ABS: A Core Language for Abstract Behavioral Specification *

Einar Broch Johnsen[1], Reiner Hähnle[2], Jan Schäfer[3], Rudolf Schlatte[1], and Martin Steffen[1]

[1] Department of Informatics, University of Oslo, Norway
{einarj,rudi,msteffen}@ifi.uio.no
[2] Chalmers University of Technology, Sweden
reiner@chalmers.se
[3] Department of Computer Science, University of Kaiserslautern
jschaefer@cs.uni-kl.de

Abstract. This paper presents ABS, an *abstract behavioral specification* language for executable designs of distributed object-oriented systems. The language combines advanced concurrency and synchronization mechanisms for concurrent object groups with a functional language for modeling data. ABS uses asynchronous method calls, interfaces to enforce encapsulation, and cooperative scheduling of method activations inside concurrent objects. This feature combination results in a concurrent object-oriented model which is inherently compositional. This paper discusses central design issues for ABS and formalizes the type system and semantics of Core ABS, a calculus with the main features of ABS. For Core ABS, we prove a subject reduction property which shows that well-typedness is preserved by execution; in particular that method not understood errors do not occur at runtime for well typed ABS models. Finally, we briefly discuss the tool support developed for ABS.

1 Introduction

This paper presents ABS, an *abstract behavioral specification* language for distributed object-oriented systems. Abstract behavioral specification languages can be situated between design-oriented and implementation-oriented specification languages. ABS targets the specification of *executable designs* for object-oriented systems: it allows a high-level specification of a system, including its concurrency and synchronization mechanisms. Thus ABS models capture the concurrent control flow of object-oriented systems, yet ABS abstracts from many implementation details which may be undesirable at the modeling level, such as the concrete representation of internal data structures, the scheduling of method activations, and the properties of the communication environment.

The target domain of ABS is distributed systems. In the distributed setting, the implementation details of other objects in the system are not necessarily

* Partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models (http://www.hats-project.eu).

known. Instead, ABS uses interfaces as types for objects, abstracting in the type system from the classes implementing the functionality of these objects. The strict separation of types and implementations makes concurrent ABS models *compositional*. The concurrency model of ABS is similar to that of JCoBox [32], which generalizes the concurrency model of Creol [22] from single concurrent objects to concurrent object groups (so-calls cogs). The language supports asynchronous method calls, which trigger activities in other objects without transferring control from the caller, using first-class futures [14]. Thus, an object may have many method activations competing to be executed. A distinguishing feature of this concurrency model is the use of *cooperative scheduling* of method activations to explicitly control the internal interleaving of activities inside cogs. Thus, a clear notion of quiescent state may be formulated, namely when the active process of each object in the cog is *idle*. This allows an approach to system verification in which local reasoning is based on the maintenance of monitor invariants which must hold in quiescent states. Because of the cooperative scheduling and the interface encapsulation mechanism, local reasoning about the concurrent object system can be done by means of standard verification systems for sequential object-oriented programs. This approach is explored in [3, 14, 17].

This paper discusses the design decisions behind ABS and defines Core ABS, a calculus formalizing the main features of ABS. The contributions of this paper may be summarized as follows:

- We define the *functional level* of ABS, which is used to abstract computations on internal data in concurrent objects. ABS supports user-defined parametric data types and functions with pattern matching. We define a syntax, type system, and reduction system for functional expressions in Core ABS.
- We define the *concurrent object level* of ABS, which is used to capture concurrent control flow and communication in ABS models. This part of ABS integrates functional expressions and concurrent object groups with cooperative scheduling. We define a syntax, type system, and an SOS style operational semantics for the concurrent object level of Core ABS.
- We show how *type preservation* is guaranteed at runtime for well typed models in Core ABS, with a particular on focus on the creation of concurrent object groups, objects, and first-class futures.

2 Abstract Behavioral Specification

Specification languages may be roughly categorized into three categories, which serve partly complementary and partly overlapping purposes:

- *Design-oriented languages* focus on structural aspects of systems, such as the relationship between features or classes, and the flow of messages between objects. Examples of design-oriented languages are UML/OCL [34], FDL [33], and architectural description languages [12, 27].
- *Foundational languages* focus on foundational aspects of, e.g., concurrency and interaction, by identifying a small set of primitives and their formal

semantics. Examples of foundational languages are process algebras [29], automata models [25], and object calculi [1, 21].
– *Implementation-oriented languages* focus on behavioral properties of implemented systems. Examples of implementation-oriented specification languages are JML [8] and Spec# [7].

Design-oriented languages often provide elegant graphical means of displaying a system's structure, but typically lack flexible constructs for expressing concurrency and synchronization aspects of a system. Foundational languages address this concern, but their focussed scope excludes language features which makes it cumbersome to develop models of real systems without complicating encodings; the resulting models will typically not reflect the structure of an object-oriented target program. Even the abstractions of object calculi make it difficult to express real systems; for example, Featherweight Java does not provide fields in objects. In contrast, implementation-oriented languages are restricted to the particular concurrency and synchronization mechanisms of their target language, and typically enforce particular solutions which may be undesirable at the design stage.

ABS takes an approach in between these three categories of specification languages. It is designed to be close to the way programmers think, by maintaining a Java-like syntax and a control flow close to an actual implementation. In fact, ABS models may be automatically compiled into, e.g., Java (see Sect. 7). On the other hand, the language has a formally defined semantics, in the style of foundational languages, and allows the modeler to abstract from undesirable implementation details by means of user-defined algebraic data types and functions. Consequently, imperative structures may be used to study particular aspects of a system, while other aspects may be abstracted to ADTs. In addition, the concurrency model of ABS abstracts from particular assumptions about the communication environment, such as ordering schemes for message transfer and scheduling policies for the selection of method activations inside the objects.

3 The Design of ABS

ABS targets distributed object-oriented systems which communicate asynchronously. The concurrency model of ABS ressembles that of JCoBox [32], which generalizes the concurrency model of Creol [14,22] from single concurrent objects to concurrent object groups (*cogs*). Cogs can be seen as object-based runtime components, with their own object heap, which solely communicate via asynchronous method calls. A cog's behavior is based on cooperative multi-tasking of method activations. Cooperative multi-tasking guarantees data-race freedom inside a cog and enables the safe combination of active and reactive behavior.

Complementing the object language, ABS supports user-defined data types with (non-higher-order) functions and pattern matching. This functional level of ABS is largely orthogonal to the concurrent object level and is intended to model data manipulation. As such data is immutable, it can safely be exchanged between cogs. Using functional data types to realize most internal data structures of cogs can significantly simplify the specification and verification of models.

ABS contains non-deterministic constructs; in particular, the outcome of executing concurrency primitives is non-deterministic. While underspecification is used for data abstraction, non-deterministic execution semantics is the prerequisite for abstracting behavior. As ABS is a modeling language, it makes no a priori assumptions about, for example, concrete scheduling mechanisms. Underspecification and non-determinism do not preclude executability: the outcome of a non-deterministic transition step is a set of possible successor states which can be systematically inspected in simulation, analysis, and visualization tools.

In this remainder of this section, we briefly describe how to represent and work with data, and then discuss the concurrent object level of ABS.

3.1 Data Types, Functions, and Pattern Matching

ABS does *not* have primitive types for working with basic values. Instead algebraic data types may be defined by the user. A library of predefined data types and operators is provided, including `Unit`, `Bool`, `Int`, and `String`. Data types in ABS can be *polymorphic*; i.e., their definition may have type parameters.

Example 1. The following code shows the polymorphic data type `Set<A>` (which is part of the ABS Standard Library), as well as a function `contains`, which checks whether an element `e` is a member of a given set `s`.

```
data Set<A> = EmptySet | Insert(A, Set<A>);
def Bool contains<A>(Set<A> s, A e) =
   case s { EmptySet => False ;
          Insert(e, _) => True;
          Insert(_, xs) => contains(xs, e); };
```

3.2 Interfaces in ABS

ABS is a class-based language, which uses interfaces for typing. ABS has no class inheritance, but multiple inheritance is allowed at the interface level. A class may implement several interfaces, provided that it supports all methods offered by these interfaces. Reasoning control is ensured at the level of interfaces: *an object supporting an interface I may be replaced by another object supporting I or a subtype of I* in a context where I is expected, although the class of the two objects may differ. Due to the typing of object variables by interfaces, it is not possible to access the fields of another object directly, only method calls to the object are possible. This way, the object controls its own state; another object can only manipulate the state indirectly via the methods made available through an interface. In fact, interfaces are the only encapsulation mechanism of ABS objects and no access modifiers are provided. However, since the class may support several interfaces, different methods may be offered to the environment through different interfaces; for example, a *super-user* interface may export methods not seen through the normal *user* interface.

3.3 The Concurrency Model of ABS

Intuitively, cogs have dedicated processors and live in a distributed environment with asynchronous and unordered communication. A set of objects is located in a cog. All communication is between named objects, typed by interfaces, by means of asynchronous method calls. Calls are asynchronous as the caller may decide at runtime when to synchronize with the reply from a call. Asynchronous method calls may be seen as triggers of concurrent activity, spawning new method activations (so-called *processes*) in the called object. Active behavior, triggered by an optional method *run*, is interleaved with passive behavior, triggered by method calls. Thus, an object has a set of processes to be executed, which stem from method activations. Among these, at most one process in the objects of a cog is *active* and the other processes are *suspended* in a process pool. Process scheduling is non-deterministic, but controlled by *processor release points* in a cooperative way. Thus, the amount of concurrency in an ABS model is reflected in the number of cogs introduced in the model. A concurrent object model (as in Creol) corresponds to an ABS model in which each object has its own cog.

Example 2. Consider a book shop where clients order `books` for delivery to a `country`. Clients connect to the shop by the `getSession` method of an `Agent` object, which hands out `Session` objects from a dynamically growing pool. Clients call the `order` method of their `Session`, which calls the `getInfo` and `confirmOrder` methods of a `Database` shared between the different sessions. `Session` objects return to the agent's pool upon order completion. (The full model is given in [4].)

```
interface Agent {
  Session getSession();
  Unit free(Session session);
}
interface Session {
  OrderResult order(
    List<Bname> books, Cname country);
}
interface Database {
 DatabaseInfo getInfo(
    List<Bname> books, Cname country);
 Bool confirmOrder(List<Bname> books);
}
```

```
class DatabaseImp(Map<Bname,Binfo> bDB,
 Map<Cname,Cinfo> cDB) implements Database {
 DatabaseInfo getInfo(
   List<Bname> books, Cname country) {
   Map<Bname,Binfo> bOrder =
     getBooks(bDB, books);
   Pair<Cname,Cinfo>cDestiny =
     getCountry(cDB, country);
   return Info(bOrder, cDestiny);
 }
 ...
}
```

The `DatabaseImp` class stores and handles the information about the shop's available books (in the `bDB` map) and about the delivery countries (in the `cDB` map). This class has a method `getInfo`; given an order with a list of `books` and a destination `country`, this method extracts information about book availability from `bDB` and shipping information from `cDB` by means of function calls `getBooks(bDB, books)` and `getCountry(cDB, country)` The result from a call to the method has type `DatabaseInfo`, with a constructor of the form: `Info(bOrder, cDestiny)`.

Syntactic categories.	*Definitions.*
T in Ground Type	$T ::= B \mid I \mid D \mid D\langle\overline{T}\rangle$
A in Type	$A ::= N \mid T \mid N\langle\overline{A}\rangle$
x in Variable	$Dd ::= \texttt{data}\ D[\langle\overline{A}\rangle] = Cons[\overline{\mid Cons}];$
e in Expression	$Cons ::= Co[(\overline{A})]$
b in Bool Expression	$F ::= \texttt{def}\ A\ fn[\langle\overline{A}\rangle](\overline{A}\,\overline{x}) = e;$
t in Ground Term	$e ::= b \mid x \mid t \mid \texttt{this} \mid Co[(\overline{e})] \mid fn(\overline{e}) \mid \texttt{case}\ e\ \{\overline{br}\}$
br in Branch	$t ::= Co[(\overline{t})] \mid \texttt{null}$
p in Pattern	$br ::= p \Rightarrow e;$
	$p ::= \_ \mid x \mid t \mid Co[(\overline{p})]$

Fig. 1. Core ABS syntax for the functional level. Terms $\overline{e}$ and $\overline{x}$ denote possibly empty lists over corresponding syntactic categories, and square brackets [ ] optional elements.

4 A Formal ABS Calculus

This section presents Core ABS, a formal calculus which simplifies ABS by excluding, e.g., the module system, type synonyms, the predefined data types (except Bool), and annotations. However, it captures the central features of ABS. (A complete formalization of ABS exists in the rewriting logic of Maude [11].)

4.1 The Syntax of Core ABS

An ABS *model* defines interfaces, classes, datatypes, and functions, and a *main block* to configure the initial state. Objects are dynamically created instances of classes; their attributes are initialized to type-correct default values (e.g., **null** for object references), but may be redefined in an optional method *init*.

A Functional Language for User-Defined Parametric Data Types and Functions. The functional level of Core ABS defines data types and functions, as shown in Fig. 1. The ground types consist of basic types such as Bool and Int, as well as names D for data types and I for interfaces. In general, a type A may also contain type variables (i.e., uninterpreted type names [30]). In data type declarations Dd, a data type D has at least one constructor $Cons$, which has a name Co and a list of types $\overline{A}$ for its arguments. Function declarations F consist of a return type A, a function name fn, a list of variable declarations $\overline{x}$ of types $\overline{A}$, and an expression e. *Expressions* e include Boolean expressions b, variables x, (ground) terms t, the self-identifier **this**, constructor expressions $Co(\overline{e})$, function expressions $fn(\overline{e})$, and case expressions **case** e $\{\overline{br}\}$. Ground terms t are constructors applied to ground terms $Co(\overline{t})$, and **null**. Case expressions have a list of branches $p \Rightarrow e$, where p is a pattern. The branches are evaluated in the listed order. Patterns include wild cards _, variables x, terms t, and constructor patterns $Co(\overline{p})$.

The Concurrent Object Level of Core ABS is given in Fig. 2. An interface *IF* has a name I and method signatures Sg. A class CL has a name C, interfaces $\overline{I}$

Syntactic categories.	*Definitions.*
C, I, m in Names	$IF ::= \texttt{interface}\ I\,\{\,\overline{Sg}\,\}$
g in Guard	$CL ::= \texttt{class}\ C\,[(\overline{T}\ \overline{x})]\,[\texttt{implements}\ \overline{I}]\,\{\,\overline{T}\ \overline{x};\ \overline{M}\}$
s in Statement	$Sg ::= T\ m\ (\overline{T}\ \overline{x})$
	$M ::= Sg\,\{\,\overline{T}\ \overline{x};\ s\,\}$
	$g ::= b \mid x? \mid g \wedge g$
	$s ::= s; s \mid x = rhs \mid \texttt{suspend} \mid \texttt{await}\ g \mid \texttt{return}\ e$
	$\mid \texttt{if}\ b\ \texttt{then}\,\{\,s\,\}\,[\texttt{else}\,\{\,s\,\}] \mid \texttt{while}\ b\,\{\,s\,\} \mid \texttt{skip}$
	$rhs ::= e \mid \texttt{new}\ [\texttt{cog}]\ C[(\overline{e})] \mid e!m(\overline{e}) \mid e.m(\overline{e}) \mid x.\texttt{get}$

Fig. 2. Core ABS syntax for the concurrent object level.

(specifying types for its instances), class parameters and state variables x of type T, and methods M (The *attributes* of the class are both its parameters and state variables). A method signature Sg declares the return type T of a method with name m and formal parameters $\overline{x}$ of types $\overline{T}$. M defines a method with signature Sg, local variable declarations $\overline{x}$ of types $\overline{T}$, and a statement s. Statements may access attributes of the current class, locally defined variables, and the method's formal parameters. A program's main block is a method body $\{\overline{T}\ \overline{x}; s\}$. There are no type variables at the concurrent object level of ABS.

Right hand side expressions *rhs* include object creation within the same cog **new** $C(\overline{e})$ and in a fresh cog **new cog** $C(\overline{e})$, method calls, and (pure) expressions e. Statements are standard for assignment $x = rhs$, sequential composition $s_1; s_2$, and **skip**, **if**, **while**, and **return** constructs. **suspend** unconditionally releases the processor, suspending the active process. In **await** g, the guard g controls processor release and consists of Boolean conditions b and return tests $x?$ (see below). If g evaluates to false, the processor is released and the process *suspended*. When the processor is idle, any enabled process from the object's pool of suspended processes may be scheduled. Explicit signaling is redundant.

Communication in ABS is based on asynchronous method calls, denoted $o!m(\overline{e})$, and synchronous method calls, denoted $o.m(\overline{e})$. Any method may be called either synchronously or asynchronously. After asynchronously calling $x = o!m(\overline{e})$, the caller may proceed with its execution without blocking on the call. Here x is a future variable, o is an object (an expression typed by an interface), and $\overline{e}$ are expressions. A future variable x refers to a return value which has yet to be computed. There are two operations on future variables, which explicitly control external synchronization in ABS. First, a return test $x?$ evaluates to false unless the reply to the call can be retrieved. (Return tests are used in guards.) Second, the return value is retrieved by the expression x.**get**, which blocks all execution in the object until the return value is available. Internally in a process, the reserved variable *destiny* refers to the future associated with the process. The statement sequence $x = o!m(\overline{e});\ v = x.\texttt{get}$ encodes a blocking, *synchronous call* between objects in different cogs and is abbreviated $v = o.m(\overline{e})$. In contrast, synchronous calls $v = o.m(\overline{e})$ inside a cog have the reentrant semantics known from, e.g., Java threads. The statement sequence $x = o!m(\overline{e});$ **await** $x?;\ v = x.\texttt{get}$ encodes a non-blocking, *preemptable call*, abbreviated **await** $v = o.m(\overline{e})$.

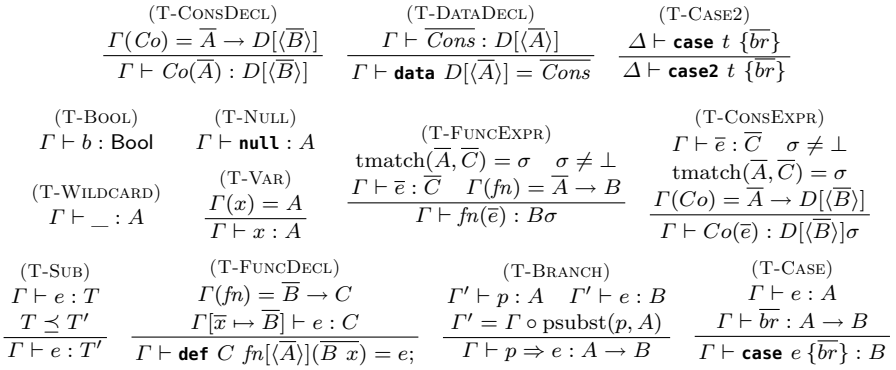

Fig. 3. The type system for the functional level of ABS.

4.2 Type System

A *mapping* binds names to values. Let Γ be a mapping, $[N \mapsto V]$ a binding from name N to value V, and denote lookup by $\Gamma(x)$. Then $\Gamma[N \mapsto V]$ denotes the mapping such that $\Gamma[N \mapsto V](N) = V$ $\Gamma[N \mapsto V](x) = \Gamma(x)$ if $x \neq N$. Denote the empty mapping by ε, lists of bindings by $[\overline{N} \mapsto \overline{V}]$ and $[\overline{N} \mapsto \overline{V}, \overline{N}' \mapsto \overline{V}']$, and mapping composition by $\Gamma \circ \Gamma'$. We say that Γ' *extends* Γ, denoted $\Gamma \subseteq \Gamma'$, if $\mathrm{dom}(\Gamma) \subseteq \mathrm{dom}(\Gamma')$ and $\Gamma(x) = \Gamma'(x)$ if $x \in \mathrm{dom}(\Gamma)$.

A *typing context* Γ is a mapping from names to typings which assigns types A to variables, type constants T to constants, and type signatures $\overline{A} \to B$ to function symbols. For simplicity, overloading is not considered. A name can only have one typing, and interface and class names are assumed to be distinct. We omit the typing of basic types such as Bool and Int, and assume that expressions of the basic types are type checked directly as in the rule T-Bool.

The functional level of the ABS type system is shown in Figure 3 and explained below. We assume a typing context Γ which maps names to their declared types; i.e., the initial typing context gives types to variables and to (user-defined) constructors and functions. The expression **null** can have any type by rule T-Null. A variable is well typed if declared in Γ by rule T-Var. In rule T-ConsDecl, constructor declarations are treated like variables. (Note that the constructor may be parametric; e.g., for $\mathsf{List}\langle A \rangle$, the list constructor *Cons* should have the type $A, \mathsf{List}\langle A \rangle \to \mathsf{List}\langle A \rangle$.) In rule T-ConsExpr, a constructor expression is well typed if its actual and formal parameter types are the same when matching the type variables of the formal parameter type to the actual parameter types by the auxiliary function *tmatch*. If there is no match, $tmatch(\overline{A}, \overline{C})$ returns $\bot$. (In this case, if x is an Int and y is a $\mathsf{List}\langle \mathsf{Int} \rangle$, then $Cons(x, y)$ should get type $\mathsf{List}\langle \mathsf{Int} \rangle$, which happens since *tmatch* binds A to Int.) Function definition and application are handled in the same way by rules T-FunDecl and T-FuncExpr. Additionally the function body is type-checked in Γ extended with the typing of formal parameters in T-FunDecl, which may again be type variables.

$$\frac{\Gamma \vdash x : \mathtt{fut}\langle T\rangle}{\Gamma \vdash x? : \mathsf{Bool}}\ (\text{T-Poll}) \qquad \frac{\Gamma \vdash x : \mathtt{fut}\langle T\rangle}{\Gamma \vdash x.\mathtt{get} : T}\ (\text{T-Get}) \qquad \frac{}{\Gamma \vdash \mathtt{skip}}\ (\text{T-Skip}) \qquad \frac{\Gamma \vdash g : \mathsf{Bool}}{\Gamma \vdash \mathtt{await}\ g}\ (\text{T-Await}) \qquad \frac{}{\Gamma \vdash \mathtt{suspend}}\ (\text{T-Suspend})$$

$$\frac{\Gamma \vdash s \quad \Gamma \vdash s'}{\Gamma \vdash s; s'}\ (\text{T-Composition}) \qquad \frac{\Gamma \vdash e : \Gamma(v)}{\Gamma \vdash v := e}\ (\text{T-Assign}) \qquad \frac{\Gamma \vdash g_1 : \mathsf{Bool} \quad \Gamma \vdash g_2 : \mathsf{Bool}}{\Gamma \vdash g_1 \wedge g_2 : \mathsf{Bool}}\ (\text{T-And}) \qquad \frac{\Gamma \vdash \overline{e} : \text{ptypes}(C) \quad T \in \text{interfaces}(C)}{\Gamma \vdash \mathtt{new}\ [\mathtt{cog}]\ C(\overline{e}) : T}\ (\text{T-New})$$

$$\frac{\Gamma \vdash e.m(\overline{e}) : T}{\Gamma \vdash e!m(\overline{e}) : \mathtt{fut}\langle T\rangle}\ (\text{T-AsyncCall}) \qquad \frac{\Gamma \vdash b : \mathsf{Bool} \quad \Gamma \vdash s_1 \quad \Gamma \vdash s_2}{\Gamma \vdash \mathtt{if}\ b\ \mathtt{then}\ \{\ s_1\ \}\ \mathtt{else}\ \{\ s_2\ \}\ \mathtt{fi}}\ (\text{T-Conditional}) \qquad \frac{\Gamma \vdash b : \mathsf{Bool} \quad \Gamma \vdash s}{\Gamma \vdash \mathtt{while}\ b\ \{\ s\ \}}\ (\text{T-While})$$

$$\frac{\Gamma \vdash e : T \quad \Gamma(\text{destiny}) = \mathtt{fut}\langle T\rangle}{\Gamma \vdash \mathtt{return}\ e}\ (\text{T-Return}) \qquad \frac{\Gamma \vdash e : N \quad \Gamma \vdash \overline{e} : \overline{T} \quad \text{match}(m, \overline{T} \to T, N)}{\Gamma \vdash e.m(\overline{e}) : T}\ (\text{T-SyncCall}) \qquad \frac{\Gamma' = \Gamma[\overline{x} \mapsto \overline{T}, \overline{x'} \mapsto \overline{T'}] \quad \Gamma'[\text{destiny} \mapsto \mathtt{fut}\langle T'\rangle] \vdash s}{\Gamma \vdash T'\ m\ (\overline{T\ x})\{\overline{T'\ x'}; s\}}\ (\text{T-Method})$$

$$\frac{\forall I \in \overline{I} \cdot \text{implements}(C, I) \quad \Gamma[\text{this} \mapsto C, \text{fields}(C)] \vdash \overline{M}}{\Gamma \vdash \mathtt{class}\ C\ \mathtt{implements}\ \overline{I}\ \{\overline{T\ f}; \overline{M}\}}\ (\text{T-Class}) \qquad \frac{\Gamma[\overline{x} \mapsto \overline{T}] \vdash s \quad \forall CL \in \overline{CL} \cdot \Gamma \vdash L}{\Gamma \vdash \overline{CL}\ \{\overline{T\ x}; s\}}\ (\text{T-Program})$$

Fig. 4. The type system for the concurrent object level of ABS.

The declaration of a data type is well typed if its constructors are well typed, by rule T-DataDecl. Case expressions are well typed by rules T-Case and T-Case2 if all branches type check to the same type. (Here, **case2** is run-time syntax.) The pattern must have the same type A as the case expression. A branch is well typed by rule T-Branch if there is an extension of Γ which adds types for the variables in the pattern p and which allows the expression e to be type-checked. The desired mapping can be reconstructed from A and p by induction over the structure of p as follows: If A is a type variable, then p is a variable and $psubst(p, A) = [p \mapsto A]$. Otherwise, we proceed by induction over p. If $p = x$, $psubst(p, A) = [p \mapsto A]$. If $p = t$ or $p = \_$, $psubst(p, A) = \varepsilon$. Otherwise $p = Co(p_1, \ldots, p_n)$ such that $\Gamma(Co) = A_1, \ldots, A_n \to A$, and $psubst(p, A) = psubst(p_1, A_1) \circ \ldots \circ psubst(p_n, A_n)$. Remark that the type of a variable x in p may be different from $\Gamma(x)$, which reflects the change of scope.

Subtyping $T \preceq T'$ is nominal and reflects the extension relation on interfaces. For simplicity we extend the subtype relation such that $C \preceq I$ if class C implements interface I, and type object identifiers by their class and object references by their interface. We don't consider subtyping for data types or type variables.

The concurrent object level of the type system is given in Fig. 4. By rule T-Program, a program is well typed if its classes and main body are well typed and by T-Class, a class is well typed if its methods are well typed in the typing context extended by the typing of its fields. By T-Method, a method declaration is well typed if its body is well typed in the typing context extended

$$
\begin{array}{rl@{\qquad}rl}
e & ::= \texttt{case2}\ t\ \{\overline{br}\} \mid \ldots & s & ::= \texttt{cont}(f) \mid \ldots \\
cn & ::= \epsilon \mid \mathit{fut} \mid \mathit{object} \mid \mathit{invoc} \mid \mathit{cog} \mid cn\ cn & \mathit{cog} & ::= \mathit{cog}(c, \mathit{act}) \\
\mathit{fut} & ::= \mathit{fut}(f, \mathit{val}) & \mathit{val} & ::= v \mid \bot \\
\mathit{object} & ::= \mathit{ob}(o, a, p, q) & a & ::= T\ x\ v \mid a, a \\
\mathit{process} & ::= \{a \mid s\} \mid \texttt{error} & p & ::= \mathit{process} \mid \texttt{idle} \\
q & ::= \epsilon \mid \mathit{process} \mid q\ q & v & ::= o \mid f \mid b \mid t \\
\mathit{invoc} & ::= \mathit{invoc}(o, f, m, \overline{v}) & \mathit{act} & ::= o \mid \varepsilon
\end{array}
$$

Fig. 5. Runtime syntax; o, f, and c are object, future, and cog identifiers.

by the typing of formal parameters and local variables. We add a name **return** to the typing context, which binds to the return type of the method. The rules for skip, suspend, assignment, composition, conditional, and while are standard. By T-Return, a return statement is well typed if its expression types to the type of the **return** variable. In rule T-Await, **await** g is well typed if g is of type Bool, rule T-And decomposes guards, and by rule T-Poll a reply-guard $x?$ is a Bool if x is a future reference. Similarly, by T-Get, the get operation unwraps the type of a future. By T-New, object creation has a type T if the actual parameters can be typed to the types of the formal parameters (given by a function ptypes and T is among the declared interfaces of the class. By T-AsyncCall, an asynchronous method call has type $\texttt{fut}\langle T\rangle$ if the corresponding synchronous call has type T. By T-SyncCall, a call to a method m has type T if its actual parameters have types $\overline{T}$ and the signature $\overline{T} \to T$ matches a signature for m in the known interface of the callee (given by an auxiliary function *match*). Remark that for internal calls, Γ(this) gets as type the class of **this**, which allows internal methods to be invoked. We omit the definitions of the auxiliary functions of the type system, which are straightforward; e.g., fields(C) returns the typing context given by the attributes of C.

5 An Operational Semantics for ABS

The operational semantics of ABS is presented as a transition system in an SOS style [31]. Rules apply to subsets of configurations (the standard context rules are not listed). For simplicity we assume that configurations can be reordered to match the left hand side of the rules (i.e., matching is modulo associativity and commutativity as in rewriting logic [28]). A run is a possibly nonterminating sequence of rule applications. When auxiliary functions are used in the semantics, these are evaluated in between the application of transition rules in a run.

Runtime Configurations. The runtime syntax is given in Fig. 5. *Configurations* cn are sets of objects, invocation messages, concurrent object groups (cogs) and futures. The associative and commutative union operator on configurations is denoted by whitespace and the empty configuration by ε. These configurations live inside curly brackets; in the term $\{cn\}$, cn captures the *entire* configuration. A *substitution* is a mapping from variable names to values (for convenience, we here associate the declared type of the variable with the binding). An *object* is a term

$ob(o, a, p, q)$ where o is the object's identifier, a a substitution representing the object's fields, p an *active process*, and q a *pool of suspended processes*. A process p consists of a substitution l of local variable bindings and a list s of statements, denoted by $\{l|s\}$ when convenient. In an *invocation message* $invoc(o, f, m, \overline{v})$, o is the callee, f the future to which the call's result is returned, m the method name, and $\overline{v}$ the call's actual parameter values. A *cog* only contains an identifier c and the currently active object o, or ϵ if no object of the cog is currently active (i.e., all objects have the idle process as active process). A *future* $fut(f, v)$ has an identifier f and a reply value v (which is $\perp$ when the future's reply value has not been received). Values are object and future identifiers, Boolean expressions, and ground terms from the functional language. For simplicity, classes are not represented explicitly in the semantics, but may be seen as static tables.

5.1 A Reduction System for ABS Functional Expressions

The evaluation $[\![e]\!]_\sigma$ of functional expressions e, given in Figure 6, happens in the context of a *substitution* σ. and is defined inductively over the data types of the functional language. Let the syntactic category t consist of ground terms; i.e., constructor terms and built-in constants such as, e.g., **`null`** and object names.

For simplicity, we let function evaluation be strict. Thus, for every (user defined) function definition **`def`** $T\ fn(\overline{T\ x}) = e_{fn}$, the evaluation of a function call $[\![fn(\overline{e})]\!]_\sigma$ reduces to the evaluation of the corresponding expression $[\![e_{fn}]\!]_{\overline{x}\mapsto\overline{t}}$ when the arguments $\overline{e}$ have already been reduced to ground terms $\overline{t}$. (Note the change in scope. Since functions are defined independently of the context where they are used, we here assume that the expression e does not contain free variables and the substitution σ does not apply in the evaluation of e.) In the case of pattern matching, variables in the pattern p may be bound to ground terms in the term t. Thus the substitution context for evaluating the right hand side e of the branch $p \to e$ extends the current substitution σ with bindings that occurred during the pattern matching. Let the function $match(p, t)$ return a substitution σ such that $\sigma(p) = t$ (if there is no match, it returns the empty substitution $\perp$). For a typing context Γ and a substitution σ, we say that σ is *well typed* in Γ, denoted $\Gamma \vdash \sigma$, if $\Gamma \vdash \sigma(x) : \Gamma(x)$.

Lemma 1 (Type preservation). *Let Γ be a typing context and let σ be a substitution such that $\Gamma \vdash \sigma$. If $\Gamma \vdash e : A$, then there is a $B \preceq A$ such that $\Gamma \vdash [\![e]\!]_\sigma : B$.*

Proof (sketch). By structural induction over e. The base cases are straightforward. For the remaining cases, we need that typing is preserved under type substitutions [30] and that well-typedness is preserved when σ is extended.

5.2 The Operational Semantics for Concurrent Objects in ABS

Evaluating Guards. Given a substitution σ and a configuration cn, we lift the functional evaluation function and denote by $[\![g]\!]^{cn}_\sigma$ an evaluation function which

$$[\![b]\!]_\sigma = b$$
$$[\![x]\!]_\sigma = \sigma(x)$$
$$[\![t]\!]_\sigma = t$$
$$[\![Co(\overline{e})]\!]_\sigma = Co(\overline{[\![e]\!]_\sigma})$$

$$[\![fn(\overline{e})]\!]_\sigma = \begin{cases} [\![e_{fn}]\!]_{\overline{x}\mapsto\overline{t}} & \text{if } \overline{e} = \overline{t} \\ [\![fn(\overline{[\![e]\!]_\sigma})]\!]_\sigma & \text{otherwise} \end{cases}$$

$$[\![\textsf{case } e\ \{\overline{br}\}]\!]_\sigma = [\![\textsf{case2 } [\![e]\!]_\sigma\ \{\overline{br}\}]\!]_\sigma$$
$$[\![\textsf{case2 } t\ \{p \Rightarrow e; \overline{br}\}]\!]_\sigma = \begin{cases} [\![e]\!]_{\sigma \circ match(p,t)} & \text{if } match(p,t) \neq \bot \\ [\![\textsf{case2 } t\ \{\overline{br}\}]\!]_\sigma & \text{otherwise} \end{cases}$$

Fig. 6. The evaluation of functional expressions.

reduces guards g to data values (the state configuration is needed to evaluate future variables). Let $[\![g_1 \wedge g_2]\!]^{cn}_\sigma = [\![g_1]\!]^{cn}_\sigma \wedge [\![g_2]\!]^{cn}_\sigma$, $[\![x?]\!]^{cn}_\sigma = \text{true}$ if $[\![x]\!]^{cn}_\sigma = f$ and $fut(f, v) \in cn$ for some value $v \neq \bot$, otherwise $[\![x?]\!]^{cn}_\sigma = \text{false}$, and $[\![b]\!]^{cn}_\sigma = [\![b]\!]_\sigma$.

Auxiliary functions. If T is the return type of a method m in a class C, we let $bind(o, f, m, \overline{v}, C)$ return a process resulting from the activation of m in C with actual parameters $\overline{v}$, callee o and associated future f. If binding succeeds, this process has a local variable *destiny* of type $\mathsf{fut}\langle T\rangle$ bound to f, and the method's formal parameters are bound to $\overline{v}$. If binding does *not* succeed, we get the **`error`** process. The function $atts(C, \overline{v}, o, c)$ returns the initial state of an instance of class C, in which the formal parameters are bound to $\overline{v}$ and the reserved variables *this* and *cog* are bound to the object identity o and the concurrent object group c, respectively. The function $init(C)$ returns an activation of the *init* method of C, if defined. Otherwise it returns the *idle* process. The predicate $fresh(n)$ asserts that a name n is globally unique (where n may be an identifier for an object, a future or a cog). Let *idle* denote the idle process.

Transition rules transform state configurations into new configurations, and are given in Figs. 7 and 8. We define different assignment rules for side effect free expressions (Assign1 and Assign2), object creation (New-Object and New-Cog-Object), method calls (Async-call, Cog-Sync-Call and Self-Sync-Call), and future dereferencing (Read-Fut). Rule Skip consumes a **`skip`** in the active process. Here and in the sequel, the variable s will match any (possibly empty) statement list. Rules Assign1 and Assign2 assign the value of expression e to a variable x in the local variables l or in the fields a, respectively. Rules Cond1 and Cond2 branch the execution depending on the value obtained by evaluating the expression e. (We omit the standard rule for **`while`**.)

Process Suspension and Activation. Three operations manipulate a process pool q; $q \cup p$ adds a process p to q, $q \setminus p$ removes p from q, and $\text{select}(q, a, cn)$ selects a process from q (if q is empty or no process is *ready*, the result is the idle process [22]). The actual definitions of these operations are left undefined; different definitions correspond to different scheduling policies for processes, although care must be taken that select always gives the initial process of an object the highest priority (otherwise another process might see uninitialized object state).

$$\frac{}{ob(o,a,\{l|\text{skip};s\},q) \rightarrow ob(o,a,\{l|s\},q)}\ (\textsc{Skip})$$

$$\frac{x \in \text{dom}(l) \quad v = [\![e]\!]_{(a \circ l)}}{ob(o,a,\{l|x = e;s\},q) \rightarrow ob(o,a,\{l[x \mapsto v]|s\},q)}\ (\textsc{Assign1})$$

$$\frac{x \in \text{dom}(a) \quad v = [\![e]\!]_{(a \circ l)}}{ob(o,a,\{l|x = e;s\},q) \rightarrow ob(o,a[x \mapsto v],\{l|s\},q)}\ (\textsc{Assign2})$$

$$\frac{o' = [\![e]\!]_{(a \circ l)} \quad \overline{v} = [\![\overline{e}]\!]_{(a \circ l)} \quad \text{fresh}(f)}{ob(o,a,\{l|x = e!m(\overline{e});s\},q) \rightarrow ob(o,a,\{l|x = f;s\},q)\ invoc(o',f,m,\overline{v})\ fut(f,\bot)}\ (\textsc{Async-Call})$$

$$\frac{p' = \text{bind}(o,f,m,\overline{v},\text{class}(o))}{ob(o,a,p,q)\ invoc(o,f,m,\overline{v}) \rightarrow ob(o,a,p,q \cup p')}\ (\textsc{Bind-Mtd})$$

$$\frac{[\![e]\!]_{(a \circ l)}}{ob(o,a,\{l|\text{if } e \text{ then } s_1 \text{ else } s_2 \text{ fi};s\},q) \rightarrow ob(o,a,\{l|s_1;s\},q)}\ (\textsc{Cond1})$$

$$\frac{\neg[\![e]\!]_{(a \circ l)}}{ob(o,a,\{l|\text{if } e \text{ then } s_1 \text{ else } s_2 \text{ fi};s\},q) \rightarrow ob(o,a,\{l|s_2;s\},q)}\ (\textsc{Cond2})$$

$$\frac{[\![g]\!]^{cn}_{(a \circ l)}}{\{ob(o,a,\{l|\text{await } g;s\},q)\ cn\} \rightarrow \{ob(o,a,\{l|s\},q)\ cn\}}\ (\textsc{Await1})$$

$$\frac{\neg[\![g]\!]^{cn}_{(a \circ l)}}{\{ob(o,a,\{l|\text{await } g;s\},q)\ cn\} \rightarrow \{ob(o,a,\{l|\text{suspend};\text{await } g;s\},q)\ cn\}}\ (\textsc{Await2})$$

$$\frac{v = [\![e]\!]_{(a \circ l)} \quad l(\text{destiny}) = f}{ob(o,a,\{l|\text{return } e;s\},q)\ fut(f,\bot) \rightarrow ob(o,a,\{l|s\},q)\ fut(f,v)}\ (\textsc{Return})$$

$$\frac{v \neq \bot \quad f = [\![e]\!]_{(a \circ l)}}{ob(o,a,\{l|x = e.\text{get};s\},q)\ fut(f,v) \rightarrow ob(o,a,\{l|x = v;s\},q)\ fut(f,v)}\ (\textsc{Read-Fut})$$

Fig. 7. ABS Semantics (1).

Let $\emptyset$ denote the empty pool. Rule Suspend suspends the active process to the process pool, leaving the active process idle. Rule Await1 consumes **await** g if g evaluates to true in the current state of the object, rule Await2 adds a suspend statement in order to suspend the process if the guard evaluates to false. Rule Activate selects a process from the process pool for execution if this process is *ready* to execute, i.e., if it would not directly be resuspended or block the processor [22]. These rules ensure that a process can only be scheduled if the cog associated with the object is idle, and that a process activation always occurs together with the object acquiring the cog. Synchronous calls and synchronous self-calls, which also influence scheduling, are discussed below.

Communication and Object Creation. Rule Async-Call sends an invocation message to o' with the unique identity f of a new future (since fresh(f)), the method name m, and actual parameters $\overline{v}$. The return value of the new future is undefined (i.e., $\bot$). Rule Bind-Mtd consumes an invocation method and places the process p corresponding to the method activation in the callee's process pool. Note that a reserved local variable 'destiny' is used to store the identity of the future associated with the call. Rule Return places the return value into the call's associated future. Rule Read-Fut dereferences the future f if $v \neq \bot$. Note that if this attribute is $\bot$ the reduction in this object is *blocked*.

Rules Cog-Sync-Call and Cog-Sync-Return-Sched address synchronous method calls between two objects that share a common cog. For a synchronous call, possession of the cog directly transfers control from the calling object to

(New-Cog-Object)

$$\frac{\text{fresh}(o')\ \text{fresh}(c')\ p = \text{init}(C)\quad a' = \text{atts}(C, [\![\overline{e}]\!]_{(a\,o\,l)}, o', c')}{\begin{array}{c} ob(o, a, \{l|x = \text{new } C(\overline{e}); s\}, q) \\ \rightarrow ob(o, a, \{l|x = o'; s\}, q) \\ ob(o', a', p, \emptyset)\quad cog(c', o') \end{array}}$$

(New-Object)

$$\frac{\text{fresh}(o')\ p = \text{init}(C)\quad a' = \text{atts}(C, [\![\overline{e}]\!]_{(a\,o\,l)}, o', c)}{\begin{array}{c} ob(o, a, \{l|x = \text{new } C(\overline{e}); s\}, q)\ cog(c, o) \\ \rightarrow ob(o, a, \{l|x = o'; s\}, q)\ cog(c, o) \\ ob(o', a', \text{idle}, \{p\}) \end{array}}$$

(Activate)

$$\frac{p = \text{select}(q, a, cn)\ c = a(\text{cog})}{\begin{array}{c} \{ob(o, a, \text{idle}, q)\ cog(c, \epsilon)\ cn\} \\ \rightarrow \{ob(o, a, p, q \backslash p)\ cog(c, o)\ cn\} \end{array}}$$

(Suspend)

$$\frac{c = a(\text{cog})}{\begin{array}{c} ob(o, a, \{l|\text{suspend}; s\}, q)\ cog(c, o) \\ \rightarrow ob(o, a, \text{idle}, q \cup \{l|s\})\ cog(c, \epsilon) \end{array}}$$

(Release-Cog)

$$\frac{c = a(\text{cog})}{\begin{array}{c} ob(o, a, \text{idle}, q)\ cog(c, o) \\ \rightarrow ob(o, a, \text{idle}, q)\ cog(c, \epsilon) \end{array}}$$

(Self-Sync-Call)

$$\frac{\begin{array}{c} f' = l(\text{destiny}) \\ o = [\![e]\!]_{(a\,o\,l)}\quad \overline{v} = [\![\overline{e}]\!]_{(a\,o\,l)}\quad \text{fresh}(f) \\ \{l'|s'\} = bind(o, f, m, \overline{v}, class(o)) \end{array}}{\begin{array}{c} ob(o, a, \{l|x = e.m(\overline{e}); s\}, q) \\ \rightarrow ob(o, a, \{l'|s'; \text{cont}(f')\}, q \cup \{l|x = f.\text{get}; s\}) \\ fut(f, \bot) \end{array}}$$

(Cog-Sync-Return-Sched)

$$\frac{a'(\text{cog}) = c\quad l'(\text{destiny}) = f}{\begin{array}{c} ob(o, a, \{l|\text{cont}(f)\}, q)\ cog(c, o) \\ ob(o', a', \text{idle}, q' \cup \{l'|s\}) \\ \rightarrow ob(o, a, \text{idle}, q)\ cog(c, o') \\ ob(o', a', \{l'|s\}, q') \end{array}}$$

(Rem-Sync-Call)

$$\frac{o' = [\![e]\!]_{(a\,o\,l)}\quad \text{fresh}(f)\quad a(\text{cog}) \neq a'(\text{cog})}{\begin{array}{c} ob(o, a, \{l|x = e.m(\overline{e}); s\}, q)\ ob(o', a', p, q') \\ \rightarrow ob(o, a, \{l|f = e!m(\overline{e}); x = f.\text{get}; s\}, q) \\ ob(o', a', p, q') \end{array}}$$

(Self-Sync-Return-Sched)

$$\frac{l'(\text{destiny}) = f}{\begin{array}{c} ob(o, a, \{l|\text{cont}(f)\}, q \cup \{l'|s\}) \\ \rightarrow ob(o, a, \{l'|s\}, q) \end{array}}$$

(Cog-Sync-Call)

$$\frac{\begin{array}{c} o' = [\![e]\!]_{(a\,o\,l)}\quad \overline{v} = [\![\overline{e}]\!]_{(a\,o\,l)}\quad \text{fresh}(f) \\ a'(\text{cog}) = c\quad f' = l(\text{destiny}) \\ \{l'|s'\} = bind(o', f, m, \overline{v}, class(o')) \end{array}}{\begin{array}{c} ob(o, a, \{l|x = e.m(\overline{e}); s\}, q) \\ ob(o', a', \text{idle}, q')\ cog(c, o) \\ \rightarrow ob(o, a, \text{idle}, q \cup \{l|x = f.\text{get}; s\})\ fut(f, \bot) \\ ob(o', a', \{l'|s'; \text{cont}(f')\}, q')\ cog(c, o') \end{array}}$$

Fig. 8. ABS Semantics (2).

the callee and back, bypassing the Suspend and Activate rules. A special cont instruction is inserted at the end of the statement list of the new process in rule Cog-Sync-Call, which is then used to re-activate the caller process in rule Cog-Sync-Return-Sched. Synchronous self-calls are implemented similarly by Self-Sync-Call and Self-Sync-Return-Sched. The cog invariant (only one object with a non-idle process per cog) is maintained by these rules.

Finally, New-Object creates a new object with a unique identifier o'. The object's fields are given default values by $\text{atts}(C, [\![\overline{e}]\!]_{(a\,o\,l)}, o', c)$, extended with the actual values $\overline{e}$ for the class parameters (evaluated in the context of the creating process), o' for this and with the creating object's cog c. (For New-Cog-Object, the cog c of the new object is also created with a fresh name.) In order to instantiate the remaining attributes, the process p is queued (we assume that this process reduces to idle if $\text{init}(C)$ is unspecified in the class definition, and that it asynchronously calls `run` if the latter is specified). p is

not directly scheduled in order to uphold the cog invariant (only one object per cog is active), hence any scheduling policy must take care to always schedule an initial process p with highest priority. The rule NEW-COG-OBJECT is equivalent to NEW-OBJECT, except that a fresh cog is created with o' as its (only) active object, and the initial process p is directly scheduled.

6 Subject Reduction for ABS

The *initial state* of a well typed program consists of an object $ob(\text{start}, \varepsilon, p, \emptyset)$, where the process p corresponds to the activation of the program's main method. A *run* is a sequence of reductions of an initial state according to the rules of the operational semantics. We now show that a run from a well typed initial configuration will maintain well typed configurations; in particular, that substitutions remain well typed and that method binding does not result in the **`error`** process.

Runtime Configurations. Typing rules are given for the runtime syntax given in Fig. 5. The typing context of the runtime configurations extends the static typing context with types for dynamically created values, i.e., object and future identifiers. Object identifiers are typed by the class of the created object.

Typing Rules for Runtime Configurations. Let $\Delta \vdash_R \mathit{config}$ **`ok`** express that the runtime configuration *config* is well typed in the typing context Δ. The typing rules for runtime configurations are given in Fig. 9. In T-OBJECT, the premise $\text{fields}(\Delta(o)) = [\overline{x} \mapsto \overline{T}]$ asserts that the object attributes have the types declared in its class. If a configuration is well typed in a typing context Δ, the substitutions a and l (for any object and any process) are well typed in Δ. Consequently, by Lemma 1, function evaluation in ABS processes preserve typing.

Well-typedness assumptions for the auxiliary functions of the operational semantics. Let C be a class with formal parameters $\overline{x}$ of types $\overline{T}$. We assume that $\mathit{init}(C)$ returns a well typed process. We assume that $\mathit{atts}(C, \overline{v}, o, c)$ returns a well typed substitution if $\overline{v}$ have types $\overline{T}$ and o and c are object and cog identifiers, respectively. If C implements a method m with return type T and formal parameters $\overline{x}$ of types $\overline{T}$, we assume that $\mathit{bind}(o, f, m, \overline{v}, C)$ returns a well typed process if f has type $\mathsf{fut}\langle T\rangle$ and $\overline{v}$ have the types $\overline{T}$.

We prove that the object corresponding to the main method of a well typed program is well typed (Lemma 2) and show that the well-typedness of runtime configuration is preserved by reductions (Theorem 1).

Lemma 2. *Let P $\{\overline{T}\ \overline{x}; s\}$ be an ABS program. If $\Gamma \vdash P$ $\{\overline{T}\ \overline{x}; s\}$ for some typing context Γ, then $\Gamma \vdash_R ob(\mathit{start}, \varepsilon, \{\overline{T}\ \overline{x}\ \mathit{default}(\overline{T})|s\}, \emptyset)$* **`ok`**.

Proof. Let $\Gamma' = \Gamma[\overline{x} \mapsto \overline{T}]$. It is obvious that $\Gamma' \vdash_R \overline{T}\ \overline{x}\ \text{default}(\overline{T})$ **`ok`**. By assumption, $\Gamma \vdash P$ $\{\overline{T}\ \overline{x}; s\}$, so $\Gamma' \vdash s$.

Theorem 1 (Subject Reduction). *If $\Delta \vdash_R cn$* **`ok`** *and $cn \to cn'$, then there is a Δ' of Δ such that $\Delta \subseteq \Delta'$ and $\Delta' \vdash_R cn'$* **`ok`**.

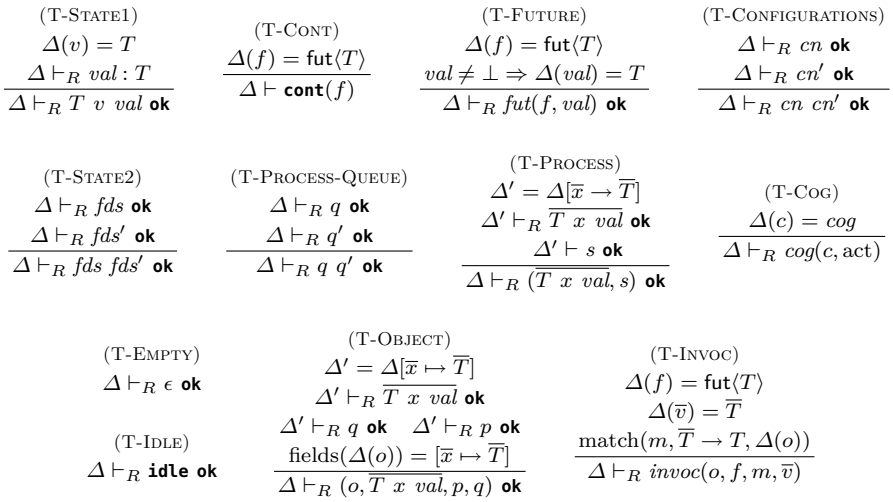

Fig. 9. The typing rules for runtime configurations.

Proof. The proof is by induction over the transition rules. We assume that the class definitions (which are omitted from the runtime syntax since they do not change) are well typed. We may assume that objects, futures, and messages not affected by a transition remain well typed, so these are ignored below. We show the cases related to object creation and asynchronous communication in ABS. The remaining cases are fairly straightforward.

Object Creation. For rule New-Object, assume that $\Delta \vdash_R ob(o, a, \{l|x = \text{new } C(\overline{e}); s\}, q)$ **ok**, that $\Delta(x) = I$, and that implements(C, I) (so $C \preceq I$). Since fresh(o'), let $\Delta' = \Delta[o' \mapsto C]$. Obviously, $\Delta' \vdash_R ob(o, a, \{l|x = o'; s\}, q)$ **ok**. By assumption, a' and p are well typed in o', and $\Delta' \vdash_R ob(o', a', \text{idle}, \{p\})$ **ok**.

Asynchronous calls. Let $\Delta \vdash_R ob(o, a, \{l|x = e!m(\overline{e}\}, q)$ **ok**. We first consider the case $e \neq$ **this**. By AsyncCall, we may assume that $\Delta \vdash e!m(\overline{e}) : \mathsf{fut}\langle T\rangle$ and by Assign that $\Delta(x) = \mathsf{fut}\langle T\rangle$. Therefore, $\Delta \vdash e : I$ and $\Delta \vdash \overline{e} : \overline{T}$ such that match$(m, \overline{T} \to T, I)$. Let $[\![e]\!]_{aol} = o'$ and let $\Delta(o') = C$. By Lemma 1, $C \preceq I$ and since by assumption class definitions are well typed , it follows that for any class C that implements interface I we have match$(m, \overline{T} \to T, C)$. Similarly, $\Delta' \vdash \overline{e} : \Delta'(\overline{v})$. Let $\Delta' = \Delta[f \mapsto \mathsf{fut}\langle T\rangle]$. Since fresh$(f)$ we may assume that $f \notin \mathrm{dom}(\Delta)$, so if $\Delta \vdash_R cn$ **ok**, then $\Delta' \vdash_R cn$ **ok**. Since $\Delta \vdash e!m(\overline{e}) = \Delta'(f)$, we get $\Delta' \vdash_R ob(o, a, \{l|x = f; s\}, q)$ **ok**. Furthermore, $\Delta' \vdash invoc(o', f, m, \overline{v})$ **ok** and $\Delta' \vdash_R fut(f, \bot)$ **ok**. The case $e =$ **this** is similar, but uses the class of **this** directly for the match (so internal methods are also visible).

Method Binding. Let $C = \Delta(o)$. By assumption $\Delta \vdash_R invoc(o, f, m, \overline{v})$ **ok** and $\Delta \vdash_R ob(o, a, p, q)$ **ok**, so $\Delta(f) = \mathsf{fut}\langle T\rangle$, $\Delta(\overline{v}) = \overline{T}$, and match$(m, \overline{T} \to T, C)$. Let $\overline{x}$ be the formal parameters of m in C. Consequently, the auxiliary function bind$(o, f, m, \overline{v}, C)$ returns a process $\{l[\overline{T}\ \overline{x}\ \overline{v}, \mathsf{fut}\langle T\rangle \text{ destiny } f]|s\}$

which is well typed in $\Delta[\text{fields}(C)]$, and it follows that $\Delta \vdash_R ob(o, a, p, q \cup \{\text{bind}(o, f, m, \overline{v}, C)\})$ **ok**.

Method Return. By assumption, $\Delta \vdash_R ob(o, a, \{l|\text{return } e; s\}, q)$ **ok** and $\Delta \vdash_R fut(f, \bot)$ **ok**. Obviously, $\Delta \vdash_R ob(o, a, \{l|s\}, q)$ **ok**. Since $l(\text{destiny}) = f$ and l is well typed, we know that $\Delta(\text{destiny}) = \Delta(f)$. Let $\Delta(f) = \mathsf{fut}\langle T\rangle$. By Lemma 1, $\Delta(v) \preceq T$ and $\Delta \vdash_R fut(f, v)$ **ok**.

Reading a future. Let $\Delta(f) = \mathsf{fut}\langle T\rangle$. By assumption, $\Delta \vdash_R ob(o, a, \{l|x = e.get; s\}, q)$ **ok** and $\Delta \vdash_R fut(f, v)$ **ok**, and $[\![e]\!]_{a \circ l} = f$. Consequently, $\Delta \vdash_R e.\mathbf{get} : T$ and $\Delta(v) = T$, $\Delta \vdash x = v$, and $\Delta \vdash_R ob(o, a, \{l|x = v; s\}, q)$ **ok**.

7 Tool Support

The ABS language is being used and developed in the EU project HATS. Considerable effort has been made towards language implementation and tool support. There is support for editing, compiling, running and visualizing ABS models in the Emacs editor and in the Eclipse integrated development environment.

Compiler Frontend. All ABS tools use a common compiler frontend which supplies basic parsing, type-checking, and error reporting. The compiler frontend is implemented using the JastAdd toolkit and provides an object-oriented, type-annotated syntax tree representing an ABS model. All backend implementations, code analyzers, etc. are implemented on top of this common base. At present there are two language backends, making ABS executable on the Maude rewriting engine and the Java virtual machine, with more backends planned.

The Maude Backend is a translation of the operational semantics given in this paper into equational logic for the functional level of ABS and rewriting logic for the concurrent object level. This semantics is executed as a language interpreter using the Maude tool [11]. Compiling an ABS model into Maude results in a set of class and function definitions (since all type checking is done at compile time, interface and datatype declarations do not have runtime representations). A special, hidden class implements the (class-less) *main* method; starting an ABS model in Maude means instantiating an object of that class.

The conciseness and high level of abstraction of Maude have made this backend a good fit for experiments with alternative language constructs and semantics. Maude also provides model-checking functionality, but the large size of each state, as well as the nondeterministic scheduling and concurrent execution that ABS provide and the resulting combinatorial explosion, make model-checking ABS models of realistic size very difficult in practice.

The Java Backend provides a translation of ABS models into Java source code, which is compiled into bytecode using the standard Java tool chain. The Java backend uses a Java translation similar to the one for JCoBox [32], which proved to be very efficient. Compared to JCoBox, the generated code of ABS has better support for configuring the scheduling strategies, for system observation, and

debugging. The ABS *main* block is translated into a standard Java main method so the generated code can be executed like standard Java programs.

The Java backend provides higher execution speed, an integration into existing Java tools, and the potential for integrating "native" or library functionality (e.g., file handling) into the language. Hence, the Java and Maude backends provide complement and attractive features for the modeller.

8 Related Work

The concurrency model provided by concurrent objects and Actor-based computation, in which software units with encapsulated processors communicate asynchronously, is increasingly attracting attention due to its intuitive and compositional nature (e.g., [2, 6, 10, 14, 19, 35]). ABS uses the communication mechanisms of Creol for remote communication, based on asynchronous method calls and first-class futures [14]. A distinguishing feature of Creol is the cooperative scheduling between asynchronously called methods [22], which allows active and reactive behavior to be combined within objects as well as compositional verification of partial correctness properties [3, 14]. Creol's model of cooperative scheduling has recently been generalized from single objects to groups of objects in a Java extension called JCoBox [32], which form the basis for cogs in ABS.

Formal models are useful to clarify the intricacies of object orientation and may thus contribute to improve programming languages, making programs easier to understand, maintain, and analyze. Object calculi such as the ς-calculus [1] and its concurrent extension [18] aim at a direct expression of object-oriented features such as self-reference, encapsulation, and method calls, but asynchronous method calls are not addressed. This also applies to Obliq [9], a programming language using similar primitives which targets distributed concurrent objects. The concurrent object calculus of Di Blasio and Fisher [15] provides both synchronous and asynchronous method calls. In contrast to ABS, return values are discarded when methods are called asynchronously and the two ways of calling a method have different semantics. Caromel, Henrio, and Serpette propose ASP [10], a concurrent object calculus with asynchronous method calls which support return values using first-class futures. In contrast to ABS, their futures are transparent (i.e., there is no polling and the get-operation is implicit). Furthermore, communication is ordered to make reductions deterministic.

The internal concurrency model of cogs in ABS follows Creol's concept of *cooperative scheduling*, but lifted from the level of objects to the level of cogs. Synchronous method calls inside a cog are reentrant, which allows standard recursive programming of internal imperative data structures. Cogs in ABS may be compared to monitors [20] or to thread pools executing on a single processor. In contrast to monitors, explicit signaling is avoided. Sufficient signaling is ensured by the semantics, which significantly simplifies reasoning [13]. However, general monitors may be encoded in the language [22]. In contrast to thread pools, processor release is explicit. The activation of suspended processes is nondeterministically handled by an unspecified scheduler. Consequently, intra-object

concurrency is similar to the interleaving semantics of concurrent process languages [5, 16], and each process resembles a series of guarded atomic actions (discarding local variables). Internal reasoning control is facilitated by the explicit declaration of release points, at which class invariants should hold [3, 17].

The type system presented in this paper resembles that of Featherweight Java [21], a core calculus for Java, because of its nominal approach. Featherweight Java is class based and uses a class table to represent class information in its type system. Subtyping is the reflexive and transitive closure of the subclass relation. In contrast ABS cleanly distinguishes classes and types. Creol combined asynchronous method calls and interfaces as in ABS with class inheritance, choice operators, and a notion of *cointerface* at the interface level to accomodate type-safe callbacks [23]. Creol's type system used an effect system [26] to track types implicitly associated with the untyped futures, which allowed a flexible reuse of future variables for method calls with different return types. By means of backwards analysis, the effect system could insert deallocation operations to garbage collect inaccessible futures from the runtime system, depending on the local control flow [24]. In contrast, futures in ABS have explicit types for return values, which restricts the reuse of future variables but allows a type analysis without effects. Furthermore, compared to previous work on Creol, this paper formalizes user-defined data types and functions in the context of concurrent objects. The type safety results presented in this paper for ABS show how the typing context is dynamically extended when new objects and futures are created.

9 Conclusion

This paper has discussed the design of ABS, an abstract behavioral specification language for the executable object-oriented design of distributed systems. The language is situated between design-oriented, foundational, and implementation-oriented languages by being abstract, yet executable. ABS is based on groups of concurrent objects which are encapsulated behing interfaces and do not share state. While the groups may execute in parallel, there is a cooperative model of interleaving concurrency inside each group, reflected by explicit processor release points in the language. This model of concurrency is inherently compositional and allows to reason about the behavior of the concurrent system in terms of monitor invariants and sequential object-oriented proof systems. The combination of a functional and a concurrent object level in the ABS language allows the modeler to focus the model on crucial parts of an imperative system, including its concurrency and synchronization mechanisms, by using functional data types to abstract from other parts of the internal data structures and by abstracting from specific scheduling policies and environmental properties. ABS is a formally defined specification language. In this paper we have defined the syntax, type system, and operational semantics of Core ABS, a formal calculus incorporating the basic features of ABS. We prove a subject reduction result showing that execution preserves well-typedness in the sense that method not understood errors do not occur for well typed ABS models.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, 1996.
2. G. A. Agha. *ACTORS: A Model of Concurrent Computations in Distributed Systems*. The MIT Press, 1986.
3. W. Ahrendt and M. Dylla. A system for compositional verification of asynchronous objects. *Science of Computer Programming*, 2010. In press.
4. E. Albert, S. Genaim, M. Gómez-Zamalloa, E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa. Simulating concurrent behaviors with worst-case cost bounds. Research Report 403, Dept. of Informatics, Univ. of Oslo, Jan. 2011. `http://einarj.at.ifi.uio.no/Papers/rr403.pdf`.
5. G. R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, 2000.
6. J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.
7. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Proc. CASSIS*, *LNCS* 3362, pages 49–69. Springer, 2005.
8. L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3), June 2004.
9. L. Cardelli. A language with distributed scope. *Comp. Sys.*, 8(1):27–59, 1995.
10. D. Caromel, L. Henrio, and B. P. Serpette. Asynchronous sequential processes. *Information and Computation*, 207(4):459–495, 2009.
11. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. *All About Maude - A High-Performance Logical Framework*, *LNCS* 4350. Springer, 2007.
12. P. C. Clements. A survey of architecture description languages. In *Proc. Workshop on Software Specification and Design (IWSSD'96)*, pages 16–25. IEEE, 1996.
13. O.-J. Dahl. Monitors revisited. In *A Classical Mind, Essays in Honour of C.A.R. Hoare*, pages 93–103. Prentice Hall, 1994.
14. F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In *Proc. ESOP*, *LNCS* 4421, pages 316–330. Springer, Mar. 2007.
15. P. Di Blasio and K. Fisher. A calculus for concurrent objects. In *Proc. CONCUR*, *LNCS* 1119, pages 655–670. Springer, Aug. 1996.
16. E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, Aug. 1975.
17. J. Dovland, E. B. Johnsen, and O. Owe. Observable behavior of dynamic systems: Component reasoning for concurrent objects. In *Proc. Foundations of Interactive Computation (FInCo'07)*, *ENTCS* 203: 19–34. Elsevier, 2008.
18. A. D. Gordon and P. D. Hankin. A concurrent object calculus: Reduction and typing. In, *Proc. High-Level Concurrent Languages (HLCL)*, *ENTCS* 16(3), 1998.
19. P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comp. Sci.*, 410(2–3):202–220, 2009.
20. C. A. R. Hoare. Monitors: an operating systems structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.
21. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Prog. Lang. and Sys*, 23(3):396–450, 2001.
22. E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.

23. E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theor. Comp. Sci.*, 365(1–2):23–66, Nov. 2006.
24. E. B. Johnsen and I. C. Yu. Backwards type analysis of asynchronous method calls. *Journal of Logic and Algebraic Programming*, 77:40–59, 2008.
25. K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1–2):134–152, 1997.
26. J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proc. POPL*, pages 47–57. ACM Press, 1988.
27. J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proc. ESEC*, *LNCS* 989, pages 137–153. Springer, 1995.
28. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theor. Comp. Sci.*, 96:73–155, 1992.
29. R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, May 1999.
30. B. C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.
31. G. D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:17–139, 2004.
32. J. Schäfer and A. Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. In *Proc. ECOOP*, *LNCS* 6183, pages 275–299. Springer, 2010.
33. A. van Deursen and P. Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10(1):1–18, 2002.
34. J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modelling with UML*. Object Technology Series. Addison-Wesley, 1999.
35. A. Welc, S. Jagannathan, and A. Hosking. Safe futures for Java. In *Proc. OOPSLA*, pages 439–453. ACM, 2005.