface behavior of object-oriented languages with monitors. *Theory of Computing Systems*, 2007. Published online 3. Oct. 2007.

[5] E. Ábrahám, A. Grüner, and M. Steffen. Heap-abstraction for open, object-oriented systems with thread classes. *Journal of Software and Systems Modelling (SoSyM)*,

2007. Published online first.

[6] ACM. *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) '99*, 1999. In *SIGPLAN Notices*.

[7] G. A. Agha. *Actors: a Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, 1986.

[8] G. A. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. Towards a theory of actor computation (extended abstract). In R. Cleaveland, editor, *Proceedings of CONCUR '92*, volume 630 of *Lecture Notes in Computer Science*, pages 565–579. Springer-Verlag, 1992.

[9] Alice project home page. `www.ps-uni-sb.de/alice`, 2006.

[10] B. Alpern, C. R. Attanasio, J. J. Barton, A. Cocchi, S. F. Hummel, D. Lieber, T. Ngo, M. Mergen, J. C. Sheperd, and S. Smith. Implementing Jalapeno in Java. In OOPSLA'99 [6], pages 313–324. In *SIGPLAN Notices*.

[11] P. America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1(4):366–411, 1989.

[12] V. Arslan, P. Eugster, P. Nienaltowski, and S. Vaucouleur. Scoop — concurrency made easy. In J. Kohlas, B. Meyer, and A. Schiper, editors, *Research Results of the DICS Program*, volume 4028 of *Lecture Notes in Computer Science*, pages 82–102. Springer, 2006.

[13] H. Baker and C. Hewitt. The incremental garbage collection of processes. *ACM Sigplan Notices*, 12:55–59, 1977.

[14] N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstraction for C#. *ACM Transactions on Programming Languages and Systems. Special Issue with papers from FOOL 9*, 2003.

[15] M. G. Burke, J.-D. Choi, S. F. Fink, D. Grove, M. Hind, V. Sarkar, M. Serranon, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeno dynamic optimizing compiler. In *Proceedings of the ACM Java Grande Conference, San Francisco*, pages 129–141, 1999.

[16] D. Caromel. Service, asynchrony and wait-by-necessity. *Journal of Object-Oriented Programming*, 2(4):12–22, Nov. 1990.

[17] D. Caromel. Towards a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, Sept. 1993.

[18] D. Caromel and L. Henrio. *A Theory of Distributed Objects. Asynchrony — Mobility — Groups — Components*. Springer-Verlag, 2005.

[19] D. Caromel, L. Henrio, and B. P. Serpette. Asynchronous sequential processes. Research Report RR-4753 (version 2), INRIA Sophia-Antipolis, May 2003.

[20] D. Caromel, L. Henrio, and B. P. Serpette. Asynchronous and deterministic objects. In *Proceedings of POPL '04*. ACM, Jan. 2004.

[21] D. Caromel, W. Klauser, and J. Vayssière. Towards seamless computing and metacomputing in Java. *Concurrency, Practice and Experience*, 10(11-13):1043–1061, 1998. ProActive available at `www.infria.fr/oasis/proactive`.

[22] R. Chandra. *The COOL Parallel Programming Language: Design, Implementation, and Performance*. PhD thesis, Stanford University, Apr. 1995.

[23] R. Chandra, A. Gupta, and J. L. Hennessy. COOL: A languange for parallel programming. In *Proceedings of the 2nd Workshop on Programming Languanges and Compilers for Parallel Computing*. IEEE CS, 1989.

[24] M. J. Compton. SCOOP: An investigation of concurrency in Eiffel. Master's thesis, Department of Computer Science, The Australian National University, 2000.

[25] The Creol language. `http:heim.ifi.uio.no/creol`, 2007.

[26] F. S. de Boer, M. M. Bonsangue, A. Grüner, and M. Steffen. Test driver generation from object-oriented interaction traces (extended abstract). In *Proceedings of the 19th Nordic Workshop on Programming Theory (NWPT'07)*, 2007.

[27] F. S. de Boer, D. Clarke, and E. B. 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.

[28] The E language. www.erights.org, 2007.

[29] C. Flanagan and M. Felleisen. The semantics of future. Technical Report TR94-238, Department of Computer Science, Rice University, 1994.

[30] C. Flanagan and M. Felleisen. Well-founded touch optimization of parallel scheme. Technical Report TR94-239, Department of Computer Science, Rice University, 1994.

[31] C. Flanagan and M. Felleisen. The semantics of future and an application. *Journal of Functional Programming*, 9(1):1–31, 1999.

[32] J. Foster, M. Fändrich, and A. Aiken. A theory of type qualifiers. In *ACM Conference on Programming Language Design and Implementation*, pages 192–203. ACM, May 1999.

[33] C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of POPL '96*, pages 372–385. ACM, Jan. 1996.

[34] C. Fournet and G. Gonthier. Th join calculus: A language for distributed mobile programming. In G. Barthe, P. Dybjer, L. Pinto, and J. Saraiva, editors, *APPSEM 2000*, volume 2395, 2002.

[35] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge University Press, 1989.

[36] I. A. M. Gul A. Agha, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1), Jan. 1997.

[37] P. Haller and M. Odersky. Actors that unify threads and events. In *Proceedings of COORDINATION '07*, volume 4467 of *Lecture Notes in Computer Science*, pages 171–190. Springer-Verlag, 2007. A longer version is available as EPFF Technical Report.

[38] R. H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, Oct. 1985.

[39] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In OOPSLA'99 [6], pages 132–146. In *SIGPLAN Notices*.

[40] Io: A small progamming language. `www.iolanguage.com`, 2007.

[41] G. S. Itzstein and D. Kearney. Join Java: An alternative concurrency semantics for java. Technical Report ACRC-01-001, University of South Australia, 2001.

[42] G. S. Itzstein and D. Kearney. Applications of join Java. In *Proceedings of the Seventh Asia-Pacific Computer Systems Architectures Conference (ACSAC 2002)*, 2002.

[43] A. Jeffrey and J. Rathke. A fully abstract may testing semantics for concurrent objects. In *Proceedings of LICS '02*. IEEE, Computer Society Press, July 2002.

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

[45] E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365(1–2):23–66, Nov. 2006.

[46] JSR 166: Concurrency utilities. `www.jcp.org/en/jsr/detail?id=166`, 2007.

[47] L. Kornstaedt. Alice in the land of Oz – an interoperability-based implementation of a functional language on top of a relational language. In *Proceedings of the First Workshop on Mulit-Language Infrastructure and Interoperability (BABEL'01)*, Electronic Notes in Theoretical Computer Science, Sept. 2001.

[48] H. Liebermann. A preview of ACT-1. AI-Memo AIM-625, Artificial Intelligence Laboratory, MIT, 1981.

[49] H. Liebermann. Concurrent object-oriented programming in ACT1. In Yonezawa and Tokoro [72].

[50] B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. *SIGPLAN Notices*, 23(7):260–267, 1988.

[51] D. A. Manolescu. Workflow enactment with continuation and future objects. In *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) '02 (Seattle, USA)*, pages 40–51. ACM, Nov. 2002. In *SIGPLAN Notices*.

[52] B. Meyer. Systematic concurrent object-oriented programming. *Communications of the ACM*, 36(9):56–80, 1993.

[53] L. Moreau. The semantics of scheme with future. In *International Conference on Functional Programming*, pages 146–156. ACM Press, 1996.

[54] J. Niehren, D. Sabel, M. Schmidt-Schauß, and J. Schwinghammer. Observational semantics for a concurrent lambda calculus with reference cells and futures. *Electronic Notes in Theoretical Computer Science*, 2007.

[55] J. Niehren, J. Schwinghammer, and G. Smolka. A concurrent lambda-calculus with futures. *Theoretical Computer Science*, 64(3):338–356, Nov. 2006.

[56] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala. A comprehensive step-by-step guide*. Artima Developer, 2008.

[57] A. Poetzsch-Heffter and J. Schäfer. A representation-independent behavioral semantics for object-oriented components. In M. M. Bonsangue and E. B. Johnsen, editors, *FMOODS '07*, volume 4468 of *Lecture Notes in Computer Science*. Springer-Verlag, June 2007.

[58] P. Pratikakis, J. Spacco, and M. W. Hicks. Transparent proxies for Java futures. In *Ninetheeth Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) '04*, pages 206–233. ACM, 2004. In *SIGPLAN Notices*.

[59] R. R. Raje, J. I. William, and M. Boyles. An asynchronous method incocation (ARMI) mechanism for Java. In *Proceedings of the ACM Workshop on Java for Science and Engineering Computation*, 1997.

[60] A. Rossberg, D. L. Botland, G. Tack, T. Brunklaus, and G. Smolka. Alice through the looking glass. In *Vol. 5 of Trends in Functional Programming*, chapter 6. Intellect Books, Bristol, 2006.

[61] J. Schwinghammer. A concurrent λ-calculus with promises and futures. Diplomarbeit, Universität des Saarlandes, Feb. 2002.

[62] M. Steffen. *Object-Connectivity and Observability for Class-Based, Object-Oriented Languages*. Habilitation thesis, Technische Faktultät der Christian-Albrechts-Universität zu Kiel, 2006. Submitted 4th. July, accepted 7. February 2007.

[63] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.

[64] T. Sysala and J. Janecek. Optimizing remote method invocation in Java. In *DEXA*, pages 29–35, June 2001.

[65] É. Tanter, J. Noyé, D. Caromel, and P. Cointe. Partial behavioral reflection: Spatial and temporal reflection of reification. In *Eighteenth Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) '03*, pages 27–46. ACM, 2003. In *SIGPLAN Notices*.

[66] K. Taura, S. Matsuoka, and A. Yoneazawa. ABCL/f: A future-based polymorphic typed concurrent object-oriented language — its design and implementation —. In *DIMACS workshop on Specification of Parallel Algorithms*, 1991.

[67] P. L. Wadler. Linear types can change the world. In C. B. Jones and M. Broy, editors, *Proceedings of PROCOMET '90*, 1990.

[68] A. Welc, S. Jagannathan, and A. Hosking. Safe futures in Java. In *Twentieth Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) '05*. ACM, 2005. In *SIGPLAN Notices*.

[69] Y. Yokote and M. Tokoro. Concurrent programming in concurrent SmallTalk. In Yonezawa and Tokoro [72], pages 129–158.

[70] A. Yonezawa. *ABCL: An Object-Oriented Concurrent System*. MIT Press, 1990.

[71] A. Yonezawa, J.-P. Briot, and E. Shibayama. Object-oriented concurrent programming in ABCL/1. In *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) '86 (Portland, Oregon)*, pages 258–268. ACM, 1986. In *SIGPLAN Notices* 21(11).

[72] A. Yonezawa and M. Tokoro, editors. *Object-oriented Concurrent Programming*. MIT Press, 1987.

[73] Y. Yonezawa, E. Shibayama, T. Takada, and Y. Honda. Modelling and programming in an object-oriented concurrent language ABCL/1. In Yonezawa and Tokoro [72], pages 55–89.

Notation index

2.1 Syntax		
Notation	Page	Meaning
n	5	name (in general, or thread / future / promise name in particular)
v	5	value
o	5	object name
c	5	class name
$()$	6	unit value
x	6	variable
C	6	component
$\mathbf{0}$	6	empty component
$\parallel$	6	parallel operator
$n\langle t\rangle$	6	thread with name n and code t
$c[(O)]$	6	class with name c and methods and fields defined in O
$o[c,F,L]$	6	instance o of class c with fields F and lock L
$\top$, resp., $\bot$	6	value of a taken, resp., free lock
ν	7	ν-operator for hiding
$\varsigma(s{:}T).\lambda(\vec{x}{:}\vec{T}).t$	7	method of the abstracted object s with formal parameters $\vec{x}$ and body t
$\bot_c$	8	undefined object reference
f	8	field name
$l = \varsigma(s{:}T).\lambda().v$ (or $l = v$) resp. $l = \varsigma(s{:}T).\lambda().\bot_c$ (or $l = \bot_c$)	8	field definition
$v.l()$ (or $v.l$)	8	field access
$v'.l := \varsigma(s{:}T).\lambda().v$ (or $v'.l := v$)	8	field update
$v_\bot$	8	either a value v or a symbol $\bot_c$
$\mathsf{new}\,c$	8	object creation
$\mathsf{promise}\,T$	8	promise creation
$\mathsf{bind}\,o.l(\vec{v}) : T \hookrightarrow n$	8	binding code to a promise
claim, get	8	get the result of a future
suspend, grab, and release	8	lock operations
2.2 Type system		
Notation	Page	Meaning
B	10	base types (such as integers, etc. Left unspecified)
Unit	10	type of the unit value $()$

$[T]^{+-}$	10	type of a reference to a future with write permisson, where the future will return a value of type T
$[T]^{+}$	10	type of a reference to a future with read-only permisson, where the future will return a value of type T
$[T]$	10	type of a reference to a future with write or read-only permisson, where the future will return a value of type T
$[l_1{:}U_1,\dots,l_k{:}U_k]$	10	interface type of unnamed objects
$[(l_1{:}U_1,\dots,l_k{:}U_k)]$	10	interface type for classes
$\vec{T}$	10	$T_1\times\dots\times T_k$
$\mathsf{Unit}\rightarrow T$	10	$T_1\times\dots\times T_k\rightarrow T$ when $k=0$
Δ	11	name context: typing assumptions about the environment
Θ	11	name context: typing commitments of the component
$\Delta\vdash C:\Theta$	11	typing judgment
$\oplus$	12	symmetric operation on well-formed name contexts
Γ	12	stack-organized variable context
$n\langle\bullet\rangle$	12	not yet fulfilled promise
$\leq$	13	subtyping relation on types
$\Gamma;\Delta\vdash e:T::\acute{\Gamma},\acute{\Delta}$	13	judgment: type system
$T.l$	14	pick the type in T associated with label l
$\Delta\backslash n:T,\Gamma\backslash n:T,\dots$	16	difference operator
2.3 Operational semantics		
Notation	Page	Meaning
$\rightsquigarrow$	16	internal transition, confluent step
$\xrightarrow{\tau}$	16	internal transition, other (non-confluent) step
$\equiv$	17	structural congruence
$\vdash C:ok$	20	write-error free component
$\Delta\vdash_m C:\Theta$	20	minimal typing judgment
$\Delta\vdash n=v$	24	Δ contains the corresponding value information
$\Delta\vdash n=\bot$	24	Δ does not contain the corresponding value information
$\Delta\vdash C:\Theta \quad \xrightarrow{a} \quad \acute{\Delta}\vdash\acute{C}:\acute{\Theta}$	24	external transition
Ξ	24	the tuple of name contexts Δ,Θ
γ_c	25	call label
γ_g	25	get label
$\gamma?$	25	incoming interaction label
$\gamma!$	25	outgoing interaction label

$\lfloor \gamma \rfloor$	25	core of the label γ
$fn(a)$	25	free names of label a
$bn(a)$	25	bound names of label a
$names(a)$	25	all names of label a
$fn_a(a)$	25	free names occurring in active position in a
$fn_p(a)$	25	the free names in passive position in a
$\vdash a$	25	a is well-formed
$\Xi \vdash o.l? : \vec{T} \to T$	25	an incoming call of the method labeled l in object o expects arguments of type $\vec{T}$ and results in a value of type T
$\acute{\Xi} \vdash a : \vec{T} \to$ _resp., $\acute{\Xi} \vdash a : \_ \to T$	26	well-typedness of an incoming core label a with expected type $\vec{T}$, resp., T, and relative to the name context $\acute{\Xi}$
$;\acute{\Xi} \vdash \vec{v} : \vec{T}$	26	$\acute{\Xi}_0$ abbreviate $;\acute{\Xi}$, then $\acute{\Xi}_i \vdash v_i : T_i$ and $\acute{\Xi}_{i+1} = \acute{\Xi}_i \setminus T_i$, for all $0 \leq i \leq n-1$
$\acute{\Xi} = \Xi + a$	28	context update
3 Interface behavior		
Notation	Page	Meaning
$\Xi \vdash s : trace$	31	judgment: legal trace
$\lfloor \Delta \rfloor$	32	binding replacement