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

12. October 2007

Erika Ábrahám\textsuperscript{1}, and Immo Grabe\textsuperscript{2}, and Andreas Grüner\textsuperscript{2}, and Martin Steffen\textsuperscript{3}

\textsuperscript{1} Jülich Research Centre, Germany
\textsuperscript{2} Christian-Albrechts-University Kiel, Germany
\textsuperscript{3} 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. The thread-based model of concurrency, as in Java and C\#\textsuperscript{4}, has 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 semantical 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\textsuperscript{15}, an object-oriented, concurrent language, featuring in particular asynchronous method calls and, since recently\textsuperscript{12}, first-class futures.

Futures and promises A future, very generally, represents a result yet to be computed. It acts as a proxy for the delayed result from some piece of code (e.g., a method or a function body). As the consumer of the result can proceed its own execution until it actually needs it, futures provide a natural, lightweight, and (in a functional setting) transparent mechanism to introduce parallelism into a language. Since their introduction
in **Multilisp** [14][6], futures have been used in various languages like Alice ML, E the ASP-calculus [8], Creol, and others. A **promise** is a generalization[5] 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*).

**Interface behavior** An open program interacts with its environment via message exchange. 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 $C \Rightarrow t$; instead we consider behaviors $C \parallel E \Rightarrow t \bar{C} \parallel E$ where $E$ is a *realizable* environment and $\bar{t}$ is complementary to $t$. To account for the abstract environment, the open semantics is given in an **assumption-commitment** way:

\[
\Delta \vdash C : \Theta \Rightarrow t \Rightarrow \Delta \vdash \bar{C} : \bar{\Theta},
\]

where $\Delta$ contains the **assumptions** about the environment, and dually $\Theta$ 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 is more restricted than the one obtained when ignoring the environments. When reasoning about 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 and allows furthermore optimization of components: only if two components show the same external behavior, one can replace one for the other without changing the interaction with any environment. 4) The formulation gives insight into the semantical nature of the language, here, the 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]. The operational semantics for components distinguishes unobservable component-internal steps from external steps which

---

[4] Sometimes the words futures and promises are used as synonyms in the literature.
represent observable component-environment interactions. We present the semantics in a way that facilitates comparison with Java’s multi-threading concurrency model.

*Linear type system for promises* The calculus extends the semantic basis of Creol [12] 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

$$[T]^-$$ and $$[T]^+$$

representing a reference to a promise that can still be written (and read), with result type $$T$$, resp. that has a *read*-permission. 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 and resembles in intention the one in [18] for a functional calculus with references. Our work is more general, in that it tackles the problem in an object-oriented setting, and in that we do not consider closed systems, but open components. Also this aspect of openness is not dealt with in [12]. Additionally, the type system presented here is simpler as in [18], 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
- *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 open systems. Section 3 describes the interface behavior. Section 4 concludes with related and future work. For more details and for the proofs see [2].

2 Calculus

This section presents the calculus, based on a version of the Creol-language with first-class futures [12] and extended with promises. It is a concurrent variant of an imperative, object-calculus in the style of the ones from [11].

2.1 Syntax

The abstract syntax in Table 1 distinguishes between *user* syntax and *run-time* syntax (the latter is underlined). The user syntax contains the phrases in which programs are written; the run-time syntax contains syntactic constituents additionally needed to express the executing program in the operational semantics.
\[ C ::= \emptyset | C \parallel C | \nu(n:T).C | n(O) | n(n,F,L) | n(t) \]  
Program

\[ O ::= F, M \]  
Object

\[ M ::= l = m, \ldots, l = m \]  
Method suite

\[ F ::= l = f, \ldots, f = f \]  
Fields

\[ m ::= \zeta(n:T).\lambda(x:T, \ldots, x:T).t \]  
Method

\[ f ::= \zeta(n:T).\lambda(l,v) | \zeta(n:T).\lambda(l) \downarrow \]  
Field

\[ t ::= v | \text{stop} | \text{let } x:T = e \in t \]  
Thread

\[ e ::= t | \text{if } v = v \text{ then } e \text{ else } e | \text{undef} (v,l) \text{ then } e \text{ else } e \]  
Expression

\[ \text{promise } T | \text{bind } n,l(T) : T \rightarrow n | \text{set } v \mapsto n | v,l() | v,l := \zeta(s:x).\lambda()v \]  
New n | Claim @(n,n) | Get @ n | Suspended(n) | Gather(n) | Release(n)

\[ v ::= x | n | () \]  
Values

\[ L ::= \bot \cup T \]  
Lock status

**Table 1. Abstract syntax**

Names \( n \) refer to classes, objects, threads, and to references to futures and promises. We use \( o \) and its syntactic variants for objects and \( c \) for classes, and \( n \) when being unspecific. The unit value is represented by \( () \). A component \( C \) is a collection of classes, objects, and threads, with \( \emptyset \) being the empty component. A class \( c(F) \) carries a name \( c \) and defines its methods and fields in \( O \). A method \( \zeta(s:c).\lambda(\bar{x}:T).l \) provides the method body \( l \) abstracted over the \( c \)-bound “self” parameter \( s \) and the formal parameters \( \bar{x} \). For uniformity, fields are represented as methods without parameters (except self), with a body being either a value or yet undefined. 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 \( T, n \), resp., \( \bot, \bot \), indicate that the lock is taken, resp., free. 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 \( \bot_c \), where class name \( c \) is the (return) type of the field.

Besides objects and classes, the dynamic configuration of a program contains incarnations of method bodies, written \( n(t) \), as active entities. The term \( t \) is basically a sequence of expressions, where the let-construct is used for sequencing and for local declarations. During execution, \( n(\text{let } x:T = t \in x) \) contains in \( t \) the currently running code of a method body and the result will be stored in the local variable \( x \). When evaluated, the thread is of the form \( n'(\text{set } v \mapsto n) \) and the value can be accessed via \( n \), the future reference, or future for short, where \( \text{set } v \mapsto n \) is an auxiliary expression.

We use \( f \) for instance variables or fields and \( l = \zeta(s:T).\lambda(l,v) \), resp. \( l = \zeta(s:T).\lambda().\bot_c \), for field variable definition. Field access is written as \( v,l() \) and field update as \( v'.l := \zeta(s:T).\lambda()v \). By convention, we abbreviate the latter constructs by \( l = v, l = \bot_c, v,l \), and \( v'.l := v \). We will also use \( \nu_{\bot} \) to denote either a value \( v \) or a symbol \( \bot_c \) for being undefined. Note that the syntax does not allow to set a field back to undefined. Direct

\[ t_1; t_2 \] (sequential composition) abbreviates \( \text{let } x:T = t_1 \text{ in } t_2 \), where \( x \) does not occur free in \( t_2 \).
access to fields across object boundaries is forbidden, and we do not allow method update. Instantiation of a new object from class \( c \) is denoted by \( \text{new } c \).

Expressions include promise \( T \) for creating a new promise, and bind \( o.@l(v) : T \leftrightarrow n \) for binding the method call \( o.@l(v) \) with return type \( T \) to promise \( n \). Asynchronous method calls, central to Creol’s concurrency model, are a derived concept. An asynchronous call, written \( o.@l(v) \) is syntactic sugar for creating a new promise and immediately binding \( o.@l(v) \) to it. Further, the expressions \( \text{claim, get, suspend, grab, and release} \) deal with communication and synchronization. The expression \( \text{claim@}(n, o) \) is the attempt to obtain the result of a method call from the future \( n \) while in possession of the lock of object \( o \). Executing \( \text{release}(o) \) 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 \( \text{grab}(o) \). Executing \( \text{suspend}(o) \) causes the activity to relinquish and re-grab the lock of object \( o \) (see the operational rules in Section 2.3.1 below). We assume by convention, that when appearing in methods of classes, the claim- and the suspend-command only refer to the self-parameter \( \text{self} \), i.e., they are written \( \text{claim@}(n, \text{self}) \) and \( \text{suspend}(\text{self}) \).

### 2.2 Type system

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

\[
T ::= B | \text{Unit} | [T]^+ \ | [T]^* \ | [l:U, \ldots, l:U] \ | [l:U, \ldots, l:U] \ | n
\]

\[
U ::= T \times \ldots \times T \rightarrow T
\]

Besides base types \( B \) (left unspecified), 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, \( [T]^* \) acts monotonely, and \( [T]^+ \) invariantly wrt. subtyping. When not interested in the access permission, we just write \( [T] \).

The name of a class serves as the type for the named instances of the class. We need as auxiliary type construction 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 Unit \( \rightarrow T \) for \( T_1 \times \ldots \times T_k \rightarrow T \) when \( k = 0 \).

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

\[
\Delta \vdash C : \Theta,
\]

where \( \Delta \) and \( \Theta \) are name contexts, i.e., finite mappings from names to types. In the judgment, \( \Delta \) plays the role of the typing assumptions about the environment, and \( \Theta \) of the commitments of the configuration, i.e., the names offered to the environment. Sometimes, the words required and provided interface are used to describe their dual roles. \( \Delta \) must contain at least all external names referenced by \( C \) and dually \( \Theta \) mentions
the names offered by \( C \), which constitute the static interface information. A pair \( \Delta \) and \( \Theta \) of assumption and commitment context with disjoint domains are called well-formed.

The (straightforward) rules for objects, classes, and the empty component are omitted in Table 2 (cf. \([2]\)). 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). The \( \nu \)-binder hides the bound object or the name of the future inside the component (cf. rule T-Nu). Named threads \( n(t) \) are treated by rule T-THREAD, where the type \([T]^+\) of the future reference \( n \) is matched against the result type \( T \) of thread \( t \). As obviously future \( n(t) \) is already fulfilled in \( n(\langle t \rangle) \), its type exports read-permission, only. For a named thread \( n(\langle t \rangle) \) in rule T-THREAD to be well-typed, the code \( t \) is checked using the assumptions \( \Delta \) of the conclusion but without using write-permissions mentioned in \( \Delta \), expressed by \( \lfloor \Delta \rfloor \). On types, the \( \lfloor \cdot \rfloor \) operation is defined as \( \lfloor [T]^+ \rfloor = [T]^+ \) and as identity on all other types. The definition is lifted pointwise to binding contexts. The last rule is a rule of subsumption, expressing a simple form of subtyping.

**Definition 1 (Subtyping).** The relation \( \leq \) on types is defined as identity for all types except for \([T]^+\) (mentioned above) and object interfaces, where we have:

\[
\lfloor \langle l_1:U_1,\ldots,l_k:U_k,l_{k+1}:U_{k+1},\ldots \rangle \rfloor \leq \lfloor \langle l_1:U_1,\ldots,l_k:U_k \rangle \rfloor.
\]

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

The same definition is applied, of course, also for name contexts \( \Theta \), 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. Especially the treatment of the write-permission 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

\[
\Gamma; \Delta \vdash e : T :: \hat{\Gamma}, \hat{\Delta},
\]

where the change from \( \Gamma \) and \( \Delta \) to \( \hat{\Gamma} \) and \( \hat{\Delta} \) reflects the potential consumption of write permissions when executing \( e \). The consumption is only potential, as the type system

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T-PAR</td>
<td>( A, \Theta_1 + C_1 : \Theta_1 \rightarrow A, \Theta_1 + C_2 : \Theta_2 )</td>
</tr>
<tr>
<td>T-Nu</td>
<td>( A + C : \Theta, n.T \rightarrow A + \nu(n;T).C : \Theta )</td>
</tr>
<tr>
<td>T-THREAD</td>
<td>( [\Delta, n([T]^+) + t : T] \rightarrow A \leq A' \rightarrow \Theta \leq \Theta' \rightarrow A + C : \Theta \rightarrow A' + C : \Theta' )</td>
</tr>
</tbody>
</table>

**Table 2. Typing (components)**
statically overapproximates the run-time behavior, of course. In Table 3, we concentrate on the treatment of futures and promises and omit the most of the typing for standard constructs (cf. again §). For brevity, we write $A; \Gamma \vdash e : T$ for $A; \Gamma \vdash e : T :: \Gamma, A$, when $\Gamma = \Gamma$ and $A = A$. Besides assumptions about the provided names of the environment kept in $A$, the typing is done relative to assumptions about occurring free variables.

They are kept separately in a variable context $\Gamma$, 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.

Table 3 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 $[T]^\ast$, 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 $[T]^\ast$. 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 $[T]^\ast$ and not just $[T]^\ast$. The auxiliary expression set $v \mapsto n$ is evaluated for its side-effect, only, and is of type Unit (cf. rule T-Compl). claim dereferences a future, i.e., it fetches a value of type $T$ from the reference of type $[T]^\ast$. Otherwise, the expression has no effect on $A$, as reading can be done arbitrarily many times. The treatment of get is analogous (cf. rules T-Claim and T-Get). For T-Suspend, 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 $\Delta$ and $\Gamma$ to $\Delta \setminus v : T$ resp. to $\Gamma \setminus v : T$. The difference-operator $\Delta \setminus n : [T]^\ast$ removes the
write-permission for \( n \) from the context \( \Delta \). In T-Bind, the premise \( \Delta; \Gamma, n; [T]^i \vdash v : \bar{T} \)
abbreviates the following: assume \( v = v_1, \ldots, v_n \) and \( \bar{T} = T_1 \ldots T_n \) and let \( \Xi_1 \) abbreviate \( \Gamma; \Delta, n; [T]^i \). Then \( \Xi \vdash v : \bar{T} \) means: \( \Xi_1 \vdash v_1 : T_1 \) and \( \Xi_{i+1} = \Xi_i \setminus T_i \), for all \( 1 \leq i \leq n \).
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.

2.3.1 Internal steps

The internal semantics describes the operational behavior of a closed system, not interacting with its environment. We show here only rules dealing with futures and lock handling. The treatment of the rest of the language is standard \([2]\). The corresponding reduction steps are shown in Table \( 4 \) distinguishing between confluent steps \( \leftrightarrow \) and other internal transitions \( \Rightarrow \), both invisible at the interface.

The expression promise \( T \) creates a fresh promise \( n \). Note that no new thread is yet allocated, as so far nothing more than the name is known. The rule Pcreate mentions the types \( T \) and \( T' \). A promise is fulfilled by the bind-command (cf. rule Bind), in that the new thread \( n \) is put together with the code \( t_1 \) to be executed and run in parallel with the rest as \( n' (\emptyset t x : T = t_1 \in \text{set } x \Rightarrow n) \) (where \( n' \) is hidden). Upon termination, the result is available via the claim- and the get-syntax (cf. the Claim-rules and rule Get), 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. Note the types of the involved let-bound variables: the future reference is typed by \([7]\), indicating that the value for \( x \) will not directly be available, but must be dereferenced first via claim. When it comes to claim a future, we add as auxiliary syntax set \( v \mapsto n \). The expression presents an evaluated thread, just in front of the step where the value \( v \) is about to be put into the thread named \( n \). The reasons for that additional syntax are largely technical, namely to achieve a clean separation of internal and externally visible behavior, in particular, to get a proper formulation of the subject reduction results. 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 written \( \equiv \), which captures the algebraic properties of parallel composition and the hiding operator. The formalization is standard and omitted here.

Next we show that the type system indeed assures what it is supposed to, most importantly that a promise is indeed fulfilled only once. A configuration \( C \) contains a write error if it is of the form \( C \equiv v(\Theta'), (C' \parallel n'(\emptyset t x : T = \text{bind } t_1 : T_1 \leftarrow n \in t_2) \parallel n(i)) \). Configurations without such write-errors are called write-error free, denoted \( \vdash C : \text{ok} \). In \([18]\), an analogous condition is called handle error.
First we show that a well-typed component does not contain a manifest write-error.

**Lemma 1.** If \( A \vdash C : \emptyset \), then \( \vdash C : ok \).

**Lemma 2 (Subject reduction).** If \( \Xi \vdash C \) and \( C \implies \hat{C} \), then \( \Xi \vdash \hat{C} \).

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

**Corollary 1.** If \( A \vdash C : \emptyset \) and \( C \implies \hat{C} \), then \( \vdash \hat{C} : ok \).

### 2.3.2 External semantics

The external semantics formalizes the environment interaction of an open component as labeled transitions between judgments of the form \( A \vdash C : \emptyset \), where \( A \) represent the **assumptions** about the environment of the component \( C \) and \( \emptyset \) the **commitments**. The assumptions require the existence of named entities in the environment (plus giving static typing information). The semantics maintains as invariant that the assumption and commitment contexts are disjoint concerning the names for objects, classes, and threads. In addition, the interface keeps information about whether the value of the future \( n \) is already known at the interface. If it is, we write \( n:T = v \) as binding of the context. We write furthermore \( A \vdash n = v \), if \( A \) contains

\[
\begin{align*}
o[c,F,L] & \parallel n(let x:T = o.l(\text{in } t)) \xrightarrow{FLLOOKUP} o[c,F',L] \parallel n(let x:T = F'.l(\text{in } t)) \\
o[c,F,L] & \parallel n(let x:T = o.l := v \text{ in } t) \xrightarrow{FUUPDATE} o[c,F,L] \parallel n(let x:T = o \text{ in } t) \\
c[F,M] & \parallel n(let x:c = \text{new } c \text{ in } t) \xrightarrow{NEWO} \\
c[F,M] & \parallel n(let x:c = o \text{ in } t) \xrightarrow{NEWO} \\
n'(let x:T' = \text{promise } T \text{ in } t) & \xrightarrow{PROM} v(n:T'),(n'(let x:T' = n \text{ in } t)) \\
c[F',M] & \parallel o[c,F,l] \parallel n(let x:T = \text{bind } o.l(\hat{v}) : T_2 \leftrightarrow n_2 \text{ in } t_1) \xrightarrow{BIND} \\
c[F',M] & \parallel o[c,F,l] \parallel n_1(let x:T = n_2 \text{ in } t_1) \\
& \quad \parallel v(n'(let x:T = \text{prom} \text{ ise } T' \text{ in } t) \xrightarrow{PROM}) \\
n'(set v \mapsto n_1) & \parallel n_2(let x:T = \text{claim@}(n_1, o) \text{ in } t) \xrightarrow{CLAIM} \\
n'(set v \mapsto n_1) & \parallel n_2(let x:T = v \text{ in } t) \\
& \quad \parallel n_2 \neq v \xrightarrow{CLAIM} \\
n'(set x \mapsto n_1) & \parallel n_2(let x:T = \text{claim@}(n_1, o) \text{ in } t) \xrightarrow{CLAIM} \\
n'(set x \mapsto n_1) & \parallel n_2(let x:T = v \text{ in } t) \\
& \quad \parallel n_2 \neq v \xrightarrow{CLAIM} \\
n(suspend(o);t) & \xrightarrow{Suspend} n(\text{release}(o);\text{grab}(o);t) \\
o[c,F,T] & \parallel n(\text{grab}(o);t) \xrightarrow{GRAB} o[c,F,T] \parallel n(t) \\
o[c,F,T] & \parallel n(\text{release}(o);t) \xrightarrow{RELEASE} o[c,F,T] \parallel n(t) \\
\end{align*}
\]

| Table 4. Internal steps |
\[ γ ::= n \langle \text{call o.l(\bar{\gamma})} \rangle | n \langle \text{get(v)} \rangle | ν(n: T).γ \quad \text{basic labels} \\
\]
\[ a ::= γ? | γ! \quad \text{receive and send labels} \]

\textbf{Table 5. Labels}

The corresponding value information and write \[ A \vdash n = \bot \], if that is not the case. This extension makes the value of a future (once claimed) available at the interface. With these judgments, the external transitions are of the form:
\[ A \vdash C : Θ \xrightarrow{a} A \vdash C : Θ'. \] \hfill (3)

\textbf{Notation 1} We abbreviate the tuple of name contexts \( \Lambda, \Theta \) as \( \Xi \). Furthermore we understand \( \Lambda', \Theta' \) as \( \Xi' \), etc.

The labels of the external transitions represent the corresponding interface interaction (cf. Table 5). A component exchanges information with the environment via call labels \( γ_c \) and get labels \( γ_g \). Interaction is either incoming or outgoing, indicated by \( ? \), resp., \( ! \). Scope extrusion of names across the interface is indicated by the \( ν \)-binder. Given a basic label \( γ = ν(\Xi)γ' \) where \( \Xi \) is a name context such that \( ν(\Xi) \) abbreviates a sequence of single \( n:T \) bindings and where \( γ' \) does not contain any binders, we call \( γ' \) the core of the label and refer to it by \( \langle γ \rangle \). We define core analogously for receive and send labels. The free names \( \text{fn}(a) \) and the bound names \( \text{bn}(a) \) of a label \( a \) are as usual, whereas \( \text{names}(a) \) 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 \( n \langle \text{call o.l(\bar{\gamma})} \rangle \). We write \( \text{fn}_p(a) \) for the free names occurring in active position, \( \text{fn}_n(a) \) for the free names in passive position, etc. All notations are used analogously for basic labels \( γ \).

An important but standard part of the external semantics is to check the static typing assumptions, e.g., 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. As being standard, we do not show the rules here (cf. [2]).

Besides checking whether the assumptions are met before a transition, the contexts are updated by a transition step, i.e., extended by the new names, whose scope extrudes.

This gives rise to the following definition.

\textbf{Definition 2 (Context update).} Let \( Ξ \) be a name context and \( a = ν(\Xi), [a] \) an incoming label. We define the (intermediate) contexts \( Θ'' = Θ \) and \( A'' = A, Ξ' \).

Let furthermore \( Ξ'' \) be the set of bindings defined as follows. In case of a call label, i.e., \([a] = n\langle \text{call o.l(\bar{\gamma})} \rangle\), let the vector of types \( \bar{T} \) be the argument types of the method \( l \). Then \( Ξ'' \) consists of bindings of the form \( ν_i:T_i'' \) for values \( ν_i \) from \( \bar{v} \) such that \( T_i = [T_i']'' \). In case of a get label, i.e., \([a] = n\langle \text{get(v)} \rangle\), the context \( Ξ'' \) is \( ν:\bar{T}'' \) if \( A'' \vdash n : [[T]'']' \), and empty otherwise.

With \( Ξ'' \) given this way, the definitions of the post-contexts \( \hat{A} \) and \( \hat{Θ} \) distinguish between calls and get-interaction: If \( a \) is a call label and \( n \in \text{names}_o(a) \), we define
\[ \hat{A} = A'' \setminus Ξ'' \setminus n::[T]' \
\text{ and } \quad \hat{Θ} = Θ'', Ξ'', n::[T]'. \] \hfill (4)
If \( a \) is a get label, \( a = \nu(\Xi') \), \( n(\nu(\nu)) \), and \( n \in \text{names}_a(a) \), \( \hat{A} \) and \( \hat{\Theta} \) are given by:

\[
\hat{A} = A'' \setminus \Sigma'' \text{, } n:\{T\}^+ = \nu \quad \text{and} \quad \hat{\Theta} = \Theta'' \text{, } \Sigma''.
\]

(5)

For outgoing communication, the definition is applied dually.

Now to the interface behavior. Corresponding to the labels from Table 5, there are a number of rules for external communication: either incoming or outgoing calls, resp., get-labels. All rules have some premises in common. In all cases, the context \( \Xi \) before the interaction is updated to \( \Xi = \Xi + a \) using Definition 2, where \( a \) is the interaction label. The rules for incoming communication differ from the ones for outgoing communication in that well-typedness and well-formedness of the label is checked by the premises \( \Xi \vdash [a] : \vec{T} \rightarrow \_ \) resp. \( \Xi \vdash [a] : \_ \rightarrow \vec{T} \). 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, corresponding to rules Claim\(_1\) and Claim\(_2\), the interface does not contain the value of the future yet, stipulated by the premise \( A \vdash n' = \bot \). Remember that \( A \vdash n \) requires that the thread \( n \) is part of the environment. 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\(_2\)). If the future value is already known at the interface (cf. rule Claim\(_3\) and especially premise \( A \vdash n' = \nu \)), executing Claim is always successful and the value \( \nu \) is (re-)transmitted. Get works analogously to Claim, except that Get insists of obtaining the value, i.e., the alternative of relinquishing the lock and trying again as in rule Claim\(_2\), is not available for Get. I.e., the rules dealing with get executed by the component correspond to Claim\(_1\) and Claim\(_3\) and omitted here (cf. [2]). The last two rules deal with the situation that the environment fetches the value.

Finally, we characterize the initial situation. 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. To characterize when a given trace is legal, the behavior of the component side, i.e., the outgoing communication, must adhere to the dual discipline we imposed on the environment for the open semantics. This means, we analogously abstract away from the program code, rendering the situation symmetric.
3.1 Legal traces system

The rules of Table 6 specify legality of traces. We use the same conventions and notations as for the operational semantics (cf. Notation 1). The judgments are of the form

$$\Xi \vdash s : trace$$

(6)

We write $$\Xi \vdash t : trace$$, if there exists a derivation according to the rules of Table 6 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-CallsI for incoming calls works completely analogously to the CallI-rule in the semantics: the second premise updates the context $$\Xi$$ with the information contained in $$a$$, premise $$\Xi' \vdash n$$ of L-CallsI assures that the identity $$n$$ of the future, carrying out the call, is fresh and the two premises $$\Xi \vdash a.L ? : T \rightarrow \_$$ and $$\Xi \vdash [a] : T \rightarrow \_$$ together assure that the transmitted values are well-typed; the latter two checks correspond to the analogous premises for the external semantics in rule CallI. The L-GetI-rules for claiming a value work similarly.

Finally we formalize that the behavioral description of Table 6 actually does what it claims to do, to characterize the possible interface behavior of well-typed components.

**Lemma 3** (Subject reduction). If $$\Xi_0 \vdash C \Rightarrow^* \hat{\Xi} \vdash \hat{C}$$, then $$\hat{\Xi} \vdash \hat{C}$$.

**Lemma 4** (Soundness of abstractions). If $$\Xi_0 \vdash C$$ and $$\Xi_0 \vdash C \Rightarrow^*$$, then $$\Xi_0 \vdash t : trace$$. 

---

Table 6. External steps
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. Concentrating 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 notion of futures was introduced by Baker and Hewitt [6]. The principle has also been called wait-by-necessity [7]. 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 [14].

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 [13] present an operational semantics (based on evaluation contexts) for a $\lambda$-calculus with futures. The formalization is used for an analysis and optimization technique to eliminate superfluous dereferencing (“touches”) of future variables. Promises is a mechanism quite similar to futures and actually the two notions are sometimes used synonymously. They have been proposed in [16]. A language featuring both futures and promises as separate concepts, is Alice ML [5].

[18] presents a concurrent call-by-value $\lambda$-calculus with reference cells (i.e., a non-purely functional calculus with an imperative part and a heap) and with futures ($\lambda_{fut}$), which serves as the core of Alice ML. Certain aspects of that work are quite close to the
material presented here. We were inspired by using a type system to avoid fulfilling a promise twice (in [18] called handle error). There are some notable differences, as well. The calculus incorporates futures and promises into a $\lambda$-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 is more high-level. In contrast, $\lambda_{\text{fut}}$ 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. Recently, an observational semantics for the (untyped) $\lambda_{\text{fut}}$-calculus has been developed in [17].

Futures has also been investigated in the object-oriented paradigm. In Java 5, futures have been introduced as part of the java.util.concurrent package. As Java does not support futures as 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 [22] introduces safe futures for Java. The safe concept is intended to make futures transparent and in this sense goes back to the origins of the concept: introducing parallelism via futures does not change the program’s meaning. Pratikakis et. al. [20] present a constraint-based static analysis for (transparent) futures and proxies in Java. Similarly, Caromel et. al. [9] tackle the problem to provide confluent, i.e., effectively deterministic system behavior for a concurrent object calculus with futures (asynchronous sequential processes, ASP, an extension of Abadi and Cardelli’s imperative, untyped object calculus imp$\$ [1]) and in the presence of imperative features. The ASP model is implemented in the ProActive Java-library [10].

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 C#. The work [4] deals with thread classes and [3] with re-entrant monitors. In [21] the proofs of full abstraction for the sequential and multi-threaded cases can be found. Poetzsch-Heffter and Schäfer [19] 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 [21]. 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 [11]. 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.

References

5. Alice project home page. [www.ps-uni-sb.de/alice](http://www.ps-uni-sb.de/alice), 2006.