The Formal Semantics of Core ABS and ABS-NET

Karl Palmskog
School of Computer Science and Communication
KTH Royal Institute of Technology
Stockholm, Sweden
palmskog@kth.se

ABS is a language and framework for modelling distributed object-oriented systems, developed in the EU FP7 HATS project. Core ABS formalizes the key parts of ABS, including the syntax, type system, and an operational semantics in the style of rewriting logic. ABS-NET is a novel operational semantics for Core ABS programs, developed as a part of work on decentralized runtime adaptation of distributed objects. ABS-NET describes program execution on top of a network of nodes connected point-to-point with asynchronous message passing links. This report describes the syntax and semantics of Core ABS and ABS-NET, and is meant to serve as a reference, while highlighting the differences between the reference semantics of ABS programs and the ABS-NET semantics.

1 Introduction

ABS [5] is a language and framework for modelling distributed object-oriented systems, developed in the EU FP7 HATS project. Core ABS [7] is a calculus which contains the main features of ABS: a functional level for expressing data structures and side-effect free internal computations of objects, and an object level for expressing concurrent objects, and communication among such objects via method invocations. Core ABS is meant to be a canonical subset of full ABS, meaning that a full ABS program, containing features such as modules and deltas, can be transformed to an equivalent (but likely much more verbose) Core ABS program. The ABS-NET semantics is a novel operational semantics for Core ABS programs, which captures how objects execute on network nodes which are connected point-to-point using asynchronous message passing links. ABS-NET is part of ongoing work [4, 3, 8] on decentralized runtime adaptation of distributed objects, and relies on location-independent routing to support efficient, transparent, and robust (lock-free) object migration.

The purpose of this report is to serve as a reference for the syntax and semantics of Core ABS and ABS-NET. Since ABS-NET aims to preserve the expected behaviour of Core ABS programs as far as possible, a standard operational semantics of Core ABS is provided as well for comparison. The basic ideas for network execution and mobility can be extended to other Actor languages than Core ABS; the parts of ABS-NET that are specific to Core ABS are therefore treated separately from the parts that are largely language-independent.

1.1 Comparison with the Original Core ABS

The runtime unit of concurrency in Core ABS as originally defined [7] is a concurrent object group (cog). A cog contains one or more runtime objects, which perform cooperative scheduling of tasks. The variant of Core ABS in this report has a single object as the unit of concurrency rather than a cog, similar to the variant of Albert et al. [1]. This means that all individual objects can be viewed as actors, having local store and communicating between themselves only via asynchronous message passing. Besides the

changes mentioned here, however, symbols, terminology, and rule names have been preserved as far as possible.

The presentation of the functional level of the Core ABS differs in minor ways from the original. Types for futures, which are artifacts of the object level, are included among ground types for completeness. For similar reasons, runtime identifiers and boolean constants have been included explicitly among ground terms. At the object level, the cog declaration has been omitted from assignments that create objects. In addition, the syntax for statements and methods has been changed so that a single return statement at the end of a method body is mandatory, which is in line with the current reference implementation [5]. The object-level syntax uses lists of statements rather than semicolon-delimited composition of statements. In the operational semantics, some transition rules have been modified based on the syntax changes, or to reflect the change in unit of concurrency. Also, the previously implicit rules for while-loops and idle processes have been made explicit.

1.2 Structure

We present the syntax and type system of Core ABS without cogs in Section 2 and Section 3. Subsequently, we define the two operational semantics: the standard operational semantics in Section 4, and the ABS-NET semantics in Section 5. The presentation of the latter semantics follows the pattern of presentation of the former, as far as possible. The transition systems are given in the style of the rewriting logic Maude [2], but at a more abstract level; there exists more implementation-oriented Maude encodings (and Java implementations) of both semantics that differ from the idealized versions presented here.

2 Syntax of Core ABS

In this section, we define the syntax of Core ABS without cogs, which involves defining the syntax of (1) its functional level with algebraic data types and (side-effect free) functions, and (2) its object level with interfaces, objects, methods and statements.

2.1 Functional Level

The definition of the syntax of the functional level of Core ABS is given in Figure 1. The use of an overline on a syntactic variable signifies a list of syntactic entities, as in $\overline{e}$ and $\overline{x}$. The delimiter for a list is implicit, but in most cases a comma. For variable and method type declarations, there is a slight abuse of notation for conciseness, namely, $\overline{T\,x}$; is a possibly empty list $T_1\,x_1; ..; T_n x_n;$.

Ground types include basic types such as Bool and Int, which can be considered built in, and also user-defined algebraic data types D and user-declared interfaces I. In contrast to ground types, a type A can contain type variables N, enabling polymorphism for data types and functions. In a data type declaration Dd, possibly parameterized with the type variables $\overline{N}$, there must be at least one constructor $Cons$, possibly with a list of parameters $\overline{A}$. Function declarations F, which again may be parameterized with variables $\overline{N}$, include a return type A, a list of function parameters $\overline{T\,x}$ with their types, and an expression e.

Expressions e are boolean expressions b, variables x, ground terms t (at the object level, referred to as values v), special variables **this** and **destiny**, data type constructor expressions $Co(\overline{e})$, function expressions $fn(\overline{e})$ and case branches **case** e $\{\overline{br}\}$. Boolean expressions are mentioned explicitly because of their use in the object level and its semantics, but do not differ significantly from expressions typed by user-defined data types. Such expressions can consist of variables, function expressions, and the ground

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

Figure 1: Core ABS functional level syntax. Square brackets [] are used for optional elements.

terms **True** and **False**, composed by standard operators such as conjunction and disjunction. Patterns p can be used to decompose a constructor in a case branch, checking for term equality or binding variables to subterms. Exhaustiveness of case branches for a given case expression type is not enforced.

The data type and function declaration below gives an example of the use of type variables and case matching. The example defines the standard binary tree data type and the accompanying containment decision function for elements. The function must be written with explicit recursion and case matching, since there are no higher-order operators such as fold; functions are not terms.

```
data Tree<A> = Tip | Node(A, Tree<A>, Tree<A>);

def Bool contains<A>(Tree<A> t, A a) =
  case t {
      Tip => False;
      Node(a, _, _) => True;
      Node(_, xt, yt) => contains(xt, a) || contains(yt, a);
  };
```

The functional level is intermingled with the object level in that interface names I are ground types, there are special expression keywords **this** and **destiny**, and there are special ground terms **null**, o and f. **null** plays the part of default value for interface and future types. A future value f has type $\mathbf{Fut}\langle T\rangle$, for some ground type T, capturing the fact that it is a placeholder for a yet-to-be-seen value of the type T. An object identifier o is generated at runtime during object instantiation and is typed by its class C.

2.2 Object Level

The object level syntax, shown in Figure 2, defines interfaces, classes, methods, object creation and method calls. A Core ABS program P defines data types, functions, interfaces, classes and a list of statements (main block) that is executed initially. An interface declaration IF consists of an interface name I and $\overline{Sg}$;, which, again by slight abuse of notation, is a possibly empty list of method signatures $Sg_1\,;..;Sg_n\,;$. A class declaration CL consists of a class name C, an optional list of interfaces $\overline{I}$ whose methods the class implements, and a list of method declarations $\overline{M}$. The first, optional, comma-separated list of variable-type declarations $\overline{T\ x}$ defines mandatory constructor parameters that must be given when instantiating an object of the class with **new** $C(\overline{e})$. The second, semicolon-separated such list declares the class fields, which assume the default values for their types on instantiation. Classes can optionally define a method named **init** to manually initialize the fields to other values.

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

Figure 2: Core ABS object level syntax.

A method signature Sg consists of a return type T, a method name m, and a list of parameter variables and their types. Methods M have a signature, a list of declarations of local variables, a list of statements $\overline{s}$, and a single, final return statement.

An asynchronous method invocation statement $x = e!m(\overline{e})$; does not block, and assigns a future identifier to the variable x, with e reducing to the callee's identifier and $\overline{e}$ the argument list. The actual return value of the invocation can later be retrieved into the variable y by the assignment $y = x.\textbf{get}$; which possibly blocks. Synchronous method invocation $x = e.m(\overline{e})$; can block and assigns a value of the method's return type directly. The special **this** variable can be used by a class to call internal methods, which are possibly not defined in any interface.

Guards g consist of ordinary boolean conditions b and special tests $e?$, which check that the future value e reduces to is resolved, i.e., that the associated method invocation has finished. If a tested future is unresolved, the guard is false and the current process is suspended, making it possible for other processes to execute. The statement **suspend**; allows direct, unconditional suspension.

An example of an interface with implementing classes from a Core ABS program is given below. The `CastNode` interface defines a method `aggregate`, which, when called on some object, is intended to perform a convergecast operation in the object-reference binary tree rooted at that object. Specifically, this means that if an object implementing `CastNode` is a leaf in the tree (an instance of class `LeafNode`), it simply returns a locally known integer, but if the object has child nodes in the tree (an instance of class `BranchNode`), `aggregate` is called on both of those objects and the results are added to the local integer and returned. In this way, the `aggregate` method for the object o always returns the aggregate (sum) of all local integer values in the binary tree of objects rooted at o.

```
interface CastNode {
    Int aggregate();
}

class LeafCastNode(Int val) implements CastNode {
    Int aggregate() { return val; }
}

class BranchCastNode(Int val, CastNode left, CastNode right) implements CastNode {
   Int aggregate() {
      Fut<Int> fLeft = left!aggregate();
      Fut<Int> fRight = right!aggregate();
      Int aggregateLeft = fLeft.get;
      Int aggregateRight = fRight.get;
      return val + aggregateLeft + aggregateRight;
    }
}
```

3 Type System of Core ABS

The type system of Core ABS can be divided into a part for the functional level and a part for the object level, with the latter building heavily upon the former.

3.1 Functional Level

The functional level well-typing relation, defined in Figure 3, is given with respect to a typing context in the form of a finite map Γ from variables and constants to types. A lookup on the variable x in Γ is given by $\Gamma(x)$. The addition to Γ of a binding of x to T is given by $\Gamma[x \mapsto T]$; an existing binding for x to another type in Γ is not relevant in the resulting map. Two finite maps Γ and Γ' can be composed to form a map $\Gamma \circ \Gamma'$, so that $\Gamma \circ \Gamma'(x) = \Gamma'(x)$ whenever there is binding for x in Γ', and $\Gamma \circ \Gamma'(x) = \Gamma(x)$ otherwise. Γ and Γ' are in the extension relation, $\Gamma \subseteq \Gamma'$, whenever Γ' has bindings for all keys with bindings in Γ, and the corresponding types coincide. $[]$ is the empty map, while $[x \mapsto T]$ is the map with the single binding of x to T.

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

$$\frac{\Gamma(x) = A}{\Gamma \vdash x : A}\ (\text{T-Var}) \qquad \frac{\texttt{tmatch}(\overline{A}, \overline{A}') = \sigma \quad \sigma \neq \bot \quad \Gamma \vdash \overline{e} : \overline{A}' \quad \Gamma(fn) = \overline{A} \to A}{\Gamma \vdash fn(\overline{e}) : A\,\sigma}\ (\text{T-FuncExpr})$$

$$\frac{\Gamma \vdash \overline{e} : \overline{A}' \quad \sigma \neq \bot \quad \texttt{tmatch}(\overline{A}, \overline{A}') = \sigma \quad \Gamma(Co) = \overline{A} \to D[\langle \overline{N} \rangle]}{\Gamma \vdash Co(\overline{e}) : D[\langle \overline{N} \rangle]\,\sigma}\ (\text{T-ConsExpr})$$

$$\frac{\Gamma(o) = C}{\Gamma \vdash o : C}\ (\text{T-ObjectId}) \qquad \frac{\Gamma(f) = \textbf{Fut}\,\langle T \rangle}{\Gamma \vdash f : \textbf{Fut}\,\langle T \rangle}\ (\text{T-FutureId}) \qquad \frac{}{\Gamma \vdash \_ : A}\ (\text{T-Wildcard}) \qquad \frac{}{\Gamma \vdash b : \mathsf{Bool}}\ (\text{T-Bool}) \qquad \frac{}{\Gamma \vdash \textbf{null} : A}\ (\text{T-Null})$$

$$\frac{\Gamma(fn) = A_1, .., A_n \to A \quad \Gamma[x_1 \mapsto A_1, .., x_n \mapsto A_n] \vdash e : A}{\Gamma \vdash \textbf{def}\,A\,fn[\langle \overline{N} \rangle](A_1\,x_1, .., A_n\,x_n) = e;}\ (\text{T-FuncDecl}) \qquad \frac{\Gamma' \vdash p : A \quad \Gamma' \vdash e : A' \quad \Gamma' = \Gamma \circ \texttt{psubst}\,(p, A, \Gamma)}{\Gamma \vdash p \Rightarrow e; : A \to A'}\ (\text{T-Branch})$$

$$\frac{\Gamma \vdash e : A \quad \Gamma \vdash \overline{br} : A \to A'}{\Gamma \vdash \textbf{case}\ e\ \{\overline{br}\} : A'}\ (\text{T-Case})$$

Figure 3: Core ABS functional level type system.

An initial typing context is assumed to map the names of the data types and function declarations under consideration to appropriate types, which is reflected in the rules T-ConsDecl and T-FuncDecl. The rule T-Null allows the **null** term to have any type. The rule T-Var types a variable according to the type recorded for it in the context, as is done in, e.g., T-FuncDecl. The `tmatch` auxiliary function, used in T-FuncExpr and T-ConsExpr, attempts to match the type variables of the formal parameter types to the actual parameter types; if there is no match, `tmatch` returns $\bot$. The `psubst` auxiliary function, used in the rule T-Branch, constructs a typing context for which a pattern must be well-typed. If A is a type variable N, then p is a variable x and $\texttt{psubst}(p, N, \Gamma) = [x \to N]$. If, on the other hand, A is not a type variable, we define the result based on the structure of p. If $p = x$ and $\Gamma(x) = T$, then $\texttt{psubst}(p, N, \Gamma) = []$. If $p = x$ and Γ has no binding for x, then $\texttt{psubst}(p, N, \Gamma) = [p \mapsto A]$. If $p = t$ or $p = \_$, then $\texttt{psubst}(p, N, \Gamma) = []$. Finally, if $p = Co(p_1, .., p_n)$ and $\Gamma(Co) = A_1, .., A_n \to A$, then $\texttt{psubst}(p, N, \Gamma) = \texttt{psubst}(p_1, A_1, \Gamma) \circ .. \circ \texttt{psubst}(p_n, A_n, \Gamma)$. The subtyping relation $\preceq$, used in the rule T-Sub, is such that $C \preceq I$ whenever the class C implements the interface I. The relation also anticipates extension to interface subtyping via inheritance, which is present in full ABS but not Core ABS.

3.2 Object Level

The well-typing relation of the object level, shown in Figure 4, proceeds in a straightforward way on the syntactic structure of the programs, for the most part. In the rule T-PROGRAM, a program is well-typed with respect to a context when its all data declarations, function declarations and class declarations are well-typed; typing of interfaces is simpler and is omitted. For the typing of the statement list $\overline{s}$, all variable declarations are added to the context.

$$\frac{\Gamma \vdash e : \mathbf{Fut}\langle T\rangle}{\Gamma \vdash e? : \mathsf{Bool}}\ (\text{T-POLL}) \qquad \frac{\Gamma \vdash e : \mathbf{Fut}\langle T\rangle}{\Gamma \vdash e.\mathbf{get} : T}\ (\text{T-GET}) \qquad \frac{}{\Gamma \vdash \mathbf{skip};}\ (\text{T-SKIP}) \qquad \frac{\Gamma \vdash g : \mathsf{Bool}}{\Gamma \vdash \mathbf{await}\, g;}\ (\text{T-AWAIT}) \qquad \frac{}{\Gamma \vdash \mathbf{suspend};}\ (\text{T-SUSPEND})$$

$$\frac{\Gamma \vdash x : T \quad \Gamma \vdash rhs : T}{\Gamma \vdash x = rhs;}\ (\text{T-ASSIGN}) \qquad \frac{\Gamma \vdash g : \mathsf{Bool} \quad \Gamma \vdash g' : \mathsf{Bool}}{\Gamma \vdash g \wedge g' : \mathsf{Bool}}\ (\text{T-AND}) \qquad \frac{\Gamma \vdash \overline{e} : \mathtt{ptypes}(C) \quad I \in \mathtt{interfaces}(C)}{\Gamma \vdash \mathbf{new}\, C(\overline{e}) : I}\ (\text{T-NEW})$$

$$\frac{\Gamma \vdash e.m(\overline{e}) : T}{\Gamma \vdash e!m(\overline{e}) : \mathbf{Fut}\langle T\rangle}\ (\text{T-ASYNCCALL})$$

$$\frac{\Gamma \vdash b : \mathsf{Bool} \quad \Gamma \vdash \overline{s} \quad [\Gamma \vdash \overline{s'}]}{\Gamma \vdash \mathbf{if}\, b\, \{\overline{s}\}\, [\mathbf{else}\, \{\overline{s'}\}]}\ (\text{T-CONDITIONAL}) \qquad \frac{\Gamma \vdash b : \mathsf{Bool} \quad \Gamma \vdash \overline{s}}{\Gamma \vdash \mathbf{while}\, b\, \{\overline{s}\}}\ (\text{T-WHILE}) \qquad \frac{\Gamma \vdash e : I \quad \Gamma \vdash \overline{e} : \overline{T} \quad \mathtt{match}(m, \overline{T} \to T, I)}{\Gamma \vdash e.m(\overline{e}) : T}\ (\text{T-SYNCCALL})$$

$$\frac{\Gamma' = \Gamma[x_1 \mapsto T_1, .., x_i \mapsto T_i, x'_1 \mapsto T'_1, .., x'_j \mapsto T'_j] \quad \Gamma'' = \Gamma'[\mathbf{destiny} \mapsto \mathbf{Fut}\langle T\rangle] \quad \Gamma'' \vdash \overline{s} \quad \Gamma'' \vdash e : T}{\Gamma \vdash T\, m(T_1\, x_1, .., T_i\, x_i)\, \{T'_1\, x'_1; ..; T'_j\, x'_j; \overline{s}\ \mathbf{return}\ e;\}}\ (\text{T-METHOD})$$

$$\frac{[\forall I \in \overline{I}.\, \mathtt{implements}(C, I)] \quad \Gamma[\mathbf{this} \mapsto C, \mathtt{fields}(C)] \vdash \overline{M}}{\Gamma \vdash \mathbf{class}\, C\, [(\overline{T}\ \overline{x})]\, [\mathbf{implements}\, \overline{I}]\, \{\overline{T'\ x'};\ \overline{M}\}}\ (\text{T-CLASS})$$

$$\frac{\Gamma[x_1 \mapsto T_1, .., x_n \mapsto T_n] \vdash \overline{s} \quad \forall Dd \in \overline{Dd}.\, \Gamma \vdash Dd \quad \forall F \in \overline{F}.\, \Gamma \vdash F \quad \forall CL \in \overline{CL}.\, \Gamma \vdash CL}{\Gamma \vdash \overline{Dd}\,\overline{F}\,\overline{IF}\,\overline{CL}\, \{T_1\, x_1; ..; T_n\, x_n;\ \overline{s}\}}\ (\text{T-PROGRAM})$$

Figure 4: Core ABS object level type system.

In the rule T-METHOD, parameter declarations and local variable declarations are again added the context before deferring to the well-typedness of the statement list and return expression. In addition, a binding is added for the special expression **destiny** to the type for the return expression e, so that **destiny** can be used as a variable containing the identifier of the future a method invocation produces.

The rule T-POLL formalizes the requirement that a future resolution test must be performed on an expression that actually reduces to a future identifier, as does T-GET for the .**get** operator. In T-NEW, for a given class identifier C, the auxiliary function `ptypes` returns the class parameter types and `interfaces` returns the set of identifiers of the interfaces which the class implements. The effect of the rule is that all variables containing object identifiers must be typed with an interface which the class of the object implements—not the implementing class itself.

In T-CLASS, the auxiliary function `implements` is used to check that the given class properly implements the methods of all the interfaces in the list $\overline{I}$. The auxiliary function `fields` produces type bindings for parameters and fields for a given class. The addition of a binding for the special expression **this** allows methods to call class-internal methods, by masquerading the class name C as an interface type. In T-SYNCCALL, the auxiliary function `match` checks that types of the given expression list of arguments coincides with the actual parameter types specified for the method in the interface or class. T-ASYNCCALL expresses the typing of asynchronous method invocations as having the future type of the corresponding synchronous invocation. The rules for skip statements, suspend statements, conditionals, and while loops are standard.

4 Standard Operational Semantics of Core ABS

In the standard operational semantics of Core ABS, a runtime configuration consists of objects, futures and method invocations. In this section, a reduction system for functional (side-effect free) expressions paves the way to a transition system that describes how the entities evolve and interact. The transition system rules apply to subsets of a global configuration, modulo rearrangement of entities to fit the left-hand side of the rules, as in the Maude style of modelling distributed systems [2]. The execution of a program is a possibly infinite sequence of global configurations, such that the transition between a previous configuration and the next is valid in the system.

4.1 Runtime Configurations

In the standard runtime syntax, shown in Figure 5, configurations *cn*, consisting of futures, objects, and method invocations (*fut*, *object*, *invoc*), are composed via a whitespace operator, with ε being the empty configuration. A global configuration is shown inside curly brackets, e.g., $\{cn\}$.

cn	::=	$\varepsilon \mid fut \mid object \mid invoc \mid cn\ cn$	pr	::=	$process \mid \textbf{idle}$
fut	::=	$\mathsf{fut}(f, val)$	a, l	::=	$\varepsilon \mid T\ x\ v \mid a, a$
$object$	::=	$\mathsf{ob}(o, a, pr, q)$	val	::=	$v \mid \bot$
$process$	::=	$\{l \mid \overline{sp}\} \mid \textbf{error}$	sp	::=	$\textbf{return}\, e; \mid \textbf{cont}(f); \mid s$
$invoc$	::=	$\mathsf{invoc}(o, f, m, \overline{v})$	q	::=	$\emptyset \mid process \mid q\ q$

Figure 5: Core ABS standard runtime syntax.

A future $\mathsf{fut}(f, val)$ has a future identifier f and, by the definition of *val*, either a resolved value (ground term) v, or $\bot$ to indicate its status of being unresolved. An object $\mathsf{ob}(o, a, pr, q)$ has an object identifier o, a store a for its field types and values, an active process pr and a pool q of suspended processes. A normal process *process* has a store l of local variable types and values, and a list of process statements. ε refers also to the empty store, disambiguated from the empty configuration by context. Process statements *sp* are statements *s* extended with a return statement $\textbf{return}\, e;$ and a continuation statement $\textbf{cont}(f);$. A method invocation $\mathsf{invoc}(o, f, m, \overline{v})$ contains the intended recipient object's identifier o, the associated future identifier f, the method name m, and a list of argument values $\overline{v}$. Data types, functions, interfaces and classes are not represented explicitly in runtime configurations, since they are assumed to be static throughout execution.

4.2 Reduction System for Functional Expressions

Given a substitution σ binding variables to terms, functional expressions can be reduced to terms in accordance with the reduction system shown in Figure 6, which defines the reduction relation $\sigma \vdash e \rightsquigarrow \sigma' \vdash e'$. The relation intuitively holds when the expression e in the context σ can be reduced to the expression e' in the context σ'. There is no guarantee that a sequence of valid reductions eventually leads to a ground term; recursive functions can lead to infinite reduction sequences, and incomplete case branch coverage can make the sequence halt on an expression that is not a ground term.

Evaluation of a function expression involving the function *fn*, as defined in the rules RedFunExp and RedFunGround, proceeds by first reducing all argument expressions to terms. Suppose the list of parameter variables in the declaration of *fn* is $x_1, \ldots, x_n$ and the expression in the declaration is e_{fn}. The function expression is then replaced with the expression that results from syntactically replacing

$$\frac{\sigma \vdash e_i \rightsquigarrow \sigma' \vdash e_i' \quad 1 \le i \le n}{\sigma \vdash Co(e_1,..,e_i,..,e_n) \rightsquigarrow \sigma' \vdash Co(e_1,..,e_i',..,e_n)}\ (\textsc{RedCons}) \qquad \frac{\sigma \vdash e_i \rightsquigarrow \sigma' \vdash e_i' \quad 1 \le i \le n}{\sigma \vdash fn(e_1,..,e_i,..,e_n) \rightsquigarrow \sigma' \vdash fn(e_1,..,e_i',..,e_n)}\ (\textsc{RedFunExp})$$

$$\frac{\sigma(x) = t}{\sigma \vdash x \rightsquigarrow \sigma \vdash t}\ (\textsc{RedVar}) \qquad \frac{\sigma \vdash e \rightsquigarrow \sigma' \vdash e'}{\sigma \vdash \textbf{case}\, e\, \{\overline{br}\} \rightsquigarrow \sigma' \vdash \textbf{case}\, e'\, \{\overline{br}\}}\ (\textsc{RedCase1}) \qquad \frac{\texttt{match}(\sigma(p),t) = \bot}{\sigma \vdash \textbf{case}\, t\, \{p \Rightarrow e; \overline{br}\} \rightsquigarrow \sigma \vdash \textbf{case}\, t\, \{\overline{br}\}}\ (\textsc{RedCase3})$$

$$\frac{\begin{array}{l}\texttt{match}(\sigma(p),t) = \sigma' \quad \sigma' \neq \bot \\ \texttt{fresh}(\{y_1,\ldots,y_n\}) \quad \texttt{vars}(\sigma(p)) = \{x_1,\ldots,x_n\} \\ \sigma'' = \sigma[y_1 \mapsto \sigma'(x_1),..,y_n \mapsto \sigma'(x_n)]\end{array}}{\begin{array}{c}\sigma \vdash \textbf{case}\, t\, \{p \Rightarrow e; \overline{br}\} \rightsquigarrow \\ \sigma'' \vdash e[x_1 \mapsto y_1,..,x_n \mapsto y_n]\end{array}}\ (\textsc{RedCase2}) \qquad \frac{\begin{array}{l}\overline{x}_{fn} = x_1,\ldots,x_n \\ \texttt{fresh}(\{y_1,\ldots,y_n\})\end{array}}{\begin{array}{c}\sigma \vdash fn(t_1,\ldots,t_n) \rightsquigarrow \\ \sigma[y_1 \mapsto t_1,..,y_n \mapsto t_n] \vdash e_{fn}[x_1 \mapsto y_1,..,x_n \mapsto y_n]\end{array}}\ (\textsc{RedFunGround})$$

Figure 6: Core ABS reduction rules for functional expressions.

the parameters in e_{fn} with the new, unique names $y_1, \ldots, y_n$, and the substitution is extended to include bindings from the new names to the respective terms.

Let $\texttt{dom}(\sigma)$ be the set of names which are bound in a substitution σ, and let `vars` be the function which returns the set of variables in a pattern. The `match` function in the rules RedCase2 and RunCase3, given a pattern p and a term t, returns a substitution σ such that $\sigma(p) = t$ and $\texttt{dom}(\sigma) = \texttt{vars}(p)$ if it exists, and $\bot$ otherwise. We define a substitution σ as well-typed for a given context Γ, written $\Gamma \vdash \sigma$, whenever, for all x bound in σ, $\Gamma \vdash \sigma(x) : \Gamma(x)$. This definition is used to state the type preservation property of Lemma 1.

Lemma 1 (Type Preservation)**.** *Let Γ be a typing context and σ be 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. By induction on rule applications [7]. □

4.3 Standard Operational Semantics of Concurrent Objects

Guard expressions are not covered by the reduction system for functional expressions in the previous subsection. Guard evaluation is different since it potentially depends on the state of a global configuration. Figure 7 shows the reduction rules for guards. Below, the boolean result of evaluating a guard expression g with respect to a configuration cn and a store a, if it exists, is written as $[\![g]\!]_a^{cn}$. Similarly, the ground term result of evaluating an expression e with respect to a store a, if it exists, is written as $[\![e]\!]_a$.

$$\frac{\sigma \vdash b \rightsquigarrow \sigma \vdash b'}{\begin{array}{c}\sigma, cn \vdash b \\ \rightsquigarrow \sigma, cn \vdash b'\end{array}}\ (\textsc{RedBoolGuard}) \qquad \frac{\sigma \vdash e \rightsquigarrow \sigma \vdash f \quad \mathsf{fut}(f,v) \in cn}{\begin{array}{c}\sigma, cn \vdash e? \\ \rightsquigarrow \sigma, cn \vdash \textbf{True}\end{array}}\ (\textsc{RedReplyGuard1}) \qquad \frac{\sigma \vdash e \rightsquigarrow \sigma \vdash f \quad \mathsf{fut}(f,\bot) \in cn}{\begin{array}{c}\sigma, cn \vdash e? \\ \rightsquigarrow \sigma, cn \vdash \textbf{False}\end{array}}\ (\textsc{RedReplyGuard2})$$

$$\frac{\sigma, cn \vdash g_1 \rightsquigarrow \sigma, cn \vdash g_1' \quad \sigma, cn \vdash g_2 \rightsquigarrow \sigma, cn \vdash g_2'}{\begin{array}{c}\sigma, cn \vdash g_1 \wedge g_2 \\ \rightsquigarrow \sigma, cn \vdash g_1' \wedge g_2'\end{array}}\ (\textsc{RedGuards})$$

Figure 7: Core ABS reduction rules for guard expressions.

The set of transition rules for configurations is split between Figure 8 and Figure 9. The rule Suspend puts the current active process into the pool of inactive processes, allowing another process to run with the help of rule Activate. The `select` auxiliary function decides the process to make active for an idle object, given the complete global state. The function is implementation-specific and thus left unspecified, in effect providing a hook for different schedulers. $q \setminus process$ is the pool q with process $process$ removed,

and $q \cup process$ is q with $process$ added. If a is a store, then $a[x \mapsto v]$ is the store that results when replacing the value for the variable x with v in a.

In BIND-MTD, the `bind` auxiliary function produces a process from a method invocation, retrieving the statements to execute and initializing local variables in the process store, among them **destiny**, which is assigned the future identifier f. The resulting process is then put in the pool, and the invocation removed. The rule RETURN subsequently uses the value stored in **destiny** to find the future to resolve in the configuration. The `class` auxiliary function, which takes an object identifier as an argument, returns the class associated with the identifier. Note that `bind` returns the process **error** if the class does not have the method indicated, or there is an argument-parameter mismatch.

$$\frac{x \in \mathtt{dom}(l) \quad [\![e]\!]_{a \circ l} = v}{\mathsf{ob}(o,a,\{l \mid x = e; \overline{sp}\},q) \to \mathsf{ob}(o,a,\{l[x \mapsto v] \mid \overline{sp}\},q)} \text{(ASSIGN-LOCAL)}$$

$$\frac{x \in \mathtt{dom}(a) \quad [\![e]\!]_{a \circ l} = v}{\mathsf{ob}(o,a,\{l \mid x = e; \overline{sp}\},q) \to \mathsf{ob}(o,a[x \mapsto v],\{l \mid \overline{sp}\},q)} \text{(ASSIGN-FIELD)}$$

$$\frac{[\![e]\!]_{a \circ l} = f}{\mathsf{ob}(o,a,\{l \mid x = e.\mathbf{get}; \overline{sp}\},q)\ \mathsf{fut}(f,v) \to \mathsf{ob}(o,a,\{l \mid x = v; \overline{sp}\},q)\ \mathsf{fut}(f,v)} \text{(READ-FUT)}$$

$$\frac{}{\mathsf{ob}(o,a,\{l \mid \mathbf{skip}; \overline{sp}\},q) \to \mathsf{ob}(o,a,\{l \mid \overline{sp}\},q)} \text{(SKIP)}$$

$$\frac{[\![b]\!]_{a \circ l} = \mathbf{True}}{\mathsf{ob}(o,a,\{l \mid \mathbf{if}\ b\ \{\overline{s}\}\ [\mathbf{else}\ \{\overline{s'}\}]\ \overline{sp}\},q) \to \mathsf{ob}(o,a,\{l \mid \overline{s}\ \overline{sp}\},q)} \text{(COND-TRUE)}$$

$$\frac{[\![b]\!]_{a \circ l} = \mathbf{False}}{\mathsf{ob}(o,a,\{l \mid \mathbf{if}\ b\ \{\overline{s}\}\ [\mathbf{else}\ \{\overline{s'}\}]\ \overline{sp}\},q) \to \mathsf{ob}(o,a,\{l \mid [\overline{s'}]\ \overline{sp}\},q)} \text{(COND-FALSE)}$$

$$\frac{[\![g]\!]^{cn}_{a \circ l} = \mathbf{True}}{\{\mathsf{ob}(o,a,\{l \mid \mathbf{await}\ g; \overline{sp}\},q)\ cn\} \to \{\mathsf{ob}(o,a,\{l \mid \overline{sp}\},q)\ cn\}} \text{(AWAIT-TRUE)}$$

$$\frac{[\![g]\!]^{cn}_{a \circ l} = \mathbf{False}}{\{\mathsf{ob}(o,a,\{l \mid \mathbf{await}\ g; \overline{sp}\},q)\ cn\} \to \{\mathsf{ob}(o,a,\{l \mid \mathbf{suspend}; \mathbf{await}\ g; \overline{sp}\},q)\ cn\}} \text{(AWAIT-FALSE)}$$

$$\frac{}{\mathsf{ob}(o,a,\{l \mid \},q) \to \mathsf{ob}(o,a,\mathbf{idle},q)} \text{(IDLE)}$$

$$\frac{\mathtt{bind}(o,f,m,\overline{v},\mathtt{class}(o)) = process}{\mathsf{ob}(o,a,p,q)\ \mathsf{invoc}(o,f,m,\overline{v}) \to \mathsf{ob}(o,a,p,q \cup process)} \text{(BIND-MTD)}$$

$$\frac{[\![e]\!]_{a \circ l} = o' \quad [\![\overline{e}]\!]_{a \circ l} = \overline{v} \quad \mathtt{fresh}(f)}{\mathsf{ob}(o,a,\{l \mid x = e!m(\overline{e}); \overline{sp}\},q) \to \mathsf{ob}(o,a,\{l \mid x = f; \overline{sp}\},q)\ \mathsf{invoc}(o',f,m,\overline{v})\ \mathsf{fut}(f,\bot)} \text{(ASYNC-CALL)}$$

$$\frac{\mathtt{select}(q,a,cn) = process}{\{\mathsf{ob}(o,a,\mathbf{idle},q)\ cn\} \to \{\mathsf{ob}(o,a,process,q \setminus process)\ cn\}} \text{(ACTIVATE)}$$

$$\frac{[\![e]\!]_{a \circ l} = v \quad l(\mathbf{destiny}) = f}{\mathsf{ob}(o,a,\{l \mid \mathbf{return}\ e; \overline{sp}\},q)\ \mathsf{fut}(f,\bot) \to \mathsf{ob}(o,a,\{l \mid \overline{sp}\},q)\ \mathsf{fut}(f,v)} \text{(RETURN)}$$

$$\frac{}{\mathsf{ob}(o,a,\{l \mid \mathbf{suspend}; \overline{sp}\},q) \to \mathsf{ob}(o,a,\mathbf{idle},q \cup \{l \mid \overline{sp}\})} \text{(SUSPEND)}$$

Figure 8: Standard Core ABS reduction rules of concurrent objects (1).

In NEW-OBJECT, $\mathtt{fresh}(o')$ ensures that the identifier o' is globally unique, `init` produces a process for the initializing method of the class (or an empty process if such a method does not exist), and `atts` sets the field values for the new object, including the special field **this** which is given the value o'. In REM-SYNC-CALL, the new, fresh variable y is introduced to hold the future identifier associated with an asynchronous call; to type the variable properly at run time in the store, the method's return type is looked up in the class using the `returns` auxiliary function.

The rules SELF-SYNC-CALL and SELF-SYNC-RETURN-SCHED provide the justification for extending process statements with a continuation statement. When an object calls itself in SELF-SYNC-CALL, control passes to a new process and must then somehow be passed back. Therefore, a continuation statement containing the future identifier of the caller process is added to the callee process, and eventually consumed through the rule SELF-SYNC-RETURN-SCHED.

The runtime typing rules in Figure 10, distinguished by the turnstile subscript R and the suffix **ok** in conclusions, extend the static typing systems in previous sections to runtime configurations. The auxiliary function `match` in T-INVOC is the same as in T-SYNCCALL. In T-OBJECT, `fields` constructs a mapping from fields names to field types for a class. A runtime typing context Δ is assumed to contain bindings

$$\frac{l(\textbf{destiny}) = f' \quad [\![e]\!]_{a\circ l} = o \quad [\![\overline{e}]\!]_{a\circ l} = \overline{v} \quad \texttt{fresh}(f) \quad \texttt{bind}(o,f,m,\overline{v},\texttt{class}(o)) = \{l' \mid \overline{sp}'\}}{\begin{array}{c}\mathsf{ob}(o,a,\{l \mid x = e.m(\overline{e});\overline{sp}\},q) \\ \to \mathsf{ob}(o,a,\{l' \mid \overline{sp}'\,\textbf{cont}(f');\},q \cup \{l \mid x = f.\textbf{get};\overline{sp}\})\;\mathsf{fut}(f,\bot)\end{array}}\ (\textsc{Self-Sync-Call})$$

$$\frac{l'(\textbf{destiny}) = f}{\mathsf{ob}(o,a,\{l \mid \textbf{cont}(f);\},q \cup \{l' \mid \overline{sp}\}) \to \mathsf{ob}(o,a,\{l' \mid \overline{sp}\},q)}\ (\textsc{Self-Sync-Return-Sched})$$

$$\frac{\texttt{fresh}(o') \quad \texttt{init}(C) = process \quad \texttt{atts}(C,[\![\overline{e}]\!]_{a\circ l},o') = a'}{\begin{array}{c}\mathsf{ob}(o,a,\{l|x = \textbf{new}\,C(\overline{e});\overline{sp}\},q) \\ \to \mathsf{ob}(o,a,\{l|x = o';\overline{sp}\},q)\;\mathsf{ob}(o',a',\textbf{idle},process)\end{array}}\ (\textsc{New-Object})$$

$$\frac{[\![b]\!]_{a\circ l} = \textbf{True}}{\begin{array}{c}\mathsf{ob}(o,a,\{l \mid \textbf{while}\,b\,\{\overline{s}\}\,\overline{sp}\},q) \\ \to \mathsf{ob}(o,a,\{l \mid \overline{s}\;\textbf{while}\,b\,\{\overline{s}\}\,\overline{sp}\},q)\end{array}}\ (\textsc{While-True})$$

$$\frac{[\![b]\!]_{a\circ l} = \textbf{False}}{\begin{array}{c}\mathsf{ob}(o,a,\{l \mid \textbf{while}\,b\,\{\overline{s}\}\,\overline{sp}\},q) \\ \to \mathsf{ob}(o,a,\{l \mid \overline{sp}\},q)\end{array}}\ (\textsc{While-False})$$

$$\frac{[\![e]\!]_{a\circ l} = o' \quad \texttt{fresh}(y) \quad \texttt{returns}(\texttt{class}(o'),m) = T}{\begin{array}{c}\mathsf{ob}(o,a,\{l \mid x = e.m(\overline{e});\overline{sp}\},q)\;\mathsf{ob}(o',a',pr,q') \\ \to \mathsf{ob}(o,a,\{l,\textbf{Fut}\,\langle T\rangle\,y\;\textbf{null} \mid y = o'!m(\overline{e}); x = y.\textbf{get};\overline{sp}\},q)\;\mathsf{ob}(o',a',pr,q')\end{array}}\ (\textsc{Rem-Sync-Call})$$

Figure 9: Standard Core ABS reduction rules of concurrent objects (2).

for runtime identifiers, i.e., object identifiers and future identifiers, to their types. This is reflected in the rules T-STATE1, T-CONT, T-FUTURE-V, T-FUTURE-BOT, and T-INVOC, when considered in conjunction with the rules T-OBJECTID and T-FUTUREID in Figure 3.

$$\frac{\Delta \vdash x : T \quad \Delta \vdash v : T}{\Delta \vdash_R T\,x\,v\;\textbf{ok}}\ (\text{T-STATE1}) \qquad \frac{\Delta \vdash f : \textbf{Fut}\,\langle T\rangle}{\Delta \vdash_R \textbf{cont}\,(f);\textbf{ok}}\ (\text{T-CONT})$$

$$\frac{\Delta \vdash f : \textbf{Fut}\,\langle T\rangle \quad \Delta \vdash v : T}{\Delta \vdash_R \mathsf{fut}\,(f,v)\,\textbf{ok}}\ (\text{T-FUTURE-V}) \qquad \frac{\Delta \vdash f : \textbf{Fut}\,\langle T\rangle}{\Delta \vdash_R \mathsf{fut}\,(f,\bot)\,\textbf{ok}}\ (\text{T-FUTURE-BOT})$$

$$\frac{\Delta \vdash_R cn\,\textbf{ok} \quad \Delta \vdash_R cn'\,\textbf{ok}}{\Delta \vdash_R cn\;cn'\,\textbf{ok}}\ (\text{T-CONFIGURATIONS}) \qquad \frac{}{\Delta \vdash_R \varepsilon\,\textbf{ok}}\ (\text{T-EMPTY})$$

$$\frac{\Delta \vdash_R q\,\textbf{ok} \quad \Delta \vdash_R q'\,\textbf{ok}}{\Delta \vdash_R q\;q'\,\textbf{ok}}\ (\text{T-PROCESS-QUEUE}) \qquad \frac{}{\Delta \vdash_R \emptyset\,\textbf{ok}}\ (\text{T-EMPTY-QUEUE})$$

$$\frac{\Delta \vdash e : T \quad \Delta(\textbf{destiny}) = \textbf{Fut}\,\langle T\rangle}{\Delta \vdash_R \textbf{return}\,e;\,\textbf{ok}}\ (\text{T-RETURN}) \qquad \frac{\Delta \vdash_R a\,\textbf{ok} \quad \Delta \vdash_R a'\,\textbf{ok}}{\Delta \vdash_R a,a'\,\textbf{ok}}\ (\text{T-STATE2})$$

$$\frac{\Delta \vdash f : \textbf{Fut}\,\langle T\rangle \quad \Delta \vdash \overline{v} : \overline{T} \quad \texttt{match}(m,\overline{T} \to T,\Delta(o))}{\Delta \vdash_R \mathsf{invoc}\,(o,f,m,\overline{v})\,\textbf{ok}}\ (\text{T-INVOC}) \qquad \frac{}{\Delta \vdash_R \textbf{idle}\,\textbf{ok}}\ (\text{T-IDLE})$$

$$\frac{\Delta' = \Delta[x_1 \mapsto T_1,..,x_n \mapsto T_n] \quad \Delta' \vdash_R T_1\,x_1\,v_1,..,T_n\,x_n\,v_n\,\textbf{ok} \quad \Delta' \vdash_R \overline{sp}\;\textbf{ok}}{\Delta \vdash_R \{T_1\,x_1\,v_1,..,T_n\,x_n\,v_n \mid \overline{sp}\}\,\textbf{ok}}\ (\text{T-PROCESS})$$

$$\frac{\texttt{fields}(\Delta(o)) = [x_1 \mapsto T_1,..,x_n \mapsto T_n] \quad \Delta' = \Delta[x_1 \mapsto T_1,..,x_n \mapsto T_n] \quad \Delta' \vdash_R pr\,\textbf{ok} \quad \Delta' \vdash_R T_1\,x_1\,v_1,..,T_n\,x_n\,v_n\,\textbf{ok} \quad \Delta' \vdash_R q\,\textbf{ok}}{\Delta \vdash_R \mathsf{ob}\,(o,T_1\,x_1\,v_1,..,T_n\,x_n\,v_n,pr,q)\,\textbf{ok}}\ (\text{T-OBJECT})$$

Figure 10: Standard Core ABS runtime typing rules.

4.4 Subject Reduction for the Standard Semantics

Lemma 2 defines a properly typed starting configuration for a given Core ABS program. The configuration consists of a starting object with a process executing the statements in the main block.

Lemma 2. *Let* $\overline{Dd}\;\overline{F}\;\overline{IF}\;\overline{CL}\;\{T_1\,x_1;..;T_n x_n\,;\,\overline{s}\}$ *be a Core ABS program, and let* `value` *be a function that returns the default value for a ground type. If* $\Gamma \vdash \overline{Dd}\;\overline{F}\;\overline{IF}\;\overline{CL}\;\{T_1\,x_1;..;T_n x_n\,;\,\overline{s}\}$ *for some typing context* Γ*, then* $\Gamma \vdash_R \mathsf{ob}\,(\textbf{start},\varepsilon,\{T_1\,x_1\,\texttt{value}(T_1),..,T_n\,x_n\,\texttt{value}(T_n) \mid \overline{s}\},\emptyset)\,\textbf{ok}$.

Proof. Let $\Gamma' = \Gamma[x_1 \mapsto T_1,..,x_n \mapsto T_n]$. Then $\Gamma' \vdash_R T_1\,x_1\,\texttt{value}(T_1),..,T_n\,x_n\,\texttt{value}(T_n)\,\textbf{ok}$ and hence $\Gamma \vdash_R \mathsf{ob}\,(\textbf{start},\varepsilon,\{T_1\,x_1\,\texttt{value}(T_1),..,T_n\,x_n\,\texttt{value}(T_n) \mid \overline{s}\},\emptyset)\,\textbf{ok}$ by T-OBJECT. □

Theorem 1 states that the standard Core ABS semantics preserves well-typing. The theorem implies, among other things, that method invocations in Core ABS cannot go wrong at runtime for type-checked programs; when an object makes a call to a method m using an object identifier o, there always exists an object associated with o, which is an instance of a class where m is defined. Hence, in an execution of a well-typed program beginning from its starting configuration, no object gets stuck trying to execute the **error** process.

Theorem 1 (Subject Reduction)**.** *Let Δ be a typing context and cn a runtime configuration. If $\Delta \vdash_R cn$ **ok** and $cn \to cn'$, then there exists a typing context Δ' such that $\Delta \subseteq \Delta'$ and $\Delta' \vdash_R cn'$ **ok**.*

Proof. By induction on rule applications [7]. □

5 ABS-NET Operational Semantics of Core ABS

The ABS-NET operational semantics of Core ABS is intended to capture decentralized ABS program execution on nodes connected point-to-point with asynchronous message passing links. Conceptually, a node consists of an interpreter layer, where local objects reside, and a node controller, which acts as a mediator between the environment and node-local objects. The semantics remains silent on the inner workings of the node controller other than that it maintains a routing table to track object locations, allowing a wide range of (possibly adaptability-oriented) behaviour through inherent nondeterminism.

5.1 Runtime Configurations

The runtime syntax of ABS-NET is shown in Figure 11. A network *net* consists of nodes and arcs, composed with the whitespace operator, with ε the empty network. In a node $\mathsf{nd}(u,\tau)$, u is a node identifier (assumed globally unique) and τ is a routing table, used to route object-related messages in the proper direction. In an arc $\mathsf{ar}(u,Q,u')$, representing a unidirectional link from u to u', Q is a FIFO-ordered queue of messages *msg*.

cn	::=	$\varepsilon \mid object \mid cn\ cn$	*net*	::=	$\varepsilon \mid node \mid arc \mid net\ net$
object	::=	$\mathsf{ob}(o,a,pr,q,Q_{in},Q_{out},\Sigma)$	*node*	::=	$\mathsf{nd}(u,\tau)$
a,l	::=	$\varepsilon \mid T\ x\ v \mid a,a$	*arc*	::=	$\mathsf{ar}(u,Q,u)$
process	::=	$\{l \mid \overline{sp}\} \mid$ **error**	q	::=	$\emptyset \mid process \mid q\ q$
msg	::=	$\textsc{Call}(o,o,f,m,\overline{v}) \mid \textsc{Future}(o,f,v)$	*sp*	::=	$\mathbf{return}\,e; \mid \mathbf{cont}(f); \mid s$
		$\mid \textsc{Table}(\tau) \mid \textsc{Object}(object)$	*pr*	::=	$process \mid$ **idle**

Figure 11: ABS-NET runtime syntax.

An object configuration *cn* consists of objects, composed by the whitespace operator, again with ε as the empty configuration. In an object $\mathsf{ob}(o,a,pr,q,Q_{in},Q_{out},\Sigma)$, a is a store for its field types and values, *pr* an active process, q a pool of suspended processes, Q_{in} and Q_{out} input and output queues, and Σ a structure for storing resolved future values and obligations to send future values. Stores, active processes, process statements and process pools are defined in the same way as in the standard semantics in Section 4.

The method invocations and futures present in runtime configurations of the standard semantics can intuitively be said to have been replaced in ABS-NET with call and future messages, which are transported from node to node via arcs. A call message $\textsc{Call}(o,o',f,m,\overline{v})$ consists of the identifier o of the

destination object, the identifier o' of the sender object, the associated future identifier f, method name m and argument list $\bar{v}$. A future value message FUTURE(o,f,v) has the identifier of the destination object o, the future identifier f and the associated resolved value v.

Table and object messages, on the other hand, have no equivalent in the standard semantics. A table message TABLE(τ) is used to pass a routing table τ from one node to another, allowing local routes to be updated with new information from a neighbour. An object message OBJECT$(object)$ contains a complete runtime object *object* and is what facilitates object mobility for network-adaptable process execution.

5.2 Reduction System for Guard Expressions

The standard reduction system for functional expressions, given in Figure 6, is carried over to the ABS-NET semantics unchanged. The rules for guard evaluation in ABS-NET, given in Figure 12, are different, however. The rules highlight how the structure Σ, instead of a configuration *cn*, is queried for the resolved values of futures. Given a future identifier f, $\texttt{valof}\,(f,\Sigma)$ returns a *val*, i.e., either a value v or $\bot$. If the result is a value, the future has been recorded as resolved locally. Note that in the semantics generally, the fact that a future is unresolved locally does not mean the associated method invocation is unfinished—there may be a future message incoming.

$$\frac{\sigma \vdash b \rightsquigarrow \sigma \vdash b'}{\sigma,\Sigma \vdash b \rightsquigarrow \sigma,\Sigma \vdash b'}\ \text{(NET-REDBOOLGUARD)} \qquad \frac{\sigma \vdash e \rightsquigarrow \sigma \vdash f \quad \texttt{valof}\,(f,\Sigma) \neq \bot}{\sigma,\Sigma \vdash e? \rightsquigarrow \sigma,\Sigma \vdash \textbf{True}}\ \text{(NET-REDREPLYGUARD1)}$$

$$\frac{\sigma \vdash e \rightsquigarrow \sigma \vdash f \quad \texttt{valof}\,(f,\Sigma) = \bot}{\sigma,\Sigma \vdash e? \rightsquigarrow \sigma,\Sigma \vdash \textbf{False}}\ \text{(NET-REDREPLYGUARD2)}$$

$$\frac{\sigma,\Sigma \vdash g_1 \rightsquigarrow \sigma,\Sigma \vdash g_1' \quad \sigma,\Sigma \vdash g_2 \rightsquigarrow \sigma,\Sigma \vdash g_2'}{\sigma,\Sigma \vdash g_1 \wedge g_2 \rightsquigarrow \sigma,\Sigma \vdash g_1' \wedge g_2'}\ \text{(NET-REDGUARDS)}$$

Figure 12: ABS-NET reduction rules for guard expressions.

The boolean result of evaluating a guard expression g with respect to a structure Σ and a store a in ABS-NET is written as $[\![g]\!]^{\Sigma}_{a}$. As in the standard semantics, the ground term result of evaluating an expression e with respect to a store a is written as $[\![e]\!]_a$.

5.3 Message Queues

The nature of a FIFO queue Q of messages is specified through three auxiliary functions: `enqueue`, `dequeue` and `first`. $\texttt{enqueue}(Q,msg)$ returns the queue that results when the message *msg* is added to the back of Q. If Q is non-empty, $\texttt{first}(Q)$ returns the message at the front of Q, and $\texttt{dequeue}(Q)$ returns the queue that results when the front message is removed. For brevity, $\texttt{enqueue}(Q,msg) = Q'$ is defined as a relation $Q \xrightarrow{\texttt{enqueue}\,(msg)} Q'$, while the conjunction that $\texttt{first}(Q) = msg$ and $\texttt{dequeue}(Q) = Q'$ is defined as $Q \xrightarrow{\texttt{dequeue}\,(msg)} Q'$. $()$ is the empty queue.

Appropriate encodings of queues in implementations vary significantly depending on the application and environment. For example, when using TCP sockets as arcs, the socket library used will determine the queue characteristics. For the purpose of formal analysis, a simple algebraic list-like encoding suffices.

5.4 Routing Tables

The nature of a routing table is specified through the four auxiliary functions `update`, `next`, `register` and `replace`, and an infix operator $\in$. The auxiliary function `update` takes three arguments: the rout-

ing table τ of the current node, the node identifier u' of the adjacent node, and the routing table τ' of the adjacent node. The function returns a routing table τ'', which incorporates the routes from τ' into τ if appropriate, with the constraint that all such routes must go through the node u'. For brevity, $\texttt{update}(\tau, u', \tau') = \tau''$ is defined as a relation $\tau \xrightarrow{\texttt{update}(\tau', u')} \tau''$. The auxiliary function `next` takes three arguments: the routing table τ of the current node, the object identifier o' of the node we want the next hop for, and the default hop u, which is the identifier of the current node. The function returns the node identifier u' which is the next hop of o' according to the table. The auxiliary function `register` takes four arguments: the routing table τ of the current node, the object identifier o' of the object we want to add a route for, the node identifier u of a neighbour node (usually self) which is the next hop, and a non-negative integer k for the distance to the object (in all instances in the rules, it is 0). The function returns a routing table τ' which incorporates the new route. For brevity, $\texttt{register}(\tau, o', u, k) = \tau'$ is defined as a relation $\tau \xrightarrow{\texttt{register}(o', u, k)} \tau'$. The auxiliary function `replace` takes four arguments (of the same kind as `register`): the routing table τ of the current node, the object identifier o' of the object we want to replace the route for, the node identifier u of a neighbour node which is the next hop, and a natural number k for the distance to the object. The function returns a routing table τ' which has removed any existing routes for o' and added the route given. For brevity, $\texttt{replace}(\tau, o', u, k) = \tau'$ is defined as a relation $\tau \xrightarrow{\texttt{replace}(o, u, k)} \tau'$. The claim $o \in \tau$, with a node identifier u given by the context, means that, according to τ, the object with identifier o is located on the node u.

One example of a relatively simple encoding of a routing table, which can be enough for some implementations, is as a finite map from object identifiers to sets of tuples of node identifiers and distances. A binding in a routing table for the identifier o to the set $\{(u,2),(u',3)\}$ then represents that the next hop for reaching the object o is either in the direction of u, for a total distance of 2 hops, or in the direction of u', for a total distance of 3 hops. Of course, due to object mobility, the routes may not be accurate, but are able to reflect the last known information.

5.5 Operational Semantics of Networks

The global state in ABS-NET consists of a pair $\{net\}\,\{cn\}$. The network part and the object part of the global state can evolve jointly by performing synchronized labelled transitions, but also separately without exchanging information. The rules for such synchronization and separate evolution are shown in Figure 13. A label α is either $\texttt{mv}(object)$ (moving an object), $\texttt{rg}(o, o')$ (registering a new object identifier), or $\texttt{tr}(o, msg)$ (transporting a message). Intuitively, a label with an overline means that information is outgoing or being sent, while a label without overline means information is incoming or being received. Like in the standard semantics, an execution is a sequence global states with valid rule transitions between every adjacent pair.

$$\frac{\{net\} \to \{net'\}}{\{net\}\,\{cn\} \to \{net'\}\,\{cn\}}\ (\textsc{Net-Red}) \qquad \frac{\{cn\} \to \{cn'\}}{\{net\}\,\{cn\} \to \{net\}\,\{cn'\}}\ (\textsc{Cn-Red})$$

$$\frac{\{cn\} \xrightarrow{\overline{\alpha}} \{cn'\} \quad \{net\} \xrightarrow{\alpha} \{net'\}}{\{net\}\,\{cn\} \to \{net'\}\,\{cn'\}}\ (\textsc{Cn-Out-Net-In-Red}) \qquad \frac{\{net\} \xrightarrow{\overline{\alpha}} \{net'\} \quad \{cn\} \xrightarrow{\alpha} \{cn'\}}{\{net\}\,\{cn\} \to \{net'\}\,\{cn'\}}\ (\textsc{Net-Out-Cn-In-Red})$$

Figure 13: ABS-NET reduction rules connecting objects and networks.

The reduction rules for networks are shown in Figure 14. The auxiliary function `id`, used in the rule Net-Object-Recv-Out, takes a runtime object as argument and returns its identifier. The function `dest`, used in the rules Net-Msg-Recv-Out, Net-Msg-Send-In and Net-Route-Further, is defined only

for CALL and FUTURE messages; it returns their first object identifier, which is the identifier of the intended recipient object (destination).

For proper progress in execution, we assume networks are such that (1) there are no dangling arcs referencing non-existent nodes, (2) for every arc between nodes there is an arc in the opposite direction, and (3) every node comes with a self-loop arc, i.e., an arc going from and to the node. Self-loop arcs are important for two reasons. First, it allows us to use the same rules for message passing in both the case where the sender object is at a different node from the receiver object, and where the sender is at the same node as the receiver. Once a message has been put in the self-loop queue, it intuitively appears as if it came from some other node when applying the rule NET-MSG-RECV-OUT. Second, it may not always be the case that there is a route (next hop) to the recipient of a message, because routing tables may not have stabilized. Such a message must be dealt with somehow when there are other important messages pending in that queue after the message. Hence, it is put in the self-loop queue, i.e., the default next hop of an object-addressed message is the node itself.

$$\frac{u' \neq u \quad Q \xrightarrow{\texttt{enqueue}(\text{TABLE}(\tau))} Q'}{\mathsf{nd}(u,\tau)\,\mathsf{ar}(u,Q,u') \to \mathsf{nd}(u,\tau)\,\mathsf{ar}(u,Q',u')} \text{ (NET-TABLE-SEND)}$$

$$\frac{Q \xrightarrow{\texttt{dequeue}(\text{TABLE}(\tau'))} Q' \quad \tau \xrightarrow{\texttt{update}(\tau',u')} \tau''}{\mathsf{ar}(u',Q,u)\,\mathsf{nd}(u,\tau) \to \mathsf{ar}(u',Q',u)\,\mathsf{nd}(u,\tau'')} \text{ (NET-TABLE-RECV)}$$

$$\frac{Q \xrightarrow{\texttt{dequeue}(msg)} Q' \quad \texttt{dest}(msg) = o \quad o \in \tau}{\mathsf{ar}(u',Q,u)\,\mathsf{nd}(u,\tau) \xrightarrow{\texttt{tr}(o,msg)} \mathsf{ar}(u',Q',u)\,\mathsf{nd}(u,\tau)} \text{ (NET-MSG-RECV-OUT)}$$

$$\frac{o \in \tau \quad \texttt{dest}(msg) = o' \quad \texttt{next}(\tau,o',u) = u' \quad Q \xrightarrow{\texttt{enqueue}(msg)} Q'}{\mathsf{nd}(u,\tau)\,\mathsf{ar}(u,Q,u') \xrightarrow{\texttt{tr}(o,msg)} \mathsf{nd}(u,\tau)\,\mathsf{ar}(u,Q',u')} \text{ (NET-MSG-SEND-IN)}$$

$$\frac{Q_1 \xrightarrow{\texttt{dequeue}(msg)} Q_1' \quad \texttt{dest}(msg) = o \quad o \notin \tau \quad \texttt{next}(\tau,o,u) = u'' \quad Q_2 \xrightarrow{\texttt{enqueue}(msg)} Q_2'}{\mathsf{ar}(u',Q_1,u)\,\mathsf{nd}(u,\tau)\,\mathsf{ar}(u,Q_2,u'') \to \mathsf{ar}(u',Q_1',u)\,\mathsf{nd}(u,\tau)\,\mathsf{ar}(u,Q_2',u'')} \text{ (NET-ROUTE-FURTHER)}$$

$$\frac{o \in \tau \quad u' \neq u \quad \tau \xrightarrow{\texttt{replace}(o,u',1)} \tau' \quad Q \xrightarrow{\texttt{enqueue}(\text{OBJECT}(object))} Q'}{\mathsf{nd}(u,\tau)\,\mathsf{ar}(u,Q,u') \xrightarrow{\texttt{mv}(object)} \mathsf{nd}(u,\tau')\,\mathsf{ar}(u,Q',u')} \text{ (NET-OBJECT-SEND-IN)}$$

$$\frac{\texttt{id}(object) = o \quad Q \xrightarrow{\texttt{dequeue}(\text{OBJECT}(object))} Q' \quad \tau \xrightarrow{\texttt{replace}(o,u,0)} \tau'}{\mathsf{ar}(u',Q,u)\,\mathsf{nd}(u,\tau) \xrightarrow{\texttt{mv}(object)} \mathsf{ar}(u',Q',u)\,\mathsf{nd}(u,\tau')} \text{ (NET-OBJECT-RECV-OUT)}$$

$$\frac{\texttt{fresh}(o') \quad o \in \tau \quad \tau \xrightarrow{\texttt{register}(o',u,0)} \tau'}{\mathsf{nd}(u,\tau) \xrightarrow{\texttt{rg}(o,o')} \mathsf{nd}(u,\tau')} \text{ (NET-NEW-OBJECT-IN)}$$

Figure 14: ABS-NET node controller reduction rules.

The property of an object being located on a node is represented indirectly through the rules, not explicitly in runtime configurations. The labelled transition rules NET-MSG-RECV-OUT, NET-MSG-SEND-IN, NET-OBJECT-SEND-IN and NET-NEW-OBJECT-IN, where a node exchanges information with an object, all use the premise $o \in \tau$ to restrict actions to pertain to node-local objects.

5.6 Future Value Distribution

In the ABS-NET semantics, future values are transmitted through messages to the objects which need them, as opposed to the centralized future access in the standard semantics. However, as described by Henrio et al. [6], there are several fundamentally different ways of propagating futures to objects, with different trade-offs in performance and resource usage. The ABS-NET semantics uses what Henrio et al. refer to as an eager forward-based strategy, where an object o that shares a future identifier f with

another object o' is obligated to forward the value of f to o' when this becomes possible. This strategy is relatively easy to implement and distributes the load of messaging related to futures over many objects, which in balanced object-node allocations translates to many nodes.

A number of auxiliary functions in the ABS-NET reduction rules for objects take a structure Σ as input and either retrieve data from it or produce a modified structure in order to accomplish future forwarding. They are reminiscent of the operations modelled by Henrio et al., but have several properties specific to the ABS setting. The function `recsof` takes a future identifier f and a structure as input, and returns a set of object identifiers; intuitively, this set contains the identifiers of the objects which are the intended recipients of the value of f. The function `sendfuts` takes a set of future identifiers, an object identifier, and a structure, and returns another structure, interpreted as the given structure updated with obligations to forward the values of all indicated futures to the indicated object. For all structures Σ and all future identifiers $f \in \{f_1, \ldots, f_n\}$, it holds that $o \in \texttt{recsof}\,(f, \texttt{sendfuts}\,(\{f_1, \ldots, f_n\}, o, \Sigma))$. The function `clrrec`, which takes an object identifier o, a future identifier f and a structure Σ, returns an updated structure Σ' where o has been cleared from the recipient set of f, i.e., $o \notin \texttt{recsof}\,(f, \Sigma')$. The function `regfuts` takes a set of future identifiers and a structure Σ, and returns an updated structure Σ' where the given futures are registered, meaning that the futures are associated through Σ' with a *val* term and a set of object identifiers. If a future f has no such associations in Σ, it is the case that $\texttt{valof}\,(f, \Sigma') = \bot$ and $\texttt{recsof}\,(f, \Sigma') = \emptyset$; otherwise, the associations are the same as in Σ. Finally, the function `resfut` takes a future identifier f, a value v and a structure Σ, and returns an updated structure Σ' where v is recorded as the resolved value for f, i.e., $\texttt{valof}\,(f, \Sigma') = v$.

Most straightforwardly, a structure Σ can be encoded as a pair $\langle M_{\text{VAL}}, M_{\text{REC}} \rangle$, where M_{VAL} is a finite map from future identifiers to *val* terms, and M_{REC} is a finite map from future identifiers to sets of object identifiers. The auxiliary functions are then defined as map operations, e.g., $\texttt{valof}(f, \langle M_{\text{VAL}}, M_{\text{REC}} \rangle) \triangleq M_{\text{VAL}}(f)$ and $\texttt{recsof}(f, \langle M_{\text{VAL}}, M_{\text{REC}} \rangle) \triangleq M_{\text{REC}}(f)$.

5.7 Operational Semantics of Concurrent Objects

The labelled rules for object configurations are shown in Figure 15. The `init` and `atts` auxiliary functions constructs the initial task of the object as given in the corresponding class definition, and initializes variables based on given arguments, respectively, and are unchanged from Core ABS. The function `futsof` returns the set of all future identifiers in a given value, if necessary by recursively examining algebraic datatype terms. $[\,]$ is the empty structure of future values and obligations.

$$\frac{}{\{object\ cn\} \overset{\overline{\texttt{mv}\,(object)}}{\rightarrow} \{cn\}}\ \text{(ABS-Object-Send-Out)}$$

$$\frac{Q_{out} \xrightarrow{\texttt{dequeue}\,(msg)} Q'_{out}}{\overset{\overline{\texttt{tr}\,(o,msg)}}{\rightarrow} \texttt{ob}\,(o,a,pr,q,Q_{in},Q_{out},\Sigma)\ \texttt{ob}\,(o,a,pr,q,Q_{in},Q'_{out},\Sigma)}\ \text{(ABS-Msg-Send-Out)}$$

$$\frac{Q_{in} \xrightarrow{\texttt{enqueue}\,(msg)} Q'_{in}}{\overset{\texttt{tr}\,(o,msg)}{\rightarrow} \texttt{ob}\,(o,a,pr,q,Q_{in},Q_{out},\Sigma)\ \texttt{ob}\,(o,a,pr,q,Q'_{in},Q_{out},\Sigma)}\ \text{(ABS-Msg-Recv-In)}$$

$$\frac{}{\{cn\} \overset{\texttt{mv}\,(object)}{\rightarrow} \{object\ cn\}}\ \text{(ABS-Object-Recv-In)}$$

$$\frac{\texttt{init}\,(C) = process \quad [\![\overline{e}]\!]_{a \circ l} = \overline{v} \quad \texttt{sendfuts}\,(\texttt{futsof}\,(\overline{v}), o', \Sigma) = \Sigma' \quad \texttt{atts}\,(C, \overline{v}, o') = a' \quad \texttt{regfuts}\,(\texttt{futsof}\,(\overline{v}), [\,]) = \Sigma''}{\overset{\overline{\texttt{rg}\,(o,o')}}{\rightarrow} \texttt{ob}\,(o,a,\{l \mid x = \textbf{new}\,C(\overline{e}); \overline{sp}\},q,Q_{in},Q_{out},\Sigma)\ \texttt{ob}\,(o,a,\{l \mid x = o'; \overline{sp}\},q,Q_{in},Q_{out},\Sigma')\ \texttt{ob}\,(o',a',\textbf{idle},process,(),(),\Sigma'')}\ \text{(ABS-New-Object-Out)}$$

Figure 15: ABS-NET object reduction rules for node controller interaction.

For object creation, both the network rule Net-New-Object-In and object rule ABS-New-Object-Out need to be involved. When such a synchronized transition has taken place, the new object has been

properly added to the interpreter layer, and its globally unique identifier registered on the node of the object that spawned it. Given that we abstract from details on marshalling and pass object states directly in messages, the rules for object mobility, ABS-OBJECT-SEND-OUT and ABS-OBJECT-RECV-IN, which interact with the rules NET-OBJECT-SEND-IN and NET-OBJECT-RECV-OUT above, are straightforward. The rules ABS-MSG-SEND-OUT and ABS-MSG-RECV-IN for passing messages back and forth with the node controller are uncomplicated since eligible messages have been put in the out queue of the object.

The remaining rules for object transitions, that do not involve information exchange with a node via a label, are given in Figure 16 and Figure 17. In ABS-ACTIVATE, the auxiliary function `select` takes two parameters: the local pool of processes q and the object store a, skipping the configuration parameter in the corresponding function in the standard semantics. The reason is that scheduling in ABS-NET is assumed to take only local information into account. In ABS-REM-SYNC-CALL, the premise $o' \neq o$ is an addition when compared to the standard semantics counterpart rule. This premise is used to preclude synchronous self calls from being dispatched asynchronously, causing a deadlock. In the Core ABS semantics, such deadlocks are ruled out by the presence of the other runtime object in the rule's left-hand side. In ABS-NET, the intent is for all rules to be implementable directly on a single node, which implies it is not possible to depend on both the caller and callee objects being present locally.

(ABS-SKIP)
$$\frac{}{\mathsf{ob}(o,a,\{l \mid \mathbf{skip};\overline{sp}\},q,Q_{in},Q_{out},\Sigma) \to \mathsf{ob}(o,a,\{l \mid \overline{sp}\},q,Q_{in},Q_{out},\Sigma)}$$

(ABS-ASSIGN-LOCAL)
$$\frac{x \in \mathtt{dom}(l) \quad [\![e]\!]_{a\circ l} = v}{\mathsf{ob}(o,a,\{l \mid x = e;\overline{sp}\},q,Q_{in},Q_{out},\Sigma) \to \mathsf{ob}(o,a,\{l[x \mapsto v] \mid \overline{sp}\},q,Q_{in},Q_{out},\Sigma)}$$

(ABS-ASSIGN-FIELD)
$$\frac{x \in \mathtt{dom}(a) \quad [\![e]\!]_{a\circ l} = v}{\mathsf{ob}(o,a,\{l \mid x = e;\overline{sp}\},q,Q_{in},Q_{out},\Sigma) \to \mathsf{ob}(o,a[x \mapsto v],\{l \mid \overline{sp}\},q,Q_{in},Q_{out},\Sigma)}$$

(ABS-COND-TRUE)
$$\frac{[\![b]\!]_{a\circ l} = \mathbf{True}}{\mathsf{ob}(o,a,\{l \mid \mathbf{if}\, b\{\overline{s}\}\,[\mathbf{else}\,\{\overline{s'}\}]\,\overline{sp}\},q,Q_{in},Q_{out},\Sigma) \to \mathsf{ob}(o,a,\{l \mid \overline{s}\;\overline{sp}\},q,Q_{in},Q_{out},\Sigma)}$$

(ABS-COND-FALSE)
$$\frac{[\![b]\!]_{a\circ l} = \mathbf{False}}{\mathsf{ob}(o,a,\{l \mid \mathbf{if}\, b\{\overline{s}\}\,[\mathbf{else}\,\{\overline{s'}\}]\,\overline{sp}\},q,Q_{in},Q_{out},\Sigma) \to \mathsf{ob}(o,a,\{l \mid [\overline{s'}]\,\overline{sp}\},q,Q_{in},Q_{out},\Sigma)}$$

(ABS-SUSPEND)
$$\frac{}{\mathsf{ob}(o,a,\{l \mid \mathbf{suspend};\overline{sp}\},q,Q_{in},Q_{out},\Sigma) \to \mathsf{ob}(o,a,\mathbf{idle},q \cup \{l \mid \overline{sp}\},Q_{in},Q_{out},\Sigma)}$$

(ABS-AWAIT-TRUE)
$$\frac{[\![g]\!]^{\Sigma}_{a\circ l} = \mathbf{True}}{\mathsf{ob}(o,a,\{l \mid \mathbf{await}\,g;\overline{sp}\},q,Q_{in},Q_{out},\Sigma) \to \mathsf{ob}(o,a,\{l \mid \overline{sp}\},q,Q_{in},Q_{out},\Sigma)}$$

(ABS-AWAIT-FALSE)
$$\frac{[\![g]\!]^{\Sigma}_{a\circ l} = \mathbf{False}}{\mathsf{ob}(o,a,\{l \mid \mathbf{await}\,g;\overline{sp}\},q,Q_{in},Q_{out},\Sigma) \to \mathsf{ob}(o,a,\{l \mid \mathbf{suspend};\mathbf{await}\,g;\overline{sp}\},q,Q_{in},Q_{out},\Sigma)}$$

(ABS-ASYNC-CALL-SEND)
$$\frac{[\![e]\!]_{a\circ l} = o' \quad [\![\overline{e}]\!]_{a\circ l} = \overline{v} \quad \mathtt{fresh}(f) \quad \mathtt{regfuts}(\{f\},\mathtt{sendfuts}(\mathtt{futsof}(\overline{v}),o',\Sigma)) = \Sigma' \quad Q_{out} \xrightarrow{\mathtt{enqueue}(\mathrm{CALL}(o',o,f,m,\overline{v}))} Q'_{out}}{\mathsf{ob}(o,a,\{l \mid x = e!m(\overline{e});\overline{sp}\},q,Q_{in},Q_{out},\Sigma) \to \mathsf{ob}(o,a,\{l \mid x = f;\overline{sp}\},q,Q_{in},Q'_{out},\Sigma')}$$

(ABS-ASYNC-CALL-RECV)
$$\frac{Q_{in} \xrightarrow{\mathtt{dequeue}(\mathrm{CALL}(o,o',f,m,\overline{v}))} Q'_{in} \quad \mathtt{sendfuts}(\{f\},o',\mathtt{regfuts}(\mathtt{futsof}(\overline{v}) \cup \{f\},\Sigma)) = \Sigma' \quad \mathtt{bind}(o,f,m,\overline{v},\mathtt{class}(o)) = process}{\mathsf{ob}(o,a,pr,q,Q_{in},Q_{out},\Sigma) \to \mathsf{ob}(o,a,pr,q \cup process,Q'_{in},Q_{out},\Sigma')}$$

Figure 16: ABS-NET object reduction rules (1).

An ABS-NET global starting state is given by a network configuration having the properties described in Section 5.5, and an object configuration containing a single starting object as in Lemma 2, extended with empty queues and an empty structure of futures and obligations.

$$\frac{\texttt{select}(q,a) = process}{\mathsf{ob}(o,a,\mathbf{idle},q,Q_{in},Q_{out},\Sigma) \to \mathsf{ob}(o,a,process,q \setminus process,Q_{in},Q_{out},\Sigma)}\ (\text{ABS-ACTIVATE})$$

$$\frac{l'(\mathbf{destiny}) = f}{\mathsf{ob}(o,a,\{l \mid \mathbf{cont}(f);\} \cup q,Q_{in},Q_{out},\Sigma) \to \mathsf{ob}(o,a,\{l' \mid \overline{sp}\},q,Q_{in},Q_{out},\Sigma)}\ (\text{ABS-SELF-SYNC-RETURN-SCHED})$$

$$\frac{[\![e]\!]_{a \circ l} = v \quad l(\mathbf{destiny}) = f \quad \texttt{resfut}(f,v,\Sigma) = \Sigma'}{\mathsf{ob}(o,a,\{l \mid \mathbf{return}\, e; \overline{sp}\},q,Q_{in},Q_{out},\Sigma) \to \mathsf{ob}(o,a,\{l \mid \overline{sp}\},q,Q_{in},Q_{out},\Sigma')}\ (\text{ABS-RETURN})$$

$$\frac{l(\mathbf{destiny}) = f' \quad [\![e]\!]_{a \circ l} = o \quad [\![\overline{e}]\!]_{a \circ l} = \overline{v} \quad \texttt{fresh}(f) \quad \texttt{regfuts}(\{f\},\Sigma) = \Sigma' \quad \texttt{bind}(o,f,m,\overline{v},\texttt{class}(o)) = \{l' \mid \overline{sp}'\}}{\mathsf{ob}(o,a,\{l \mid x = e.m(\overline{e}); \overline{sp}\},q,Q_{in},Q_{out},\Sigma) \to \mathsf{ob}(o,a,\{l' \mid \overline{sp}'\ \mathbf{cont}(f');\},q \cup \{l \mid x = f.\mathbf{get}; \overline{sp}\},Q_{in},Q_{out},\Sigma')}\ (\text{ABS-SELF-SYNC-CALL})$$

$$\frac{Q_{in} \xrightarrow{\texttt{dequeue}(\text{FUTURE}(o,f,v))} Q'_{in} \quad \texttt{resfut}(f,v,\Sigma) = \Sigma'}{\mathsf{ob}(o,a,pr,q,Q_{in},Q_{out},\Sigma) \to \mathsf{ob}(o,a,pr,q,Q'_{in},Q_{out},\Sigma')}\ (\text{ABS-FUTURE-RECV})$$

$$\frac{Q_{out} \xrightarrow{\texttt{enqueue}(\text{FUTURE}(o',f,v))} Q'_{out} \quad \texttt{valof}(f,\Sigma) = v \quad o' \in \texttt{recsof}(f,\Sigma) \quad \texttt{sendfuts}(\texttt{futsof}(v),o',\texttt{clrrec}(o',f,\Sigma)) = \Sigma'}{\mathsf{ob}(o,a,pr,q,Q_{in},Q_{out},\Sigma) \to \mathsf{ob}(o,a,pr,q,Q_{in},Q'_{out},\Sigma')}\ (\text{ABS-FUTURE-SEND})$$

$$\frac{[\![e]\!]_{a \circ l} = f \quad \texttt{valof}(f,\Sigma) = v}{\mathsf{ob}(o,a,\{l \mid x = e.\mathbf{get}; \overline{sp}\},q,Q_{in},Q_{out},\Sigma) \to \mathsf{ob}(o,a,\{l \mid x = v; \overline{sp}\},q,Q_{in},Q_{out},\Sigma)}\ (\text{ABS-READ-FUT})$$

$$\frac{}{\mathsf{ob}(o,a,\{l \mid \},q,Q_{in},Q_{out},\Sigma) \to \mathsf{ob}(o,a,\mathbf{idle},q,Q_{in},Q_{out},\Sigma)}\ (\text{ABS-IDLE})$$

$$\frac{[\![b]\!]_{a \circ l} = \mathbf{True}}{\mathsf{ob}(o,a,\{l \mid \mathbf{while}\, b\, \{\overline{s}\}\, \overline{sp}\},q,Q_{in},Q_{out},\Sigma) \to \mathsf{ob}(o,a,\{l \mid \overline{s}\ \mathbf{while}\, b \{\overline{s}\}\, \overline{sp}\},q,Q_{in},Q_{out},\Sigma)}\ (\text{ABS-WHILE-TRUE})$$

$$\frac{[\![b]\!]_{a \circ l} = \mathbf{False}}{\mathsf{ob}(o,a,\{l \mid \mathbf{while}\, b\, \{\overline{s}\}\, \overline{sp}\},q,Q_{in},Q_{out},\Sigma) \to \mathsf{ob}(o,a,\{l \mid \overline{sp}\},q,Q_{in},Q_{out},\Sigma)}\ (\text{ABS-WHILE-FALSE})$$

$$\frac{[\![e]\!]_{a \circ l} = o' \quad o' \neq o \quad \texttt{fresh}(y) \quad \texttt{returns}(\texttt{class}(o'),m) = T}{\mathsf{ob}(o,a,\{l \mid x = e.m(\overline{e}); \overline{sp}\},q,Q_{in},Q_{out},\Sigma) \to \mathsf{ob}(o,a,\{l, \mathbf{Fut}\langle T\rangle\, y\, \mathbf{null} \mid y = o'!m(\overline{e}); x = y.\mathbf{get}; \overline{sp}\},q,Q_{in},Q_{out},\Sigma)}\ (\text{ABS-REM-SYNC-CALL})$$

Figure 17: ABS-NET object reduction rules (2).

References

[1] Elvira Albert, Samir Genaim, Miguel Gómez-Zamalloa, Einar Broch Johnsen, Rudolf Schlatte & S. Lizeth Tapia Tarifa (2011): *Simulating Concurrent Behaviors with Worst-Case Cost Bounds*. In Michael Butler & Wolfram Schulte, editors: *FM 2011: Formal Methods*, *Lecture Notes in Computer Science* 6664, Springer Berlin Heidelberg, pp. 353–368, doi:10.1007/978-3-642-21437-0_27.

[2] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer & J. F. Quesada (2002): *Maude: specification and programming in rewriting logic*. *Theoretical Computer Science* 285(2), pp. 187–243, doi:10.1016/S0304-3975(01)00359-0.

[3] Mads Dam & Karl Palmskog (2013): *Efficient and Fully Abstract Routing of Futures in Object Network Overlays*. Available at `http://www.csc.kth.se/~palmskog/publications/efarfono.pdf`. Manuscript, submitted for publication.

[4] Mads Dam & Karl Palmskog (2013): *Location Independent Routing in Process Network Overlays*. Available at `http://www.csc.kth.se/~palmskog/publications/lirpno.pdf`. Manuscript, submitted for publication.

[5] FP7-231620 (HATS) Project (2011): *Deliverable 1.2: Full ABS Modeling Framework*. Available at `http://www.hats-project.eu/sites/default/files/Deliverable12.pdf`.

[6] Ludovic Henrio, Muhammad Uzair Khan, Nadia Ranaldo & Eugenio Zimeo (2011): *First Class Futures: Specification and Implementation of Update Strategies*. In Mario R. Guarracino, Vivien Frédéric, Jesper Larsson Träff, Mario Cannatoro, Marco Danelutto, Anders Hast, Francesca Perla, Andreas Knüpfer, Beniamino Martino & Michael Alexander, editors: *Euro-Par 2010 Parallel Processing Workshops*, *Lecture Notes in Computer Science* 6586, Springer Berlin Heidelberg, pp. 295–303, doi:10.1007/978-3-642-21878-1_37.

[7] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte & Martin Steffen (2012): *ABS: A Core Language for Abstract Behavioral Specification*. In Bernhard K. Aichernig, Frank S. de Boer & Marcello M. Bonsangue, editors: *Formal Methods for Components and Objects*, *Lecture Notes in Computer Science* 6957, Springer Berlin Heidelberg, pp. 142–164, doi:10.1007/978-3-642-25271-6_8.

[8] Karl Palmskog, Mads Dam, Andreas Lundblad & Ali Jafari (2013): *ABS-NET: Fully Decentralized Runtime Adaptation for Distributed Objects*. In: *Proceedings of 6th Interaction and Concurrency Experience*. To appear.