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 designing executable models 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 for encapsulation, and cooperative scheduling of method activations inside concurrent objects. This feature combination results in a concurrent object-oriented model which is inherently compositional. We discuss central design issues for ABS and formalize 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 during execution; in particular, "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 addresses the specification of *executable formal models* for distributed object-oriented systems: it allows a high-level specification of a system, including its concurrency and synchronization mechanisms as well as local state updates. Thus ABS models capture the concurrent control flow of object-oriented systems, yet abstract away 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.

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

The target domain of ABS is *distributed systems*. In a distributed setting, the implementation details of other objects in the system are not necessarily 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 [34], which generalizes the concurrency model of Creol [24] from single concurrent objects to concurrent object groups.

The language supports asynchronous method calls, which trigger activities in other objects without transferring control from the caller, using first-class futures [13]. 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 concurrent object groups. 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 cooperative scheduling and the interface encapsulation mechanism, local reasoning about the concurrent object system can be done by suitable extensions of standard verification systems for sequential object-oriented programs. This approach is carried out in a number of papers [3, 13, 15, 18] for both Creol and Core ABS.

The present 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, imperative object-based programming, 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 focus on the creation of concurrent object groups, objects, and first-class futures.

An extended discussion and further technical details on the ABS language may be found in [14] while a program logic for Core ABS is in [15, Chap. 2].

2 Abstract Behavioral Specification

Specification languages can be categorized into three categories with 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 [36], FDL [35], and architectural description languages [11, 29].
– *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 [31], automata models [27], and object calculi [1, 23].
– *Implementation-oriented languages* focus on behavioral properties of implemented systems. Examples of implementation-oriented specification languages are JML [7] and Spec# [6].

Design-oriented languages often provide visual 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 minimalistic set of language features makes it cumbersome to develop models of real systems without complicated encodings; the resulting models typically do 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 [23] does not provide fields in objects. In contrast, implementation-oriented specification languages are restricted to the particular, often extremely complex, concurrency and synchronization mechanisms of their target language, and typically enforce particular solutions which may be undesirable at the design stage.

ABS is situated between these three categories of specification languages. It has in common with implementation-oriented languages that 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 objects.

3 The Design of ABS

3.1 The Overall Structure of ABS

ABS targets distributed object-oriented systems. The concurrency model of ABS is *two-tiered* and separates local, synchronous, shared-memory communication in the *lower tier* from asynchronous communication with only message passing in the *upper tier*. The lower tier is inspired by JCoBox [34] which generalizes the concurrency model of Creol [13, 24] from concurrent objects to concurrent object groups, so-called *cogs*. Cogs can be seen as object-based runtime components with their own object heaps. A cog's behavior is based on cooperative multi-tasking of external requests and synchronous internal method activations. Cooperative multi-tasking guarantees data-race freedom inside a cog and enables active and reactive behavior to be safely combined. Only asynchronous method calls can occur between different cogs, different cogs have no shared heap.

Complementing the concurrent object language, ABS supports user-defined data types with (first-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. Such data is immutable and can safely be exchanged between cogs. Using functional data types to realize most internal data structures in the cogs can significantly simplify the specification and verification of models. The value of functional expressions can be underspecified which is important in order to realize abstraction from concrete implementations.

ABS contains non-deterministic constructs, notably, 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 a modeling language, ABS makes no a priori assumptions about, e.g., concrete scheduling mechanisms. Importantly, underspecification and non-determinism do not preclude executability: the outcome of a non-deterministic transition step is a *finite set* of possible successor states which can be systematically inspected in simulation, analysis, and visualization tools.

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

3.2 Data Types, Functions, and Pattern Matching

ABS does *not* have primitive types for 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 and functions in ABS can be *polymorphic*; i.e., their definition may have type parameters.

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

```
data List<A> = Nil | Cons(A, List<A>);
def Bool contains<A>(List<A> l, A e) =
   case l { Nil => False;
            Cons(e, _) => True;
            Cons(_, xl) => contains(xl, e); };
```

3.3 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. Subtype polymorphism is permitted 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 classes of the two objects may differ.

Due to the typing of object variables by interfaces, the fields of another object cannot be accessed directly, only method calls to the object are possible. The object controls its own state; another object can only manipulate the state indirectly via the methods exported through an interface. In fact, interfaces are the only encapsulation mechanism of ABS objects and no access modifiers are provided. Since the class may support several interfaces, different methods may be exported to the environment through different interfaces; for example, a *super user* interface may export methods not seen through the normal *user* interface.

Example 2. Consider a peer-to-peer system whose participant *nodes* act both as *servers* and *clients*, and exchange *files* which are composed of *packets*. In this setting, it is important for each node to remain responsive during file transfer, both to serve concurrent requests and for simultaneous downloads. Files are transferred packet by packet, with one request per packet.

Client behavior is modeled by a `Client` interface, declaring a `getFile` method which is invoked from the outside, e.g., via a graphical user interface. Server behavior consists of a method `getFilenames` for querying the server about its available files, a method `getLength` for querying the length in packets for a given file, and a method `getPack` which requests the *n*'th packet of a given file.

```
type Packet = String;
type File = List<Packet>;

interface Server {
  List<Filename> getFilenames();
  Int getLength(Filename fId);
  Packet getPack(Filename fId,
    Int pNbr);
}
```

```
interface Client {
  File getFile(Server sId,
    Filename fId);
}

interface Peer
extends Client, Server {
}
```

3.4 The Concurrency Model of ABS

Conceptually, each cog has a dedicated processor and lives in a distributed environment with asynchronous and unordered communication. A set of objects is located within a cog. On the upper tier of the ABS concurrency model, 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 asynchronous method calls. Thus, an object has a set of processes to be executed, which originate from method activations. Among these, at most one process among 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. Hence, the amount of concurrency in an ABS model is directly reflected in the number of cogs introduced in the model. A Creol-like concurrent object model corresponds to an ABS model in which each object has its own cog.

Example 3. Consider a class `Node` which implements the peer behavior of Example 2, providing both client and server functionality. The `getFile` method first obtains the length of the requested file, then fetches the packets one by one. As it uses asynchronous method calls, the object can interleave the execution of `getFile` with answering requests in its server role and other invocations of `getFile`. (For brevity the implementation of the fields and remaining methods, such as `getFilenames`, is omitted. For a full model in ABS, see [14, App. E].)

```
class Node implements Peer {

 // Fields and other methods
 // of Node omitted

 File getFile(Server sId,
        Filename fId) {
  File file = Nil;
  Fut<Int> l1 = sId!getLength(fId);
  await l1?;
  Int lth = l1.get;
  while (lth > 0) {
   lth = lth - 1;
   Fut<Packet> l2
    = sId!getPack(fId, lth);
   await l2?;
   Packet pack = l2.get;
   file = Cons(pack, file);
  }
  return file;
 }
}
```

4 A Formal ABS Calculus

Core ABS is a formal calculus which simplifies the full ABS language by excluding, e.g., the module system, type synonyms, the predefined data types (except Bool), and annotations. However, Core ABS captures the central features of ABS. (A complete formalization of ABS exists in the rewriting logic of Maude [10].)

4.1 The Syntax of Core ABS

An ABS *model* P defines interfaces, classes, datatypes, functions, and a *main block* to configure the initial state (see Fig. 2). Objects are dynamically created from classes with attributes initialized to type-correct default values (e.g., **`null`** for object references) that may be reassigned 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 T consist of basic types B such as Bool and Int, as well as names D for data types and I for interfaces. In general, a type A

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

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.

may also contain type variables N (i.e., uninterpreted type names [32]). 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. Data type declarations Dd and function declarations F may optionally have type parameters. *Expressions* e include Boolean expressions b, variables x, (ground) terms t, the self-identifier **this**, the return address **destiny** of the method activation, 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 of case expressions are evaluated in the listed order. Patterns include wild cards $\_$, variables x, terms t, and constructor patterns $Co(\overline{p})$. Let the function $vars(p)$ return the set of *variables* in a pattern p, defined inductively by $vars(\_) = vars(t) = \emptyset$, $vars(x) = \{x\}$, and $vars(Co(p_1, \ldots, p_n)) = \bigcup_{i=1}^{n} vars(p_i)$.

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 $\overline{I}$ (specifying types for its instances), formal parameters and state variables $\overline{x}$ of types $\overline{T}$, and methods $\overline{M}$. (The *fields* 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 $\{\overline{T\ x}; s\}$. There are no type variables at the concurrent object level of ABS.

Right-hand side expressions *rhs* include object creation within the same cog (written "**new** $C(\overline{e})$") and in a fresh cog (written "**new cog** $C(\overline{e})$"), method calls, and (pure) expressions e. Statements are standard for sequential composition, assignment, **skip**, **if**, **while**, and **return** constructs. The statement **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

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

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

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. Consequently, explicit signaling is redundant in ABS.

Communication in ABS is based on asynchronous method calls, denoted $o!m(\overline{e})$, and synchronous method calls, denoted $o.m(\overline{e})$, where o is an object expression (i.e., an expression typed by an interface). 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; i.e., a variable which 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. Let e be an expression denoting a future variable. First, a return test $e?$ 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 $e.$**get**, which blocks all execution in the object until the return value is available.

When executed between objects in *different* cogs, then the statement sequence $x = o!m(\overline{e});\ v = x.$**get** amounts to a blocking, *synchronous call* 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** $x?;\ v = x.$**get** codes a non-blocking, *preemptable call*, abbreviated **await** $v = o.m(\overline{e})$. In many cases, these method calls with *implicit* futures provide sufficiently flexible concurrency control to the modeler.

4.2 The Type System of Core ABS

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 $\Gamma[N \mapsto V](N) = V$ and $\Gamma[N \mapsto V](x) = \Gamma(x)$ 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 $\Gamma \circ \Gamma'$ (where $\Gamma \circ \Gamma'(x) = \Gamma'(x)$ if $x \in dom(\Gamma')$ and $\Gamma \circ \Gamma'(x) = \Gamma(x)$ otherwise). We say that Γ' *extends* Γ, denoted $\Gamma \subseteq \Gamma'$, if $dom(\Gamma) \subseteq dom(\Gamma')$ and $\Gamma(x) = \Gamma'(x)$ for $x \in dom(\Gamma)$.

$$\text{(T-ConsDecl)}\quad \frac{\Gamma(Co) = \overline{A} \to D[\langle\overline{B}\rangle]}{\Gamma \vdash Co(\overline{A}) : D[\langle\overline{B}\rangle]} \qquad \text{(T-DataDecl)}\quad \frac{\Gamma \vdash \overline{Cons} : D[\langle\overline{A}\rangle]}{\Gamma \vdash \mathbf{data}\ D[\langle\overline{A}\rangle] = \overline{Cons};} \qquad \text{(T-Sub)}\quad \frac{\Gamma \vdash e : T \quad T \preceq T'}{\Gamma \vdash e : T'}$$

$$\text{(T-Bool)}\quad \Gamma \vdash b : \mathsf{Bool} \qquad \text{(T-Null)}\quad \Gamma \vdash \mathbf{null} : A \qquad \text{(T-Wildcard)}\quad \Gamma \vdash \_ : A \qquad \text{(T-Var)}\quad \frac{\Gamma(x) = A}{\Gamma \vdash x : A}$$

$$\text{(T-FuncExpr)}\quad \frac{\begin{array}{c}\mathrm{tmatch}(\overline{A}, \overline{C}) = \sigma \quad \sigma \neq \bot \\ \Gamma \vdash \overline{e} : \overline{C} \quad \Gamma(fn) = \overline{A} \to B\end{array}}{\Gamma \vdash fn(\overline{e}) : B\sigma} \qquad \text{(T-ConsExpr)}\quad \frac{\begin{array}{c}\Gamma \vdash \overline{e} : \overline{C} \quad \sigma \neq \bot \\ \mathrm{tmatch}(\overline{A}, \overline{C}) = \sigma \\ \Gamma(Co) = \overline{A} \to D[\langle\overline{B}\rangle]\end{array}}{\Gamma \vdash Co(\overline{e}) : D[\langle\overline{B}\rangle]\sigma}$$

$$\text{(T-FuncDecl)}\quad \frac{\begin{array}{c}\Gamma(fn) = \overline{B} \to C \\ \Gamma[\overline{x} \mapsto \overline{B}] \vdash e : C\end{array}}{\Gamma \vdash \mathbf{def}\ C\ fn[\langle\overline{A}\rangle](\overline{B\ x}) = e;} \qquad \text{(T-Branch)}\quad \frac{\begin{array}{c}\Gamma' \vdash p : A \quad \Gamma' \vdash e : B \\ \Gamma' = \Gamma \circ \mathrm{psubst}(p, A, \Gamma)\end{array}}{\Gamma \vdash p \Rightarrow e : A \to B} \qquad \text{(T-Case)}\quad \frac{\begin{array}{c}\Gamma \vdash e : A \\ \Gamma \vdash \overline{br} : A \to B\end{array}}{\Gamma \vdash \mathbf{case}\ e\ \{\overline{br}\} : B}$$

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

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 in Fig. 3.

The Functional Level of the ABS Type System is shown in Fig. 3. 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 T-Null. A variable is well-typed if declared in Γ by T-Var. In 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 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 $\bot$. (For example, 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 T-FuncDecl and T-FuncExpr. Additionally the function body is type-checked in Γ extended with the typing of formal parameters in T-FuncDecl, which may again be type variables.

The declaration of a data type is well-typed if its constructors are well-typed, by T-DataDecl. Case expressions are well-typed by T-Case if all branches type check to the same type. The pattern must have the same type A as the case expression. A branch $p \Rightarrow e$ is well-typed by T-Branch if there is an extension of Γ which adds types for the new variables in the pattern p and which allows the expression e to be type-checked. The desired mapping can be constructed 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 $psubst(p, A, \Gamma) = [p \mapsto A]$. Otherwise, we proceed by induction

$$
\begin{array}{ccccc}
\text{(T-}\textsc{Poll}\text{)} & \text{(T-}\textsc{Get}\text{)} & \text{(T-}\textsc{Skip}\text{)} & \text{(T-}\textsc{Await}\text{)} & \text{(T-}\textsc{Suspend}\text{)} \\
\dfrac{\Gamma \vdash e : \texttt{\textbf{fut}}\langle T\rangle}{\Gamma \vdash e? : \mathsf{Bool}} &
\dfrac{\Gamma \vdash x : \texttt{\textbf{fut}}\langle T\rangle}{\Gamma \vdash x.\texttt{\textbf{get}} : T} &
\dfrac{}{\Gamma \vdash \texttt{\textbf{skip}}} &
\dfrac{\Gamma \vdash g : \mathsf{Bool}}{\Gamma \vdash \texttt{\textbf{await}}\ g} &
\dfrac{}{\Gamma \vdash \texttt{\textbf{suspend}}}
\end{array}
$$

$$
\begin{array}{cccc}
\text{(T-}\textsc{Composition}\text{)} & \text{(T-}\textsc{Assign}\text{)} & \text{(T-}\textsc{And}\text{)} & \text{(T-}\textsc{New}\text{)} \\
\dfrac{\Gamma \vdash s \quad \Gamma \vdash s'}{\Gamma \vdash s; s'} &
\dfrac{\Gamma \vdash rhs : \Gamma(v)}{\Gamma \vdash v = rhs} &
\dfrac{\Gamma \vdash g_1 : \mathsf{Bool} \\ \Gamma \vdash g_2 : \mathsf{Bool}}{\Gamma \vdash g_1 \wedge g_2 : \mathsf{Bool}} &
\dfrac{\Gamma \vdash \overline{e} : \mathrm{ptypes}(C) \\ T \in \mathrm{interfaces}(C)}{\Gamma \vdash \texttt{\textbf{new}}\ [\texttt{\textbf{cog}}]\ C(\overline{e}) : T}
\end{array}
$$

$$
\begin{array}{ccc}
\text{(T-}\textsc{AsyncCall}\text{)} & \text{(T-}\textsc{Conditional}\text{)} & \text{(T-}\textsc{While}\text{)} \\
\dfrac{\Gamma \vdash e.m(\overline{e}) : T}{\Gamma \vdash e!m(\overline{e}) : \texttt{\textbf{fut}}\langle T\rangle} &
\dfrac{\Gamma \vdash b : \mathsf{Bool} \quad \Gamma \vdash s_1 \quad \Gamma \vdash s_2}{\Gamma \vdash \texttt{\textbf{if}}\ b\,\{s_1\}\,\texttt{\textbf{else}}\,\{s_2\}} &
\dfrac{\Gamma \vdash b : \mathsf{Bool} \quad \Gamma \vdash s}{\Gamma \vdash \texttt{\textbf{while}}\ b\,\{s\}}
\end{array}
$$

$$
\begin{array}{ccc}
\text{(T-}\textsc{Return}\text{)} & \text{(T-}\textsc{SyncCall}\text{)} & \text{(T-}\textsc{Method}\text{)} \\
\dfrac{\Gamma \vdash e : T \\ \Gamma(\mathrm{destiny}) = \texttt{\textbf{fut}}\langle T\rangle}{\Gamma \vdash \texttt{\textbf{return}}\ e} &
\dfrac{\Gamma \vdash e : N \quad \Gamma \vdash \overline{e} : \overline{T} \\ \mathrm{match}(m, \overline{T} \to T, N)}{\Gamma \vdash e.m(\overline{e}) : T} &
\dfrac{\Gamma' = \Gamma[\overline{x} \mapsto \overline{T}, \overline{x'} \mapsto \overline{T'}] \\ \Gamma'[\mathrm{destiny} \mapsto \texttt{\textbf{fut}}\langle T''\rangle] \vdash s}{\Gamma \vdash T''\ m\ (\overline{T}\ \overline{x})\{\overline{T'}\ \overline{x'}; s\}}
\end{array}
$$

$$
\begin{array}{cc}
\text{(T-}\textsc{Class}\text{)} & \text{(T-}\textsc{Program}\text{)} \\
\dfrac{\forall I \in \overline{I} \cdot \mathrm{implements}(C, I) \\ \Gamma[\mathrm{this} \mapsto C, \mathrm{fields}(C)] \vdash \overline{M}}{\Gamma \vdash \texttt{\textbf{class}}\ C\ \texttt{\textbf{implements}}\ \overline{I}\ \{\overline{T}\ \overline{f}; \overline{M}\}} &
\dfrac{\Gamma[\overline{x} \mapsto \overline{T}] \vdash s \qquad \forall Dd \in \overline{Dd} \cdot \Gamma \vdash Dd \\ \forall CL \in \overline{CL} \cdot \Gamma \vdash CL \quad \forall F \in \overline{F} \cdot \Gamma \vdash F}{\Gamma \vdash \overline{Dd}\ \overline{F}\ \overline{IF}\ \overline{CL}\ \{\overline{T}\ \overline{x}; s\}}
\end{array}
$$

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

over p. If $p = x$, $psubst(p, A, \Gamma) =$ if $\Gamma(x) = T$ then ε else $[p \mapsto A]$ fi. If $p = t$ or $p = \_$, $psubst(p, A, \Gamma) = \varepsilon$. Otherwise $p = Co(p_1, \ldots, p_n)$ such that $\Gamma(Co) = A_1, \ldots, A_n \to A$, and $psubst(p, A, \Gamma) = psubst(p_1, A_1) \circ \ldots \circ psubst(p_n, A_n, \Gamma)$. The type of a variable x in p must be the same as in $\Gamma(x)$, unless it is new.

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; object identifiers are typed 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 ABS Type System is given in Fig. 4. By T-Program, a program is well-typed if its data types, functions, interfaces, classes, and its main block are well-typed (we ignore the straightforward type checking of interface declarations). 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. We add a fresh name *this* to the typing context, which is typed by C, allowing 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. By T-Method, a method declaration is well-typed if its body is well-typed in the typing context extended by the typing of formal parameters and local variables. We add a fresh name *destiny* to the typing context, which binds to the type of the method's future. The rules for **skip**, **suspend**, assignment, composition, conditional, and **while** are standard.

By T-Return, a **return** statement from an asynchronous method call is well-typed if its expression types to the type of the method's future. By T-Await,

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

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

await g is well-typed if g is of type Bool, rule T-And decomposes guards, and by T-Poll a reply-guard $e?$ is a Bool if e 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 $\mathbf{fut}\langle T\rangle$ 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 $\overline{T} \rightarrow T$ matches a signature for m in the known interface of the callee (given by an auxiliary function *match*).

5 An Operational Semantics for Core ABS

The operational semantics of ABS is presented as a transition system in an SOS style [33]. 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 [30]). A run is a possibly nonterminating sequence of rule applications. When auxiliary functions are used in the semantics, these are evaluated in between the applications of transition rules in a run.

5.1 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. Configurations are written 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 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 $\{l \mid s\}$ when convenient or it is *idle.* The statement $\mathbf{cont}(f)$ is used to control scheduling when local synchronous calls complete their execution, returning control to the caller. In an *invocation message* $invoc(o, f, m, \overline{v})$, o is the callee, f the future to

$$\textsc{(RedCons)}\quad \dfrac{\sigma \vdash e_i \rightsquigarrow \sigma' \vdash e_i' \quad 1 \leq i \leq n}{\begin{array}{c}\sigma \vdash Co(e_1, \ldots, e_i, \ldots, e_n)\\ \rightsquigarrow \sigma' \vdash Co(e_1, \ldots, e_i', \ldots, e_n)\end{array}} \qquad \textsc{(RedFunExp)}\quad \dfrac{\sigma \vdash e_i \rightsquigarrow \sigma' \vdash e_i' \quad 1 \leq i \leq n}{\begin{array}{c}\sigma \vdash fn(e_1, \ldots, e_i, \ldots, e_n)\\ \rightsquigarrow \sigma' \vdash fn(e_1, \ldots, e_i', \ldots, e_n)\end{array}}$$

$$\textsc{(RedVar)}\quad \sigma \vdash x \rightsquigarrow \sigma \vdash \sigma(x) \qquad \textsc{(RedCase1)}\quad \dfrac{\sigma \vdash e \rightsquigarrow \sigma' \vdash e'}{\begin{array}{c}\sigma \vdash \textbf{case}\ e\ \{\overline{br}\}\\ \rightsquigarrow \sigma' \vdash \textbf{case}\ e'\ \{\overline{br}\}\end{array}} \qquad \textsc{(RedCase3)}\quad \dfrac{\mathrm{match}(\sigma(p), t) = \bot}{\begin{array}{c}\sigma \vdash \textbf{case}\ t\ \{p \Rightarrow e; \overline{br}\}\\ \rightsquigarrow \sigma \vdash \textbf{case}\ t\ \{\overline{br}\}\end{array}}$$

$$\textsc{(RedFunGround)}\quad \dfrac{\begin{array}{c}\mathrm{fresh}(\{y_1, \ldots, y_n\})\\ \overline{y} = y_1, \ldots, y_n\\ \mathrm{length}(\overline{x}_{fn}) = n\end{array}}{\begin{array}{c}\sigma \vdash fn(\overline{t})\\ \rightsquigarrow \sigma[\overline{y} \mapsto \overline{t}] \vdash e_{fn}[\overline{x}_{fn} \mapsto \overline{y}]\end{array}} \qquad \textsc{(RedCase2)}\quad \dfrac{\begin{array}{c}\mathrm{match}(\sigma(p), t) \neq \bot\\ \sigma' = \sigma[y_i \mapsto \mathrm{match}(\sigma(p), t)(x_i)] \text{ for } 1 \leq i \leq n\\ \overline{y} = y_1, \ldots, y_n \qquad \mathrm{fresh}(\{y_1, \ldots, y_n\})\\ \overline{x} = x_1, \ldots, x_n \qquad \{x_1, \ldots, x_n\} = \mathrm{vars}(\sigma(p))\end{array}}{\sigma \vdash \textbf{case}\ t\ \{p \Rightarrow e; \overline{br}\} \rightsquigarrow \sigma' \vdash e[\overline{x} \mapsto \overline{y}]}$$

Fig. 6. The evaluation of functional expressions.

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 $\bot$ when the future's reply value has not been received). Values are object and future identifiers, Boolean values, and ground terms from the functional language. For simplicity, classes are not represented explicitly in the semantics, as they may be seen as static tables.

5.2 A Reduction System for ABS Functional Expressions

The evaluation of functional expressions is defined by the small-step reduction relation $\sigma \vdash e \rightsquigarrow \sigma' \vdash e'$, given in Fig. 6, which reduces an expression e in the context of a *substitution* σ to e' in the context of σ'. A substitution σ is *well-typed* in a typing context Γ, denoted $\Gamma \vdash \sigma$, if $\Gamma \vdash \sigma(x) : \Gamma(x)$ for all $x \in dom(\sigma)$.

Let $e[\overline{x} \mapsto \overline{y}]$ denote the expression e in which every occurrence of x_i has been replaced by the corresponding y_i. The predicate $fresh(\{x_1, \ldots, x_j\})$ asserts that any variable name x_i (for $1 \leq i \leq j$) is globally unique. Let the syntactic category t consist of ground terms, i.e., constructor terms with only ground terms in argument positions, built-in constants such as **`null`**, and object names.

Function evaluation is strict. For a function fn defined by **`def`** $T\ fn(\overline{T}\ \overline{x}) = e$, denote by $\overline{x}_{fn}$ the formal parameter list $\overline{x}$ and by e_{fn} the body e. The evaluation of a function call $fn(\overline{e})$ in a context σ reduces to the evaluation of $e_{fn}[\overline{x}_{fn} \mapsto \overline{y}]$ in the context $\sigma[\overline{y} \mapsto \overline{t}]$ after the arguments $\overline{e}$ have successfully been reduced to ground terms $\overline{t}$. The change in scope for evaluating a function body is obtained by replacing the formal parameters $\overline{x}_{fn}$ with *fresh variables* $\overline{y}$ in the function body, thus avoiding name capture while keeping the full context σ.

Case expressions will only be reduced if the pattern in one of the branches matches. The function $match(p, t)$ returns the unique substitution σ such that

$\sigma(p) = t$ and $dom(\sigma) = vars(p)$ (otherwise, $match(p,t) = \bot$). Note how in RedCase2, the current substitution σ is applied to the pattern before matching, which allows the pattern to first match with the current state. For pattern matching, variables in the pattern p are bound to ground terms in the term t, applying a similar variable renaming as for function evaluation. Thus the context for evaluating the right-hand side e of the branch $p \Rightarrow e$ extends the current substitution σ with the bindings that occurred during the pattern matching.

The variable renaming in the rules which change the scope of variables, i.e., RedFunGround and RedCase2, allows small-step reductions in the arguments to constructors and function application in RedCons and RedFunExp, since the additional variables will not introduce name conflicts.

Lemma 1 (Type preservation). *Let Γ be a typing context and σ a substitution such that $\Gamma \vdash \sigma$. If $\Gamma \vdash e : A$ and $\sigma \vdash e \rightsquigarrow \sigma' \vdash e'$, then there is a typing context Γ' such that $\Gamma \subseteq \Gamma'$, $\Gamma' \vdash \sigma'$, and $\Gamma' \vdash e' : A$.*

Proof. The proof is by induction over the application of reduction rules.

- RedVar. By assumption, $\Gamma \vdash \sigma$ and $\Gamma \vdash x : A$. Since σ is well-typed, $\Gamma \vdash \sigma(x) : \Gamma(x)$, so $\Gamma \vdash \sigma(x) : A$.
- RedCons. By the induction hypothesis (IH), $\Gamma \vdash e_i : A_i$, $\Gamma' \vdash e_i' : A_i$, $\Gamma \subseteq \Gamma'$, and $\Gamma' \vdash \sigma'$. By assumption, $\Gamma \vdash Co(e_1, \ldots, e_i, \ldots, e_n) : A$. Since $\Gamma \subseteq \Gamma'$, $\Gamma' \vdash Co(e_1, \ldots, e_i', \ldots, e_n) : A$.
- RedFunExp. Similar to the case for RedCons.
- RedFunGround. By assumption, $\Gamma \vdash \sigma$, $\Gamma \vdash fn(\overline{t}) : A$, and $\Gamma \vdash t_i : A_i$ for all t_i in $\overline{t}$. Thus, we may assume a function declaration for fn such that $\Gamma(fn) = \overline{T} \to T'$ and a type substitution ρ such that $T'\rho = A$ and $T_i\rho = A_i$ for all T_i in $\overline{T}$. Obviously, $\Gamma[\overline{x}_{fn} \mapsto \overline{T\rho}] \vdash [\overline{x}_{fn} \mapsto \overline{T}]$. By T-FuncDecl, $\Gamma[\overline{x}_{fn} \mapsto \overline{T}] \vdash e_{fn} : T'$. Typing is preserved under type substitutions [32], so $\Gamma[\overline{x}_{fn} \mapsto \overline{T\rho}] \vdash e_{fn} : T'\rho$, which is the same as $\Gamma[\overline{x}_{fn} \mapsto \overline{A_i}] \vdash e_{fn} : A$. After variable renaming, we let $\Gamma' = \Gamma[\overline{y} \mapsto \overline{A_i}]$ and get $\Gamma' \vdash e_{fn}[\overline{x}_{fn} \mapsto \overline{y}] : A$. Since $\{y_1, \ldots, y_n\}$ are fresh, we have $\Gamma \subseteq \Gamma'$ and $\Gamma' \vdash \sigma$, so $\Gamma' \vdash \sigma'$.
- RedCase1. By assumption $\Gamma \vdash e : T$, and by the IH $\Gamma \subseteq \Gamma'$,$\Gamma' \vdash \sigma'$, and $\Gamma' \vdash e' : T$. Since $\Gamma \subseteq \Gamma'$, $\Gamma' \vdash$ **case** e' $\{\overline{br}\}$.
- RedCase2. By assumption, $\Gamma \vdash \sigma$, $\Gamma \vdash$ **case** $t \, \{p \Rightarrow e; \overline{br}\} : A$, and $match(\sigma(p), t) \neq \bot$. Since we match with $\sigma(p)$, $vars(\sigma(p)) \cap dom(\sigma) = \emptyset$. By T-Case, there is some type T such that $\Gamma \vdash t : T$ and $\Gamma \vdash p \Rightarrow e : T \to A$. By T-Branch, there is a type substitution $\rho = psubst(\sigma(p), T) \neq \bot$ such that for $\Gamma'' = \Gamma \circ \rho$, $\Gamma'' \vdash \sigma(p) : T$ and $\Gamma'' \vdash e : A$. Since $dom(\rho) \cap dom(\sigma) = \emptyset$, $\Gamma'' \vdash \sigma \circ match(\sigma(p), t)$. Renaming the variables in $\sigma(p)$, we define $\Gamma' = \Gamma[y_i \mapsto \Gamma''(x_i)]$ for $1 \leq i \leq n$. Obviously, $\Gamma \subseteq \Gamma'$. Since $y_1, \ldots, y_n$ are fresh, renaming variables uniformly does not change the derivations, so we get $\Gamma' \vdash \sigma'$ and $\Gamma'' \vdash e[\overline{x} \mapsto \overline{y}] : A$.
- RedCase3. Since $\Gamma \vdash$ **case** $t \, \{p \Rightarrow e; \overline{br}\} : A$, we have $\Gamma \vdash$ **case** $t \, \{\overline{br}\} : A$. □

It follows from Lemma 1 that given a well-typed expression e and a well-typed context σ, then all states in the reduction sequence from $\sigma \vdash e$ will be well-typed,

$$
\begin{array}{cccc}
(\textsc{RedBoolGuard}) & (\textsc{RedReplyGuard1}) & (\textsc{RedReplyGuard2}) & (\textsc{RedGuards}) \\
\dfrac{\begin{array}{c}\sigma \vdash b \\ \leadsto \sigma \vdash b'\end{array}}{\begin{array}{c}\sigma, cn \vdash b \\ \leadsto \sigma, cn \vdash b'\end{array}} &
\dfrac{\begin{array}{c}\sigma \vdash e \leadsto \sigma \vdash f \\ \mathit{fut}(f, v) \in cn \quad v \neq \bot\end{array}}{\begin{array}{c}\sigma, cn \vdash e? \\ \leadsto \sigma, cn \vdash \text{true}\end{array}} &
\dfrac{\begin{array}{c}\sigma \vdash e \leadsto \sigma \vdash f \\ \mathit{fut}(f, \bot) \in cn\end{array}}{\begin{array}{c}\sigma, cn \vdash e? \\ \leadsto \sigma, cn \vdash \text{false}\end{array}} &
\dfrac{\begin{array}{c}\sigma, cn \vdash g_1 \leadsto \sigma, cn \vdash g'_1 \\ \sigma, cn \vdash g_2 \leadsto \sigma, cn \vdash g'_2\end{array}}{\begin{array}{c}\sigma, cn \vdash g_1 \wedge g_2 \\ \leadsto \sigma, cn \vdash g'_1 \wedge g'_2\end{array}}
\end{array}
$$

Fig. 7. The evaluation of guard expressions.

independent of the order of reductions. If an expression e in a context σ reduces to a ground term t, we denote the resulting value by $[\![e]\!]_\sigma$. This value, however, is not guaranteed to exist, for two reasons: first, the reduction sequence might not terminate; second, the normal form may not be a ground term, because branches in case expressions need not have full coverage. (We assume here that normal forms are unique, although we do not prove this.) In either case, we know by Lemma 1 that all states in such a reduction sequence are well-typed.

5.3 The Operational Semantics for Concurrent Objects in ABS

Evaluating Guards. Given a substitution σ and a configuration cn, we lift the reduction relation for functional expressions to guards by the rules of Fig. 7. It follows from Lemma 1 that well-typedness is preserved by guard reduction in the context of well-typed substitutions and configurations. If a guard g in a context σ, cn reduces to a ground term b, we denote the resulting value by $[\![g]\!]^{cn}_\sigma$.

Auxiliary functions. If T is the return type of a method m in a class C, we let $\mathit{bind}(o, f, m, \overline{v}, C)$ 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 formal parameters are bound to $\overline{v}$. If binding does *not* succeed, we get the **`error`** process. The function $\mathit{atts}(C, \overline{v}, o, c)$ 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 and cog identities o and c, respectively. The function $\mathit{init}(C)$ returns an activation of the *init* method of C, if defined, and otherwise the *idle* process. The predicate $\mathit{fresh}(n)$ asserts that a name n is globally unique (where n may be an identifier for an object, a future, or a cog).

Transition rules transform state configurations into new configurations, and are given in Figs. 8 and 9. There are different assignment rules for functional expressions (Assign-Local and Assign-Field), 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 Assign-Local and Assign-Field assign the value of expression e to a variable x in the local variables l or in the fields a, respectively. Rules Cond-True and Cond-False 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, $q \backslash p$ removes p from q, and $\mathit{select}(q, a, cn)$ selects

$$\textsc{(Skip)}\quad \begin{array}{c} ob(o, a, \{l|\texttt{skip}; s\}, q) \\ \to ob(o, a, \{l|s\}, q) \end{array}$$

$$\textsc{(Assign-Local)}\quad \frac{x \in \mathrm{dom}(l) \quad v = \llbracket e \rrbracket_{(a \circ l)}}{\begin{array}{c} ob(o, a, \{l|x = e; s\}, q) \\ \to ob(o, a, \{l[x \mapsto v]|s\}, q) \end{array}}$$

$$\textsc{(Assign-Field)}\quad \frac{x \in \mathrm{dom}(a) \quad v = \llbracket e \rrbracket_{(a \circ l)}}{\begin{array}{c} ob(o, a, \{l|x = e; s\}, q) \\ \to ob(o, a[x \mapsto v], \{l|s\}, q) \end{array}}$$

$$\textsc{(Cond-True)}\quad \frac{\llbracket e \rrbracket_{(a \circ l)}}{\begin{array}{c} ob(o, a, \{l|\texttt{if}\ e\ \texttt{then}\ \{s_1\}\ \texttt{else}\ \{s_2\}; s\}, q) \\ \to ob(o, a, \{l|s_1; s\}, q) \end{array}}$$

$$\textsc{(Cond-False)}\quad \frac{\neg\llbracket e \rrbracket_{(a \circ l)}}{\begin{array}{c} ob(o, a, \{l|\texttt{if}\ e\ \texttt{then}\ \{s_1\}\ \texttt{else}\ \{s_2\}; s\}, q) \\ \to ob(o, a, \{l|s_2; s\}, q) \end{array}}$$

$$\textsc{(Suspend)}\quad \begin{array}{c} ob(o, a, \{l|\texttt{suspend}; s\}, q) \\ \to ob(o, a, \mathrm{idle}, q \cup \{l|s\}) \end{array}$$

$$\textsc{(Release-Cog)}\quad \frac{c = a(\mathrm{cog})}{\begin{array}{c} ob(o, a, \mathrm{idle}, q)\ \ cog(c, o) \\ \to ob(o, a, \mathrm{idle}, q)\ \ cog(c, \epsilon) \end{array}}$$

$$\textsc{(Await-True)}\quad \frac{\llbracket g \rrbracket^{cn}_{(a \circ l)}}{\begin{array}{c} \{ob(o, a, \{l|\texttt{await}\ g; s\}, q)\ \ cn\} \\ \to \{ob(o, a, \{l|s\}, q)\ \ cn\} \end{array}}$$

$$\textsc{(Activate)}\quad \frac{p = \mathrm{select}(q, a, cn) \quad c = a(\mathrm{cog})}{\begin{array}{c} \{ob(o, a, \mathrm{idle}, q)\ \ cog(c, \epsilon)\ \ cn\} \\ \to \{ob(o, a, p, q \backslash p)\ \ cog(c, o)\ \ cn\} \end{array}}$$

$$\textsc{(Await-False)}\quad \frac{\neg\llbracket g \rrbracket^{cn}_{(a \circ l)}}{\begin{array}{c} \{ob(o, a, \{l|\texttt{await}\ g; s\}, q)\ \ cn\} \\ \to \{ob(o, a, \{l|\texttt{suspend}; \texttt{await}\ g; s\}, q)\ \ cn\} \end{array}}$$

$$\textsc{(Async-Call)}\quad \frac{o' = \llbracket e \rrbracket_{(a \circ l)} \quad \overline{v} = \llbracket \overline{e} \rrbracket_{(a \circ l)} \quad \mathrm{fresh}(f)}{\begin{array}{c} ob(o, a, \{l|x = e!m(\overline{e}); s\}, q) \\ \to ob(o, a, \{l|x = f; s\}, q)\ \ invoc(o', f, m, \overline{v})\ \ fut(f, \bot) \end{array}}$$

$$\textsc{(Bind-Mtd)}\quad \frac{p' = \mathrm{bind}(o, f, m, \overline{v}, \mathrm{class}(o))}{\begin{array}{c} ob(o, a, p, q)\ \ invoc(o, f, m, \overline{v}) \\ \to ob(o, a, p, q \cup p') \end{array}}$$

Fig. 8. Semantics of the concurrent object level of Core ABS (1).

a process from q (if q is empty or no process is *ready*, the result is the idle process [24]). The actual definitions of these operations are left unspecified; 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 states).

Let $\emptyset$ denote the empty pool. Rule Suspend suspends the active process to the process pool, leaving the processor idle, and Release-Cog makes the cog idle if its active object is idle. Rule Await-True consumes **await** g if g evaluates to true in the object's current state, Await-False adds a **suspend** statement to the process if the guard evaluates to false. Rule Activate selects a process p from the process pool for execution if p is *ready* to execute, i.e., if p would not directly be resuspended or block the processor [24]. These rules ensure that a process can only be scheduled if the cog associated with the object is idle, and that an object always acquires the cog if its process is activated. 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 a new future f (which is unique since fresh(f)), the method name m, and actual parameters $\overline{v}$. The return value of f is undefined (i.e., $\bot$). Rule Bind-Mtd consumes an invocation message and places the process corresponding to the method activation in the callee's process pool. A reserved local variable *destiny* stores the identity of the future associated with the call.

Rule Return in Fig. 9 places the return value into the call's associated future. Rule Read-Fut dereferences the future f if $v \neq \bot$. If $v = \bot$, the reduction

$$\text{(Return)}\quad \frac{v = \llbracket e \rrbracket_{(a\circ l)} \quad l(\text{destiny}) = f}{\begin{array}{c} ob(o, a, \{l|\textbf{return } e; s\}, q)\ fut(f, \bot) \\ \to ob(o, a, \{l|s\}, q)\ fut(f, v) \end{array}}$$

$$\text{(Read-Fut)}\quad \frac{v \neq \bot \quad f = \llbracket e \rrbracket_{(a\circ l)}}{\begin{array}{c} ob(o, a, \{l|x = e.\textbf{get}; s\}, q)\ fut(f, v) \\ \to ob(o, a, \{l|x = v; s\}, q)\ fut(f, v) \end{array}}$$

$$\text{(Cog-Sync-Call)}\quad \frac{\begin{array}{c} o' = \llbracket e \rrbracket_{(a\circ l)} \quad \overline{v} = \llbracket \overline{e} \rrbracket_{(a\circ l)} \quad \text{fresh}(f) \\ a'(\text{cog}) = c \quad f' = l(\text{destiny}) \\ \{l'|s'\} = \text{bind}(o', f, m, \overline{v}, class(o')) \end{array}}{\begin{array}{c} ob(o, a, \{l|x = e.m(\overline{e}); s\}, q) \\ ob(o', a', \text{idle}, q')\ cog(c, o) \\ \to ob(o, a, \text{idle}, q \cup \{l|x = f.\textbf{get}; s\})\ fut(f, \bot) \\ ob(o', a', \{l'|s'; \textbf{cont}(f')\}, q')\ cog(c, o') \end{array}}$$

$$\text{(Cog-Sync-Return-Sched)}\quad \frac{a'(\text{cog}) = c \quad l'(\text{destiny}) = f}{\begin{array}{c} ob(o, a, \{l|\textbf{cont}(f)\}, q)\ cog(c, o) \\ ob(o', a', \text{idle}, q' \cup \{l'|s\}) \\ \to ob(o, a, \text{idle}, q)\ cog(c, o') \\ ob(o', a', \{l'|s\}, q') \end{array}}$$

$$\text{(Self-Sync-Call)}\quad \frac{\begin{array}{c} f' = l(\text{destiny}) \quad o = \llbracket e \rrbracket_{(a\circ l)} \quad \overline{v} = \llbracket \overline{e} \rrbracket_{(a\circ l)} \\ \text{fresh}(f) \quad \{l'|s'\} = \text{bind}(o, f, m, \overline{v}, class(o)) \end{array}}{\begin{array}{c} ob(o, a, \{l|x = e.m(\overline{e}); s\}, q) \\ \to ob(o, a, \{l'|s'; \textbf{cont}(f')\}, q \cup \{l|x = f.\textbf{get}; s\})\ fut(f, \bot) \end{array}}$$

$$\text{(Self-Sync-Return-Sched)}\quad \frac{l'(\text{destiny}) = f}{\begin{array}{c} ob(o, a, \{l|\textbf{cont}(f)\}, q \cup \{l'|s\}) \\ \to ob(o, a, \{l'|s\}, q) \end{array}}$$

$$\text{(Rem-Sync-Call)}\quad \frac{o' = \llbracket e \rrbracket_{(a\circ l)} \quad \text{fresh}(f) \quad a(\text{cog}) \neq a'(\text{cog})}{\begin{array}{c} ob(o, a, \{l|x = e.m(\overline{e}); s\}, q)\ ob(o', a', p, q') \\ \to ob(o, a, \{l|f = e!m(\overline{e}); x = f.\textbf{get}; s\}, q) \\ ob(o', a', p, q') \end{array}}$$

$$\text{(New-Object)}\quad \frac{\begin{array}{c} \text{fresh}(o')\ p = \text{init}(C) \\ a' = \text{atts}(C, \llbracket \overline{e} \rrbracket_{(a\circ l)}, o', c) \end{array}}{\begin{array}{c} ob(o, a, \{l|x = \textbf{new } C(\overline{e}); s\}, q)\ cog(c, o) \\ \to ob(o, a, \{l|x = o'; s\}, q)\ cog(c, o) \\ ob(o', a', \text{idle}, \{p\}) \end{array}}$$

$$\text{(New-Cog-Object)}\quad \frac{\begin{array}{c} \text{fresh}(o')\ \text{fresh}(c')\ p = \text{init}(C) \\ a' = \text{atts}(C, \llbracket \overline{e} \rrbracket_{(a\circ l)}, o', c') \end{array}}{\begin{array}{c} ob(o, a, \{l|x = \textbf{new cog } C(\overline{e}); s\}, q) \\ \to ob(o, a, \{l|x = o'; s\}, q) \\ ob(o', a', p, \emptyset) \quad cog(c', o') \end{array}}$$

Fig. 9. Semantics of the concurrent object level of Core ABS (2).

on this object is *blocked.* Rules Cog-Sync-Call and Cog-Sync-Return-Sched address synchronous method calls between two objects that are in the same cog. For a synchronous call, possession of the cog directly transfers control from the calling object to 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 Cog-Sync-Call, which is then used to re-activate the caller process in 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. A synchronous call to an object of another cog is syntactic sugar for an asynchronous call which is immediately followed by a blocking **get** operation, captured by Rem-Sync-Call.

Rule New-Object creates an object with a unique identifier o'. The object's fields are given default values by $atts(C, \llbracket \overline{e} \rrbracket_{(a\circ l)}, o', c)$, 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. To instantiate the remaining fields, the process p is queued (we assume that this process reduces to idle if

$$\textsc{(T-State1)}\ \frac{\Delta(v) = T \quad \Delta \vdash_R \mathit{val} : T}{\Delta \vdash_R T\ v\ \mathit{val}\ \mathbf{ok}} \qquad \textsc{(T-Cont)}\ \frac{\Delta(f) = \mathsf{fut}\langle T\rangle}{\Delta \vdash \mathbf{cont}(f)} \qquad \textsc{(T-Future)}\ \frac{\Delta(f) = \mathsf{fut}\langle T\rangle \quad \mathit{val} \neq \bot \Rightarrow \Delta(\mathit{val}) = T}{\Delta \vdash_R \mathit{fut}(f, \mathit{val})\ \mathbf{ok}} \qquad \textsc{(T-Configurations)}\ \frac{\Delta \vdash_R \mathit{cn}\ \mathbf{ok} \quad \Delta \vdash_R \mathit{cn}'\ \mathbf{ok}}{\Delta \vdash_R \mathit{cn}\ \mathit{cn}'\ \mathbf{ok}}$$

$$\textsc{(T-State2)}\ \frac{\Delta \vdash_R \mathit{fds}\ \mathbf{ok} \quad \Delta \vdash_R \mathit{fds}'\ \mathbf{ok}}{\Delta \vdash_R \mathit{fds}\ \mathit{fds}'\ \mathbf{ok}} \qquad \textsc{(T-Process-Queue)}\ \frac{\Delta \vdash_R q\ \mathbf{ok} \quad \Delta \vdash_R q'\ \mathbf{ok}}{\Delta \vdash_R q\ q'\ \mathbf{ok}} \qquad \textsc{(T-Process)}\ \frac{\Delta' \vdash_R \overline{T\ x\ \mathit{val}}\ \mathbf{ok} \quad \Delta' = \Delta[\overline{x} \to \overline{T}] \quad \Delta' \vdash_R s\ \mathbf{ok}}{\Delta \vdash_R (\overline{T\ x\ \mathit{val}}, s)\ \mathbf{ok}} \qquad \textsc{(T-Cog)}\ \frac{\Delta(c) = \mathit{cog}}{\Delta \vdash_R \mathit{cog}(c, \mathrm{act})}$$

$$\textsc{(T-Empty)}\ \Delta \vdash_R \epsilon\ \mathbf{ok} \qquad \textsc{(T-Idle)}\ \Delta \vdash_R \mathbf{idle}\ \mathbf{ok}$$

$$\textsc{(T-Object)}\ \frac{\mathrm{fields}(\Delta(o)) = [\overline{x} \mapsto \overline{T}] \quad \Delta' = \Delta[\overline{x} \mapsto \overline{T}] \quad \Delta' \vdash_R q\ \mathbf{ok} \quad \Delta' \vdash_R \overline{T\ x\ \mathit{val}}\ \mathbf{ok} \quad \Delta' \vdash_R p\ \mathbf{ok}}{\Delta \vdash_R \mathit{ob}(o, \overline{T\ x\ \mathit{val}}, p, q)\ \mathbf{ok}} \qquad \textsc{(T-Invoc)}\ \frac{\Delta(f) = \mathsf{fut}\langle T\rangle \quad \Delta(\overline{v}) = \overline{T} \quad \mathrm{match}(m, \overline{T} \to T, \Delta(o))}{\Delta \vdash_R \mathit{invoc}(o, f, m, \overline{v})}$$

Fig. 10. The typing rules for runtime configurations.

init(C) is unspecified in the class definition, and that it asynchronously calls `run` if the latter is given). Process p is not directly scheduled in order to uphold the cog invariant (ie., only one object per cog is active), hence any scheduling policy must take care to always schedule an initial process p with highest priority. Rule New-Cog-Object is like 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 $ob(\mathrm{start}, \varepsilon, p, \emptyset)$, where the process p corresponds to the activation of the program's main block. 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, substitutions remain well-typed and method binding does not result in the **error** process.

Runtime Configurations. Typing rules are given for the runtime syntax shown 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 configuration *config* is well-typed in the typing context Δ. The typing rules for runtime configurations are given in Fig. 10. In rule T-Object, the premise $\mathit{fields}(\Delta(o)) = [\overline{x} \mapsto \overline{T}]$ asserts that the object's fields 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, expression and guard evaluation in ABS processes preserves typing.

Well-typedness Assumptions for Auxiliary Functions. Let C be a class with formal parameters $\overline{x}$ of types $\overline{T}$. We assume that *init*(C) returns a well-typed process, and $\mathit{atts}(C, \overline{v}, o, c)$ 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 $\overline{T'}$, let $bind(o, f, m, \overline{v'}, C)$ return 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 block 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 $\Gamma \vdash_R ob(start, \varepsilon, \{\overline{T}\ \overline{x}\ atts(\overline{T})|s\}, \emptyset)$ **ok**.*

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

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

Proof. The proof is by induction over the application of transition rules. We assume that class definitions are well-typed (and omit them from the runtime syntax since they do not change). Objects, futures, and messages not affected by a transition remain well-typed, and are ignored below. It follows from Lemma 1 that the reduction of an expression in a well-typed object results in a well-typed object. The transition rules apply when these reductions terminate, reducing an expression e in the state σ to the ground term $[\![e]\!]_\sigma$. Hence, the reduction of expressions e in states σ occur as $[\![e]\!]_\sigma$ in the assumptions to the transition rules, and similarly for the evaluation $[\![g]\!]^{cn}_\sigma$ of guards g in a configuration cn.

- Skip. If $\Delta \vdash_R ob(o, a, \{l|\mathbf{skip}; s\}, q)$ **ok**, then $\Delta \vdash_R ob(o, a, \{l|s\}, q)$ **ok**.
- *Assignment.* Let $\Delta \vdash_R ob(o, \overline{T}_1\ \overline{x}_1\ \overline{v}_1, \{\overline{T}_2\ \overline{x}_2\ \overline{v}_2|x = e; s\}, q)$ **ok**. Let $\Delta' = \Delta[\overline{x}_1 \mapsto \overline{T}_1, \overline{x}_2 \mapsto \overline{T}_2]$. Then $\Delta' \vdash x = e; s$, so $\Delta' \vdash e : \Delta'(x)$. Assume that $v = [\![e]\!]_{a\circ l}$. For Assign-Local, we need to show $\Delta \vdash_R ob(o, \overline{T}_1\ \overline{x}_1\ \overline{v}_1,$ $\{\overline{T}_2\ \overline{x}_2\ \overline{v}_2[x \mapsto v]|x = e; s\}, q)$ **ok**, which follows from Lemma 1 since $\Delta' \vdash v : \Delta'(x)$. Similarly for Assign-Field.
- *Conditionals.* Let $\Delta \vdash_R ob(o, a, \{l|\mathbf{if}\ e\ \{s_1\}\ \mathbf{else}\ \{s_s\}; s\}, q)$ **ok**. By assumption there is a Δ' extending Δ with a and l, such that $\Delta' \vdash e$, $\Delta' \vdash s_1$, $\Delta' \vdash s_2$, and $\Delta' \vdash s$. Consequently, $\Delta' \vdash s_1; s$ and $\Delta' \vdash s_2; s$, and both Cond-True and Cond-False preserve well-typedness.
- *Process Suspension and Activation.* It is immediate that the rules Await-True, Await-False, Activate, Suspend, Release-Cog, Self-Sync-Return-Sched, and Cog-Sync-Return-Sched preserve the well-typedness.
- *Object creation.* For New-Object, assume $\Delta \vdash_R ob(o, a, \{l|x = \mathbf{new}\ C(\overline{e}); s\},$ $q)$ **ok**, $\Delta(x) = I$, and $implements(C, I)$ (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**. For New-Cog-Object, let $\Delta' = \Delta[o' \mapsto C, c' \mapsto cog]$. Since $fresh(c')$, this does not affect the well-typedness of o and o'. We must additionally show that $\Delta' \vdash_R cog(c', o')$ **ok**, which is immediate.
- AsyncCall. Let $\Delta \vdash_R ob(o, a, \{l|x = e!m(\overline{e}\}, q)$ **ok**. We first consider the case $e \neq \mathbf{this}$. 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 $\mathit{match}(m, \overline{T} \to T, I)$. Assume that $[\![e]\!]_{a\circ l} = o'$ and let $\Delta(o') = C$ for some class C. By Lemma 1, there is a Δ' such that $\Delta' \vdash_R [\![e]\!]_{a\circ l} : I$ and $\Delta'(o') = C$, so $C \preceq I$. By assumption class definitions are well-typed, so for any class C that implements interface I we have $\mathit{match}(m, \overline{T} \to T, C)$. By Lemma 1, $[\![\overline{e}]\!]_{a\circ l}$ similarly preserves the type of $\overline{e}$. Let $\Delta'' = \Delta'[f \mapsto \mathsf{fut}\langle T\rangle]$. Since $\mathit{fresh}(f)$ we know that $f \notin \mathit{dom}(\Delta')$, so if $\Delta' \vdash_R \mathit{cn}$ **ok**, then $\Delta'' \vdash_R \mathit{cn}$ **ok**. Since $\Delta' \vdash e!m(\overline{e}) = \Delta''(f)$, we get $\Delta'' \vdash_R \mathit{ob}(o, a, \{l|x = f; s\}, q)$ **ok**. Furthermore, $\Delta'' \vdash \mathit{invoc}(o', f, m, \overline{v})$ **ok** and $\Delta'' \vdash_R \mathit{fut}(f, \perp)$ **ok**. The case $e =$ **this** is similar, but uses the class of **this** directly for the match (so internal methods are also visible).

– Bind-Mtd. Let $C = \Delta(o)$. By assumption $\Delta \vdash_R \mathit{invoc}(o, f, m, \overline{v})$ **ok** and $\Delta \vdash_R \mathit{ob}(o, a, p, q)$ **ok**, so $\Delta(f) = \mathsf{fut}\langle T\rangle$, $\Delta(\overline{v}) = \overline{T}$, and $\mathit{match}(m, \overline{T} \to T, C)$. Let $\overline{x}$ be the formal parameters of m in C. Consequently, the auxiliary function $\mathit{bind}(o, f, m, \overline{v}, C)$ returns a process $\{l[\overline{T}\ \overline{x}\ \overline{v}, \mathsf{fut}\langle T\rangle$ destiny $f]|s\}$ which is well-typed in $\Delta \circ \mathit{fields}(C)$, and it follows that $\Delta \vdash_R \mathit{ob}(o, a, p, q \cup \{\mathit{bind}(o, f, m, \overline{v}, C)\})$ **ok**.
– Return. By assumption, we have $\Delta \vdash_R \mathit{ob}(o, a, \{l|$**return** $e; s\}, q)$ **ok** and $\Delta \vdash_R \mathit{fut}(f, \perp)$ **ok**. Obviously, $\Delta \vdash_R \mathit{ob}(o, a, \{l|s\}, q)$ **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 T-Return, $\Delta \vdash_R e : T$ and by Lemma 1, $\Delta(v) = T$, so $\Delta \vdash_R \mathit{fut}(f, v)$ **ok**.
– Read-Fut. By assumption, $\Delta \vdash_R \mathit{ob}(o, a, \{l|x = e.$**get**$; s\}, q)$ **ok**, $\Delta \vdash_R \mathit{fut}(f, v)$ **ok**, and $[\![e]\!]_{a\circ l} = f$. Let $\Delta(f) = \mathsf{fut}\langle T\rangle$. Consequently, $\Delta \vdash_R e.$**get** $: T$ and $\Delta(v) = T$, so $\Delta \vdash x = v$, and $\Delta \vdash_R \mathit{ob}(o, a, \{l|x = v; s\}, q)$ **ok**.
– Rem-Sync-Call. By assumption, $\Delta \vdash_R \mathit{ob}(o, a, \{l|x = e.m(\overline{e}); s\}, q)$ **ok**, $\Delta \vdash e.m(\overline{e}) : T$, and $\mathit{fresh}(f)$. Let $\Delta' = \Delta[f \mapsto \mathsf{fut}\langle T\rangle]$. Then obviously $\Delta' \vdash f = e!m(\overline{e}); x = f.$**get**.
– Self-Sync-Call and Cog-Sync-Call. By assumption, the judgments $\Delta \vdash_R \mathit{ob}(o, a, \{l|x = e.m(\overline{e}); s\}, q)$ **ok**, $\Delta \vdash e.m(\overline{e}) : T$, $\Delta \vdash_R \{l'|s'\}$ **ok**, and $\mathit{fresh}(f)$ hold. Let $\Delta' = \Delta[f \mapsto \mathsf{fut}\langle T\rangle]$. Obviously $\Delta' \vdash \{l'|s';$ **cont**$(f)\}$ **ok**, $\Delta' \vdash x = f.$**get**, and $\Delta' \vdash_R \mathit{fut}(f, \perp)$ **ok**. □

7 Tool Support

The ABS language is being used and developed as part of the EU project HATS (`www.hats-project.eu`). ABS comes with considerable tool support, including 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 parsing, type checking, and basic error reporting. The compiler frontend is implemented using the JastAdd toolkit [19] 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 [10]. 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 class implements the *main* block; starting an ABS model in Maude means instantiating an object of that class. The conciseness and high level of abstraction of the Maude backend make it well-suited for experiments with language constructs and semantics. Maude also provides model-checking support, but the large size of each state, as well as the non-deterministic scheduling and concurrent execution of ABS, 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 [34], 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 complementary and attractive features for the modeler.

8 Related Work

The concurrency model provided by concurrent objects and in actor-based computation, where software units with encapsulated processors communicate asynchronously, is increasingly attracting attention due to its intuitive and compositional nature (e.g., [2, 5, 9, 13, 21, 37]). ABS uses the communication mechanisms of Creol [24] for remote communication, based on asynchronous method calls and first-class futures [13]. A distinguishing feature of Creol is the cooperative scheduling between asynchronously called methods [24], which allows active and reactive behavior to be combined within objects as well as compositional verification of partial correctness properties [3, 13]. Creol's model of cooperative scheduling has recently been generalized from single objects to groups of objects in a Java extension called JCoBox [34], which forms the basis for cogs in ABS.

Formal models are useful to clarify the intricacies of object orientation and may contribute to improve programming languages by making programs easier to understand, maintain, and analyze. Object calculi such as the ς-calculus [1] and its concurrent extension [20] aim at directly expressing features such as self-reference, encapsulation, and method calls, but asynchronous method calls are not addressed. This also applies to Obliq [8], a programming language using similar primitives to target distributed concurrent objects. The object calculus of Di Blasio and Fisher [16] has both synchronous and asynchronous method calls, but, in contrast to ABS, return values are discarded when methods are called asynchronously and the synchronous and asynchronous calls have different semantics. Caromel, Henrio, and Serpette propose ASP [9], a concurrent object calculus with asynchronous method calls and first-class futures. Compared to ABS, ASP's futures are transparent (i.e., there is no polling and the get-operation is implicit) and communication is ordered to make reductions deterministic.

The internal concurrency model of cogs in ABS follows Creol's concept of *cooperative scheduling* [24], but is 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 [22] 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 [12]. However, general monitors may be encoded in the language [24]. 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 [4, 17], and each process resembles a series of guarded atomic actions (ignoring local variable scopes). Internal reasoning control is facilitated by the explicit declaration of release points, at which class invariants should hold [3, 18].

Our type system resembles that of Featherweight Java [23], a core calculus for Java, because of its nominal approach. Featherweight Java is class-based with single inheritance, and subtyping is the reflexive and transitive closure of the subclass relation. In contrast, ABS cleanly distinguishes classes and types. Creol combined asynchronous 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 [25]. Creol's type system used an effect system [28] to infer types for future variables, 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 depending on the local control flow at runtime [26]. In contrast, future variables in ABS have explicit types for return values, which restricts reuse but allows a type analysis without effects. Compared to previous work on Creol, this paper formalizes user-defined data types and functions in the context of concurrent objects. The presented type safety results show how the typing context is dynamically extended when new objects and futures are created.

9 Conclusion

This paper presents ABS, an abstract behavioral specification language for designing executable, object-oriented, formal models of distributed systems. The language is situated between design-oriented, foundational, and implementation-oriented languages by being abstract, yet executable. The concurrency model of ABS is based on concurrent object groups (cogs) which are encapsulated behind interfaces and do not share state. While cogs may execute in parallel, there is a cooperative model of interleaving concurrency inside each cog, reflected by explicit processor release points in the language. This concurrency model is inherently compositional and allows to reason about concurrent system behavior using 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, executable specification language. We gave rigorous, mathematical definitions of its core syntax, type system, and operational semantics. We proved 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.

Acknowledgment. We thank Frank S. de Boer, Olaf Owe, and the HATS consortium for interesting discussions and their contributions to the ABS, and the anonymous referees for excellent feedback, significantly improving the paper.

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. G. R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison-Wesley, 2000.
5. J. Armstrong. *Programming Erlang: Software for a Concurrent World.* Pragmatic Bookshelf, 2007.
6. 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.
7. 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.
8. L. Cardelli. A language with distributed scope. *Comp. Sys.*, 8(1):27–59, 1995.
9. D. Caromel, L. Henrio, and B. P. Serpette. Asynchronous sequential processes. *Information and Computation*, 207(4):459–495, 2009.
10. 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.
11. P. C. Clements. A survey of architecture description languages. In *Proc. Workshop on Software Specification and Design (IWSSD'96)*, pages 16–25. IEEE, 1996.
12. O.-J. Dahl. Monitors revisited. In *A Classical Mind, Essays in Honour of C.A.R. Hoare*, pages 93–103. Prentice Hall, 1994.
13. 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.

14. Full ABS Modeling Framework (Mar 2011), Deliverable 1.2 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`
15. Verification of Behavioral Properties (Mar 2011), Deliverable 2.5 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`
16. P. Di Blasio and K. Fisher. A calculus for concurrent objects. In *Proc. CONCUR*, *LNCS* 1119, pages 655–670. Springer, Aug. 1996.
17. E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, Aug. 1975.
18. 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.
19. T. Ekman and G. Hedin. The JastAdd system: modular extensible compiler construction. *Science of Computer Programming*, 69(1–3):14–26, 2007.
20. 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.
21. P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comp. Sci.*, 410(2–3):202–220, 2009.
22. C. A. R. Hoare. Monitors: an operating systems structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.
23. 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.
24. E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.
25. 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.
26. E. B. Johnsen and I. C. Yu. Backwards type analysis of asynchronous method calls. *Journal of Logic and Algebraic Programming*, 77:40–59, 2008.
27. 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.
28. J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proc. POPL*, pages 47–57. ACM Press, 1988.
29. J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proc. ESEC*, *LNCS* 989, pages 137–153. Springer, 1995.
30. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theor. Comp. Sci.*, 96:73–155, 1992.
31. R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, May 1999.
32. B. C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.
33. G. D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:17–139, 2004.
34. J. Schäfer and A. Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. In *Proc. ECOOP*, *LNCS* 6183, pages 275–299. Springer, 2010.
35. A. van Deursen and P. Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10(1):1–18, 2002.
36. J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modelling with UML*. Object Technology Series. Addison-Wesley, 1999.
37. A. Welc, S. Jagannathan, and A. Hosking. Safe futures for Java. In *Proc. OOPSLA*, pages 439–453. ACM, 2005.