Noname manuscript No.
(will be inserted by the editor)

A Framework for Deadlock Detection in `core ABS`

Elena Giachino · Cosimo Laneve · Michael Lienhardt

Received: date / Accepted: date

Abstract We present a framework for statically detecting deadlocks in a concurrent object-oriented language with asynchronous method calls and cooperative scheduling of method activations. Since this language features recursion and dynamic resource creation, deadlock detection is extremely complex and state-of-the-art solutions either give imprecise answers or do not scale.

In order to augment precision and scalability we propose a modular framework that allows several techniques to be combined. The basic component of the framework is a front-end inference algorithm that extracts abstract behavioural descriptions of methods, called contracts, which retain resource dependency information. This component is integrated with a number of possible different back-ends that analyse contracts and derive deadlock information. As a proof-of-concept, we discuss two such back-ends: (i) an evaluator that computes a fixpoint semantics and (ii) an evaluator using abstract model checking.

1 Introduction

Modern systems are designed to support a high degree of parallelism by letting as many system components as possible operate concurrently. When such systems also exhibit a high degree of resource and data sharing then deadlocks represent an insidious and recurring threat. In particular, deadlocks arise as a consequence of exclusive resource access and circular wait for accessing resources. A standard example is when two processes are exclusively holding a different resource and are requesting access to the resource held by the other. That is, the correct termination of each of the two process activities *depends* on the termination of the other. The presence of a *circular dependency* makes termination impossible.

Deadlocks may be particularly hard to detect in systems with unbounded (mutual) recursion and dynamic resource creation. A paradigm case is an adaptive system that creates an unbounded number of processes such as server applications. In these systems, the interaction protocols are extremely complex and state-of-the-art solutions either give imprecise answers or do not scale – see Section 8 and, for instance, [32] and the references therein.

In order to augment precision and scalability we propose a modular framework that allows several techniques to be combined. We meet scalability requirement by designing a front-end inference system that automatically extracts abstract behavioural descriptions pertinent to deadlock analysis, called *contracts*, from code. The inference system is *modular* because it (partially) supports separate inference of modules. To meet precision of contracts' analysis, as a proof-of-concept we define and implement two different techniques: (i) an evaluator that computes a fixpoint semantics and (ii) an evaluator using abstract model checking.

Our framework targets `core ABS` [23], which is an abstract, executable, object-oriented modelling language with a formal semantics, targeting distributed systems. In `core ABS`, method invocations are asynchronous: the caller continues after the invocation and the called code runs on a different task. Tasks are cooperatively scheduled, that is there is a notion of group of objects, called *cog*, and there is at most one active task at each time per cog. The active task explicitly returns the control in

Partly funded by the EU project FP7-610582 ENVISAGE: Engineering Virtualized Services.

Department of Computer Science and Engineering, University of Bologna – INRIA Focus Team, Italy

order to let other tasks progress. The synchronisation between the caller and the called methods is performed when the result is strictly necessary [6, 24, 40]. Technically, the decoupling of method invocation and the returned value is realised using *future variables* (see [3] and the references in there), which are pointers to values that may be not available yet. Clearly, the access to values of future variables may require waiting for the value to be returned. We discuss the syntax and the semantics of `core ABS`, in Section 2.

Because of the presence of explicit synchronisation operations, the analysis of deadlocks in `core ABS` is more fine-grained than in thread-based languages (such as `Java`). However, as usual with (concurrent) programming languages, analyses are hard and time-consuming because most part of the code is irrelevant for the properties one intends to derive. For this reason, in Section 4, we design an inference system that *automatically extracts contracts* from `core ABS` code. These contracts are similar to those ranging from languages for session types [14] to process contracts [29] and to calculi of processes as Milner's CCS or pi-calculus [30, 31]. The inference system mostly collects method behaviours and uses constraints to enforce consistencies among behaviours. Then a standard semiunification technique is used for solving the set of generated constraints.

Since our inference system addresses a language with asynchronous method invocations, it is possible that a method triggers behaviours that will last *after* its lifetime (and therefore will contribute to *future* deadlocks). In order to support a more precise analysis, we split contracts of methods in *synchronised* and *unsynchronised contracts*, with the intended meaning that the formers collect the invocations that are explicitly synchronised in the method body and the latter ones collect the other invocations.

The current release of the inference system does not cover the full range of features of `core ABS`. In Section 3 we discuss the restrictions of `core ABS` and the techniques that may be used to remove these restrictions.

Our contracts feature recursion and resource creation; therefore their underlying models are infinite states and their analysis cannot be exhaustive. We propose two techniques for analysing contracts (and to show the modularity of our framework). The first one, which is discussed in Section 5, is a fixpoint technique on models with a limited capacity of name creation. This entails fixpoint existence and finiteness of models. While we lose precision, our technique is sound (in some cases, this technique may signal false positives). The second technique, which is detailed in Section 6, is an abstract model checking that evaluates the contract program up-to some point, which is possible to determine by analysing the recursive patterns of the program. This technique is precise when the recursions are linear, while it is over-approximating in general.

We have prototyped an implementation of our framework, called the `DF4ABS` tool and, in Section 7, we assess the precision and performance of the prototype. In particular, we have applied it to an industrial case study that is based on the Fredhopper Access Server (FAS) developed by SDL Fredhopper[1]. It is worth to recall that, because of the modularity of `DF4ABS`, the current analyses techniques may be integrated and/or replaced by other ones. We discuss this idea in Section 9.

Origin of the material. The basic ideas of this article have appeared in conference proceedings. In particular, the contract language and (a simplified form of) the inference system have been introduced in [15, 17], while the fixpoint analysis technique has been explored in [15] and an introduction to the abstract model checking technique can be found in [18], while the details are in [19]. This article is a thoroughly revised and enhanced version of [15] that presents the whole framework in a uniform setting and includes the full proofs of all the results. A more detailed comparison with other related work is postponed to Section 8.

2 The language `core ABS`

The syntax and the semantics (of the concurrent object level) of `core ABS` are defined in the following two subsections; the third subsection is devoted to the discussion of examples, and the last one to the definition of deadlock. In this contribution we overlook the functional level of `core ABS` that defines data types and functions because their analysis can be performed with techniques that may (easily) complement those discussed in this paper (such as data-flow analysis). Details of `core ABS`, its semantics and its standard type system can be also found in [23].

2.1 *Syntax*

Figure 1 displays `core ABS` syntax, where an overlined element corresponds to any finite sequence of such element. The elements of the sequence are separated by commas, except for $\overline{C}$, which has no separator. For example $\overline{T}$ means a (possibly empty) sequence $T_1, \cdots, T_n$. When we write $\overline{T\ x\ ;}$ we mean a sequence $T_1\ x_1\ ;\ \cdots\ ;\ T_n\ x_n\ ;$ when the sequence is not empty; we mean the empty sequence otherwise.

[1] `http://sdl.com/products/fredhopper/`

P	$::= \overline{I}\ \overline{C}\ \{\overline{T\ x\ ;}\ s\}$	program
T	$::=$ `D` $\mid$ `Fut<`T`>` $\mid$ `I`	type
I	$::=$ `interface I` $\{\overline{S\ ;}\}$	interface
S	$::= T$ `m`$(\overline{T\ x})$	method signature
C	$::=$ `class C`$(\overline{T\ x})$ [`implements` $\overline{\texttt{I}}$] $\{\overline{T\ x\ ;}\ \overline{M}\}$	class
M	$::= S\{\overline{T\ x\ ;}\ s\}$	method definition
s	$::=$ `skip` $\mid x = z \mid$ `if` $e\ \{s\}$ `else` $\{s\} \mid$ `return` $e \mid s\ ;\ s \mid$ `await` e`?`	statement
z	$::= e \mid e$`.m`$(\overline{e}) \mid e$`!m`$(\overline{e}) \mid$ `new C` $(\overline{e}) \mid$ `new cog C` $(\overline{e}) \mid e$`.get`	expression with side effects
e	$::= v \mid x \mid$ `this` $\mid$ *arithmetic-and-bool-exp*	expression
v	$::=$ `null` $\mid$ *primitive values*	value

Fig. 1 The language `core ABS`

A program P is a list of interface and class declarations (resp. I and C) followed by a *main function* $\{\overline{T\ x\ ;}\ s\}$. A type T is the name of either a primitive type `D` such as `Int`, `Bool`, `String`, or a *future type* `Fut<`T`>`, or an interface name `I`.

A class declaration `class C`$(\overline{T\ x})$ $\{\overline{T'\ x'\ ;}\ \overline{M}\}$ has a name `C` and declares its fields $\overline{T\ x}, \overline{T'\ x'}$ and its methods M. The fields $\overline{T\ x}$ will be initialised when the object is created; the fields $\overline{T'\ x'}$ will be initialised by the main function of the class (or by the other methods).

A statement s may be either one of the standard operations of an imperative language or one of the operations for scheduling. This operation is `await` x`?` (the other one is `get`, see below), which suspends method's execution until the argument x, is resolved. This means that `await` requires the value of x to be resolved before resuming method's execution.

An expression z may have side effects (may change the state of the system) and is either an object creation `new C`$(\overline{e})$ in the same group of the creator or an object creation `new cog C`$(\overline{e})$ in a new group. In `core ABS`, (runtime) objects are partitioned in groups, called *cogs*, which own a lock for regulating the executions of threads. Every threads acquires its own cog lock in order to be evaluated and releases it upon termination or suspension. Clearly, threads running on different cogs may be evaluated in parallel, while threads running on the same cog do compete for the lock and interleave their evaluation. The two operations `new C`$(\overline{e})$ and `new cog C`$(\overline{e})$ allow one to add an object to a previously created cog or to create new singleton cogs, respectively.

An expression z may also be either a (synchronous) method call e`.m`$(\overline{e})$ or an *asynchronous* method call e`!m`$(\overline{e})$. Synchronous method invocations suspend the execution of the caller, without releasing the lock of the corresponding cog; asynchronous method invocations do not suspend caller's execution. Expressions z also include the operation e`.get` that suspends method's execution until the value of e is computed. The type of e is a future type that is associated with a method invocation. The difference between `await` x`?` and e`.get` is that the former releases cog's lock when the value of x is still unavailable; the latter does not release cog's lock (thus being the potential cause of a deadlock).

A *pure* expression e is either a value, or a variable x, or the reserved identifier `this`. Values include the `null` object, and primitive type values, such as `true` and `1`.

In the whole paper, we assume that sequences of field declarations $\overline{T\ x}$, method declarations $\overline{M}$, and parameter declarations $\overline{T\ x}$ do not contain duplicate names. It is also assumed that every class and interface name in a program has a unique definition.

2.2 *Semantics*

`core ABS` semantics is defined as a transition relation between *configurations*, noted *cn* and defined in Figure 2. Configurations are sets of elements – therefore we identify configurations that are equal up-to associativity and commutativity – and are denoted by the juxtaposition of the elements *cn cn*; the empty configuration is denoted by ε. The transition relation uses three infinite sets of names: *object names*, ranged over by o, o', $\cdots$, *cog names*, ranged over by c, c', $\cdots$, and *future names*, ranged over by f, f', $\cdots$. Object names are partitioned according to the class and the cog they belongs. We assume there are infinitely many object names per class and the function fresh(`C`) returns a new object name of class `C`. Given an object name o, the function class(o) returns its class. The function fresh( ) returns either a fresh cog name or a fresh future name; the context will disambiguate between the twos.

Runtime values are either values v in Figure 1 or object and future names or an undefined value, which is denoted by $\perp$.

Runtime statements extend normal statements with `cont`(f) that is used to model explicit continuations in synchronous invocations. With an abuse of notation, we range over runtime values with v, v', $\cdots$ and over runtime statements with s, s', $\cdots$. We finally use a and l, possibly indexed, to range over maps from fields to runtime values and local variables to runtime values,

$$\begin{array}{rcl} cn & ::= & \epsilon \mid \mathit{fut}(f, \mathit{val}) \mid \mathit{ob}(o, a, p, q) \mid \mathit{invoc}(o, f, \mathtt{m}, \overline{v}) \mid \mathit{cog}(c, \mathit{act}) \mid cn\ cn \\ p & ::= & \{l \mid s\} \mid \mathtt{idle} \\ q & ::= & \epsilon \mid \{l \mid s\} \mid q\ q \\ s & ::= & \mathtt{cont}(f) \mid \dots \end{array} \qquad \begin{array}{rcl} \mathit{act} & ::= & o \mid \varepsilon \\ \mathit{val} & ::= & v \mid \bot \\ a & ::= & [\cdots, x \mapsto v, \cdots] \\ v & ::= & o \mid f \mid \dots \end{array}$$

Fig. 2 Runtime syntax of `core ABS`.

respectively. The map l also binds the special name destiny to a future value.

The elements of configurations are

- *objects* $ob(o, a, p, q)$ where o is an object name; a returns the values of object's fields, p is either idle, representing inactivity, or is the *active process* $\{l \mid s\}$, where l returns the values of local identifiers and s is the statement to evaluate; q is a set of processes to evaluate.
- *future binders* $fut(f, v)$ where v, called *the reply value* may be also $\bot$ meaning that the value has still not computed.
- *cog binders* $cog(c, o)$ where o is the active object; it may be ε meaning that the cog c has no active object.
- *method invocations* $invoc(o, f, \mathtt{m}, \overline{v})$.

The following auxiliary functions are used in the semantic rules (we assume a fixed `core ABS` program):

- $\mathrm{dom}(l)$ and $\mathrm{dom}(a)$ return the domain of l and a, respectively.
- $l[x \mapsto v]$ is the function such that $(l[x \mapsto v])(x) = v$ and $(l[x \mapsto v])(y) = l(y)$, when $y \neq x$. Similarly for $a[x \mapsto v]$.
- $[\![e]\!]_{(a+l)}$ returns the value of e by computing the arithmetic and boolean expressions and retrieving the value of the identifiers that is stored either in a or in l. Since a and l are assumed to have disjoint domains, we denote the union map with $a + l$. $[\![\overline{e}]\!]_{(a+l)}$ returns the tuple of values of $\overline{e}$. When e is a future name, the function $[\![\cdot]\!]_{(a+l)}$ is the identity. Namely $[\![f]\!]_{(a+l)} = f$.
- $\mathtt{C.m}$ returns the term $(\overline{T\ x})\{\overline{T'\ z}; s\}$ that contains the arguments and the body of the method $\mathtt{m}$ in the class $\mathtt{C}$.
- $\mathrm{bind}(o, f, \mathtt{m}, \overline{v}, \mathtt{C}) = \{[\mathrm{destiny} \mapsto f, \overline{x} \mapsto \overline{v}, \overline{z} \mapsto \bot] \mid s[^{o}/_{\mathtt{this}}]\}$, where $\mathtt{C.m} = (\overline{T\ x})\{\overline{T'\ z}; s\}$.
- $\mathrm{init}(\mathtt{C}, o)$ returns the process

  $\{\varnothing[\mathrm{destiny} \mapsto f_\bot] \mid s[^{o}/_{\mathtt{this}}]\}$

  where $\{\overline{T\ x}; s\}$ is the main function of the class $\mathtt{C}$. The special name destiny is initialised to a fresh (future) name $f_\bot$.
- $\mathrm{atts}(\mathtt{C}, \overline{v}, c)$ returns the map $[\mathit{cog} \mapsto c, \overline{x} \mapsto \overline{v}, \overline{x'} \mapsto \bot]$, where the class $\mathtt{C}$ is defined as

  $\mathtt{class}\ \mathtt{C}(\overline{T\ x})\{\ \overline{T'\ x'}\ ;\ \ \overline{M}\ \}$

  and where *cog* is a special field storing the cog name of the object.

The transition relation rules are collected in Figures 3 and 4. They define transitions of objects $ob(o, a, p, q)$ according to the shape of the statement in p. We focus on rules concerning the concurrent part of `core ABS`, since the other ones are standard. Rules (Await-True) and (Await-False) model the `await` e`?` operation: if the (future) value of e has been computed then `await` terminates; otherwise the active process becomes idle. In this case, if the object owns the control of the cog then it may release such control – rule (Release-Cog). Otherwise, when the cog has no active process, the object gets the control of the cog and activates one of its processes – rule (Activate). Rule (Read-Fut) permits the retrieval of the value returned by a method; the object does not release the control of the cog until this value has been computed.

The two types of object creation are modeled by (New-Object) and (New-Cog-Object). The first one creates the new object in the same cog. The new object is idle because the cog has already an active object. The second one creates the object in a new cog and makes it active by scheduling the process corresponding to the main function of the class. The special field *cog* is initialized accordingly; the other object's fields are initialized by evaluating the arguments of the operation – see definition of atts.

Rule (Async-Call) defines asynchronous method invocation $x = e\mathtt{!m}(\overline{e})$. This rule creates a fresh future name that is assigned to the identifier x. The evaluation of the called method is transferred to a different process – see rule (Bind-Mtd). Therefore the caller can progress without waiting for callee's termination. Rule (Cog-Sync-Call) defines synchronous method invocation on an object in the *same* cog (because of the premise $a'(\mathrm{cog}) = c$ and the element $cog(c, o)$ in the configuration). The control is passed to the called object that executes the body of the called method followed by a special statement $\mathtt{cont}(f')$, where f' is a fresh future name. When the evaluation of the body terminates, the caller process is scheduled again using the name f' – see rule (Cog-Sync-Return-Sched). Rules (Self-Sync-Call) and (Rem-Sync-Call) deal with synchronous method invocations of the same object and of objects in different cogs, respectively. The former is similar to (Cog-Sync-

(Skip)
$$\begin{array}{c} ob(o,a,\{l \mid \texttt{skip}; s\},q) \\ \rightarrow ob(o,a,\{l \mid s\},q) \end{array}$$

(Assign-Local)
$$\frac{x \in \mathrm{dom}(l) \quad v = [\![e]\!]_{(a+l)}}{\begin{array}{c} ob(o,a,\{l \mid x = e; s\},q) \\ \rightarrow ob(o,a,\{l[x \mapsto v] \mid s\},q) \end{array}}$$

(Assign-Field)
$$\frac{x \in \mathrm{dom}(a) \setminus \mathrm{dom}(l) \quad v = [\![e]\!]_{(a+l)}}{\begin{array}{c} ob(o,a,\{l \mid x = e; s\},q) \\ \rightarrow ob(o,a[x \mapsto v],\{l \mid s\},q) \end{array}}$$

(Cond-True)
$$\frac{\texttt{true} = [\![e]\!]_{(a+l)}}{\begin{array}{c} ob(o,a,\{l \mid \texttt{if}\ e\ \texttt{then}\ \{s_1\}\ \texttt{else}\ \{s_2\}; s\},q) \\ \rightarrow ob(o,a,\{l \mid s_1; s\},q) \end{array}}$$

(Cond-False)
$$\frac{\texttt{false} = [\![e]\!]_{(a+l)}}{\begin{array}{c} ob(o,a,\{l \mid \texttt{if}\ e\ \texttt{then}\ \{s_1\}\ \texttt{else}\ \{s_2\}; s\},q) \\ \rightarrow ob(o,a,\{l \mid s_2; s\},q) \end{array}}$$

(Await-True)
$$\frac{f = [\![e]\!]_{(a+l)} \quad v \neq \bot}{\begin{array}{c} ob(o,a,\{l \mid \texttt{await}\, e\,?; s\},q)\ fut(f,v) \\ \rightarrow ob(o,a,\{l \mid s\},q)\ fut(f,v) \end{array}}$$

(Await-False)
$$\frac{f = [\![e]\!]_{(a+l)}}{\begin{array}{c} ob(o,a,\{l \mid \texttt{await}\, e\,?; s\},q)\ fut(f,\bot) \\ \rightarrow ob(o,a,\mathrm{idle},q \cup \{l \mid \texttt{await}\, e\,?; s\})\ fut(f,\bot) \end{array}}$$

(Release-Cog)
$$\begin{array}{c} ob(o,a,\mathrm{idle},q)\ cog(c,o) \\ \rightarrow ob(o,a,\mathrm{idle},q)\ cog(c,\epsilon) \end{array}$$

(Activate)
$$\frac{c = a(\mathrm{cog})}{\begin{array}{c} ob(o,a,\mathrm{idle},q \cup \{l \mid s\})\ cog(c,\epsilon) \\ \rightarrow ob(o,a,\{l \mid s\},q)\ cog(c,o) \end{array}}$$

(Read-Fut)
$$\frac{f = [\![e]\!]_{(a+l)} \quad v \neq \bot}{\begin{array}{c} ob(o,a,\{l \mid x = e.\texttt{get}; s\},q)\ fut(f,v) \\ \rightarrow ob(o,a,\{l \mid x = v; s\},q)\ fut(f,v) \end{array}}$$

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

(New-Cog-Object)
$$\frac{\begin{array}{c} c' = \mathrm{fresh}(\,) \quad o' = \mathrm{fresh}(\texttt{C}) \quad p = \mathrm{init}(\texttt{C},o') \\ a' = \mathrm{atts}(\texttt{C},[\![\overline{e}]\!]_{(a+l)},c') \end{array}}{\begin{array}{c} ob(o,a,\{l \mid x = \texttt{new}\ \texttt{cog}\ \texttt{C}(\overline{e}); s\},q) \\ \rightarrow ob(o,a,\{l \mid x = o'; s\},q) \\ ob(o',a',p,\varnothing) \quad cog(c',o') \end{array}}$$

Fig. 3 Semantics of `core ABS`(1).

(Async-Call)
$$\frac{o' = [\![e]\!]_{(a+l)} \quad \overline{v} = [\![\overline{e}]\!]_{(a+l)} \quad f = \mathrm{fresh}(\,)}{\begin{array}{c} ob(o,a,\{l \mid x = e!\texttt{m}(\overline{e}); s\},q) \\ \rightarrow ob(o,a,\{l \mid x = f; s\},q)\ invoc(o',f,\texttt{m},\overline{v})\ fut(f,\bot) \end{array}}$$

(Bind-Mtd)
$$\frac{\{l \mid s\} = \mathrm{bind}(o,f,\texttt{m},\overline{v},\mathrm{class}(o))}{\begin{array}{c} ob(o,a,p,q)\ invoc(o,f,\texttt{m},\overline{v}) \\ \rightarrow ob(o,a,p,q \cup \{l \mid s\}) \end{array}}$$

(Cog-Sync-Call)
$$\frac{\begin{array}{c} o' = [\![e]\!]_{(a+l)} \quad \overline{v} = [\![\overline{e}]\!]_{(a+l)} \quad f = \mathrm{fresh}(\,) \\ c = a'(\mathrm{cog}) \quad f' = l(\mathrm{destiny}) \\ \{l' \mid s'\} = \mathrm{bind}(o',f,\texttt{m},\overline{v},class(o')) \end{array}}{\begin{array}{c} ob(o,a,\{l \mid x = e.\texttt{m}(\overline{e}); s\},q) \\ ob(o',a',\mathrm{idle},q')\ cog(c,o) \\ \rightarrow ob(o,a,\mathrm{idle},q \cup \{l \mid \texttt{await}\ f?; x = f.\texttt{get}; s\})\ fut(f,\bot) \\ ob(o',a',\{l' \mid s'; \texttt{cont}\ f'\},q')\ cog(c,o') \end{array}}$$

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

(Self-Sync-Call)
$$\frac{\begin{array}{c} f' = l(\mathrm{destiny}) \quad o = [\![e]\!]_{(a+l)} \quad \overline{v} = [\![\overline{e}]\!]_{(a+l)} \\ f = \mathrm{fresh}(\,) \quad \{l' \mid s'\} = \mathrm{bind}(o,f,\texttt{m},\overline{v},class(o)) \end{array}}{\begin{array}{c} ob(o,a,\{l \mid x = e.\texttt{m}(\overline{e}); s\},q) \\ \rightarrow ob(o,a,\{l' \mid s'; \texttt{cont}(f')\},q \cup \{l \mid \texttt{await}\ f?; x = f.\texttt{get}; s\})\ fut(f,\bot) \end{array}}$$

(Return)
$$\frac{v = [\![e]\!]_{(a+l)} \quad f = l(\mathrm{destiny})}{\begin{array}{c} ob(o,a,\{l \mid \texttt{return}\ e; s\},q)\ fut(f,\bot) \\ \rightarrow ob(o,a,\{l \mid s\},q)\ fut(f,v) \end{array}}$$

(Rem-Sync-Call)
$$\frac{o' = [\![e]\!]_{(a+l)} \quad f = \mathrm{fresh}(\,) \quad a(\mathrm{cog}) \neq a'(\mathrm{cog})}{\begin{array}{c} ob(o,a,\{l \mid x = e.\texttt{m}(\overline{e}); s\},q)\ ob(o',a',p,q') \\ \rightarrow ob(o,a,\{l \mid f = e!\texttt{m}(\overline{e}); x = f.\texttt{get}; s\},q) \\ ob(o',a',p,q') \end{array}}$$

(Self-Sync-Return-Sched)
$$\frac{f = l'(\mathrm{destiny})}{\begin{array}{c} ob(o,a,\{l \mid \texttt{cont}(f)\},q \cup \{l' \mid s\}) \\ \rightarrow ob(o,a,\{l' \mid s\},q) \end{array}}$$

(Context)
$$\frac{cn \rightarrow cn'}{cn\ cn'' \rightarrow cn'\ cn''}$$

Fig. 4 Semantics of `core ABS`(2).

Call) except that there is no control on cogs. The latter one implements the synchronous invocation through an asynchronous one followed by an explicit synchronisation operation.

It is worth to observe that the rules (Activate), (Cog-Sync-Call) and (Self-Sync-Call) are different from the corresponding ones in [23]. In fact, in [23] rule (Activate) uses an unspecified *select* predicate that activates one task from the queue of processes to evaluate. According to the rules (Cog-Sync-Call) and (Self-Sync-Call) in that paper, the activated process might be a caller of a synchronous invocation, which has a `get` operation. To avoid potential deadlock of a wrong *select* implementation, we have prefixed the `get`s in (Cog-Sync-Call) and (Self-Sync-Call) with `await` operations.

The initial configuration of a `core ABS` program with main function $\{\overline{T} : \overline{x}\,;\, s\}$ is

$$ob(start, \varepsilon, \{[\mathrm{destiny} \mapsto f_{start}, \overline{x} \mapsto \bot] \mid s\}, \varnothing) \\ cog(\mathrm{start}, start)$$

where start and *start* are special cog and object names, respectively, and f_{start} is a fresh future name. As usual, let $\longrightarrow^*$ be the reflexive and transitive closure of $\longrightarrow$.

A configuration *cn* is *sound* if

(i) different elements $cog(c, o)$ and $cog(c', o')$ in *cn* are such that $c \neq c'$ and $o \neq \varepsilon$ implies $o \neq o'$,

(ii) if $ob(o, a, p, q)$ and $ob(o', a', p', q')$ are different objects in *cn* such that $a(\text{cog}) = a'(\text{cog})$ then either $p = \text{idle}$ or $p' = \text{idle}$.

We notice that the initial configurations of `core ABS` programs are sound. The following statement guarantees that the property "there is at most one active object per cog" is an invariance of the transition relation.

Proposition 1 *If cn is* sound *and* $cn \longrightarrow cn'$ *then* cn' *is sound as well.*

As an example of `core ABS` semantics, in Figure 7 we have detailed the transitions of the program in Example 2. The non-interested reader may safely skip it.

2.3 *Samples of concurrent programs in* `core ABS`

The `core ABS` code of two concurrent programs are discussed. These codes will be analysed in the following sections.

Example 1 Figure 5 collects three different implementations of the factorial function in a class `Math`. The function `fact_g` is the standard definition of factorial: the recursive invocation `this!fact_g(n-1)` is followed by a `get` operation that retrieves the value returned by the invocation. Yet, `get` does not allow the task to release the cog lock; therefore the task evaluating `this!fact_g(n-1)` is fated to be delayed forever because its object (and, therefore, the corresponding cog) is the same as that of the caller. The function `fact_ag` solves this problem by permitting the caller to release the lock with an explicit `await` operation, before getting the actual value with `x.get`. An alternative solution is defined by the function `fact_nc`, whose code is similar to that of `fact_g`, except for that `fact_nc` invokes `z!fact_nc(n-1)` recursively, where `z` is an object in a new cog. This means the task of `z!fact_nc(n-1)` may start without waiting for the termination of the caller.

Programs that are particularly hard to verify are those that may manifest misbehaviours according to the schedulers choices. The following example discusses one case.

Example 2 The class `CpxSched` of Figure 6 defines three methods. Method `m1` asynchronously invokes `m2` on its own argument `y`, passing to it the field `x` as argument .

```
class Math {
   Int fact_g(Int n){
      Fut<Int> x ;
      Int m ;
      if (n==0) { return 1; }
      else { x = this!fact_g(n-1); m = x.get;
              return n*m; }
   }
   Int fact_ag(Int n){
      Fut<Int> x ;
      Int m ;
      if (n==0) { return 1; }
      else { x = this!fact_ag(n-1);
              await x?; m = x.get;
              return n*m; }
   }
   Int fact_nc(Int n){
      Fut<Int> x ;
      Int m ;
      Math z ;
      if (n==0) { return 1 ; }
      else { z = new cog Math();
              x = z!fact_nc(n-1); m = x.get;
              return n*m; }
   }
}
```

Fig. 5 The class `Math`

```
interface I { Fut<Unit> m1(I y); Unit m2(I z);
                Unit m3() ; }

class CpxSched (I u) implements I {
    Fut<Unit> m1(I y) {
        Fut<Unit> h;
        Fut<Unit> g ;
        h = y!m2(u);
        g = u!m2(y);
        return g;
    }
    Unit m2(I z) {
        Fut<Unit> h ;
        h = z!m3();
        h.get;
     }
    Unit m3(){
    }
}
```

Fig. 6 The class `CpxSched`

Then it asynchronously invokes `m2` on the field `x`, passing its same argument `y`. Method `m2` invokes `m3` on the argument `z` and blocks waiting for the result. Method `m3` simply returns.

Next, consider the following main function:

```
  { I x; I y; I z;
    Fut<Fut<Unit>> w ;
    x = new CpxSched(null);
    y = new CpxSched(x);
    z = new cog CpxSched(null);
    w = y!m1(z); }
```

The initial configuration is

$ob(start, \varepsilon, \{l \mid s\}, \varnothing)\, cog(\text{start}, start)$

where $l = [\text{destiny} \mapsto f_{start}, x \mapsto \bot, y \mapsto \bot, z \mapsto \bot, w \mapsto \bot]$ and s is the statement of the main function. The sequence of transitions of this configuration is illustrated in Figure 7, where

s', s'',	s''' are the obvious sub-statements of the main function
$l_o =$	$[\text{destiny} \mapsto f'', z \mapsto o'', u \mapsto \bot, h \mapsto \bot]$
$l_{o'} =$	$[\text{destiny} \mapsto f, y \mapsto o'', g \mapsto \bot, h \mapsto \bot]$
$l_{o''} =$	$[\text{destiny} \mapsto f', z \mapsto o, u \mapsto \bot, h \mapsto \bot]$
$l'_o =$	$[\text{destiny} \mapsto f'''']$
$l'_{o''} =$	$[\text{destiny} \mapsto f''']$
$a_{\texttt{null}} =$	$[cog \mapsto \text{start}, \texttt{x} \mapsto \texttt{null}]$
$a_o =$	$[cog \mapsto \text{start}, \texttt{x} \mapsto o]$
$s_{o'} =$	`h= y!m2(this.x); g= this.x!m2(y); return g;`
$s_o =$	$s_{o''} =$ `h= z!m3(); h.get;`

We notice that the last configuration of Figure 7 is stuck, *i.e.* it cannot progress anymore. In fact, it is a deadlock according the forthcoming Definition 1.

2.4 *Deadlocks*

The definition below identifies deadlocked configurations by detecting chains of dependencies between tasks that cannot progress. To ease the reading, we write

- $p[f.\texttt{get}]^a$ whenever $p = \{l|s\}$ and s is $x = y.\texttt{get}; s'$ and $[\![y]\!]_{(a+l)} = f$;
- $p[\texttt{await}\ f]^a$ whenever $p = \{l|s\}$ and s is $\texttt{await}\ e?; s'$ and $[\![e]\!]_{(a+l)} = f$;
- $p.f$ whenever $p = \{l|s\}$ and $l(\text{destiny}) = f$.

Definition 1 A configuration *cn* is *deadlocked* if there are

$ob(o_0, a_0, p_0, q_0), \cdots, ob(o_{n-1}, a_{n-1}, p_{n-1}, q_{n-1}) \in cn$
and
$p'_i \in p_i \cup q_i, \qquad \text{with } 0 \le i \le n-1$

such that (let $+$ be computed modulo n in the following)

1. $p'_0 = p_0[f_0.\texttt{get}]^{a_0}$ and if $p'_i[f_i.\texttt{get}]^{a_i}$ then $p'_i = p_i$;
2. if $p'_i[f_i.\texttt{get}]^{a_i}$ or $p'_i[\texttt{await}\ f_i]^{a_i}$ then $fut(f_i, \bot) \in cn$ and
   - either $p'_{i+1}[f_{i+1}.\texttt{get}]^{a_{i+1}}$ and $p'_{i+1} = \{l_{i+1}|s_{i+1}\}$ and $f_i = l_{i+1}(\text{destiny})$;
   - or $p'_{i+1}[\texttt{await}\ f_{i+1}]^{a_{i+1}}$ and $p'_{i+1} = \{l_{i+1}|s_{i+1}\}$ and $f_i = l_{i+1}(\text{destiny})$;
   - or $p'_{i+1} = p_{i+1} = \text{idle}$ and $a_{i+1}(cog) = a_{i+2}(cog)$ and $p'_{i+2}[f_{i+2}.\texttt{get}]^{a_{i+2}}$ (in this case p_{i+1} is idle, by soundness).

A configuration *cn* is *deadlock-free* if, for every $cn \longrightarrow^* cn'$, cn' is not deadlocked. A `core ABS` program is *deadlock-free* if its initial configuration is *deadlock-free*.

According to Definition 1, a configuration is deadlocked when there is a circular dependency between processes. The processes involved in such circularities are performing a `get` or `await` synchronisation or they are idle and will never grab the lock because another active process in the same cog will not return. We notice that, by Definition 1, at least one active process is blocked on a `get` synchronisation. We also notice that the objects in Definition 1 may be not pairwise different (see example 1 below). The following examples should make the definition clearer; the reader is recommended to instantiate the definition every time.

1. (self deadlock)

   $$\begin{array}{l} ob(o_1, a_1, \{l_1|x_1 = e_1.\texttt{get}; s_1\}, q_1) \\ ob(o_2, a_2, \text{idle}, q_2 \cup \{l_2|s_2\}) \\ fut(f_2, \bot), \end{array}$$

   where $[\![e_1]\!]_{(a_1+l_1)} = l_2(destiny) = f_2$ and $a_1(cog) = a_2(cog)$. In this case, the object o_1 keeps the control of its own cog while waiting for the result of a process in o_2. This process cannot be scheduled because the corresponding cog is not released. A similar situation can be obtained with one object:

   $$\begin{array}{l} ob(o_1, a_1, \{l_1|x_1 = e_1.\texttt{get}; s_1\}, q_1 \cup \{l_2|s_2\}) \\ fut(f_2, \bot), \end{array}$$

   where $[\![e_1]\!]_{(a_1+l_1)} = l_2(destiny) = f_2$. In this case, the objects of the Definition 1 are

   $$ob(o_1, a_1, p_1, q_1) \quad ob(o_1, a_1, p_2, q_2 \cup \{l_2|s_2\})$$

   where $p'_1 = p_1 = \{l_1|x_1 = e_1.\texttt{get}; s_1\}$, $p'_2 = \{l_2|s_2\}$ and $q_1 = q_2 \cup \{l_2|s_2\}$.
2. (`get`-`await` deadlock)

   $$\begin{array}{l} ob(o_1, a_1, \{l_1|x_1 = e_1.\texttt{get}; s_1\}, q_1) \\ ob(o_2, a_2, \{l_2|\texttt{await}\ e_2?; s_2\}, q_2) \\ ob(o_3, a_3, \text{idle}, q_3 \cup \{l_3|s_3\}) \end{array}$$

   where $l_3(destiny) = [\![e_2]\!]_{a_2+l_2}$, $l_2(destiny) = [\![e_1]\!]_{a_1+l_1}$, $a_1(cog) = a_3(cog)$ and $a_1(cog) \neq a_2(cog)$. In this case, the objects o_1 and o_2 have different cogs. However o_2 cannot progress because it is waiting for a result of a process that cannot be scheduled (because it has the same cog of o_1).

$$
\begin{array}{l}
\mathit{ob}(\mathit{start}, \varepsilon, \{l \mid s\}, \varnothing)\ \mathit{cog}(\text{start}, \mathit{start}) \\
\quad \longrightarrow^2 \quad (\textsc{New-Object}) \text{ and } (\textsc{Assign-Local}) \\
\mathit{ob}(\mathit{start}, \varepsilon, \{l[\texttt{x} \mapsto o] \mid \texttt{y = new C(x);} s''\}, \varnothing)\ \mathit{cog}(\text{start}, \mathit{start})\ \mathit{ob}(o, a_{\texttt{null}}, \text{idle}, \varnothing) \\
\quad \longrightarrow^2 \quad (\textsc{New-Object}) \text{ and } (\textsc{Assign-Local}) \\
\mathit{ob}(\mathit{start}, \varepsilon, \{l[\texttt{x} \mapsto o, \texttt{y} \mapsto o'] \mid \texttt{z = new cog C(null);} s'''\}, \varnothing)\ \mathit{cog}(\text{start}, \mathit{start})\ \mathit{ob}(o, a_{\texttt{null}}, \text{idle}, \varnothing)\ \mathit{ob}(o', a_o, \text{idle}, \varnothing) \\
\quad \longrightarrow^2 \quad (\textsc{New-Cog-Object}) \text{ and } (\textsc{Assign-Local}) \\
\mathit{ob}(\mathit{start}, \varepsilon, \{l[\texttt{x} \mapsto o, \texttt{y} \mapsto o', \texttt{z} \mapsto o''] \mid \texttt{w = y!m1(z);}\}, \varnothing)\ \mathit{cog}(\text{start}, \mathit{start})\ \mathit{ob}(o, a_{\texttt{null}}, \text{idle}, \varnothing)\ \mathit{ob}(o', a_o, \text{idle}, \varnothing) \\
\qquad \mathit{ob}(o'', [\mathit{cog} \mapsto c, \texttt{x} \mapsto \texttt{null}], \text{idle}, \varnothing)\ \mathit{cog}(c, o'') \\
\quad \longrightarrow \quad (\textsc{Async-Call}) \\
\mathit{ob}(\mathit{start}, \varepsilon, \{l[\texttt{x} \mapsto o, \texttt{y} \mapsto o', \texttt{z} \mapsto o''] \mid \texttt{w = } \text{f;}\}, \varnothing)\ \mathit{cog}(\text{start}, \mathit{start})\ \mathit{ob}(o, a_{\texttt{null}}, \text{idle}, \varnothing)\ \mathit{ob}(o', a_o, \text{idle}, \varnothing) \\
\qquad \mathit{ob}(o'', [\mathit{cog} \mapsto c, \texttt{x} \mapsto \texttt{null}], \text{idle}, \varnothing)\ \mathit{cog}(c, o'')\ \mathit{invoc}(o', f, \texttt{m1}, o'')\ \mathit{fut}(f, \bot) \\
\quad \longrightarrow \quad (\textsc{Bind-Mtd}) \\
\mathit{ob}(\mathit{start}, \varepsilon, \{l[\texttt{x} \mapsto o, \texttt{y} \mapsto o', \texttt{z} \mapsto o''] \mid \texttt{w = } \text{f;}\}, \varnothing)\ \mathit{cog}(\text{start}, \mathit{start})\ \mathit{ob}(o, a_{\texttt{null}}, \text{idle}, \varnothing)\ \mathit{ob}(o', a_o, \text{idle}, \{l_{o'} \mid s_{o'}\}) \\
\mathit{ob}(o'', [\mathit{cog} \mapsto c, \texttt{x} \mapsto \texttt{null}], \text{idle}, \varnothing) \qquad \mathit{cog}(c, o'')\ \mathit{fut}(f, \bot) \\
\quad \longrightarrow^+ (\textsc{Activate}) \text{ and twice } (\textsc{Async-Call}) \text{ and } (\textsc{Return}) \\
\mathit{ob}(\mathit{start}, \varepsilon, \{l[\texttt{x} \mapsto o, \texttt{y} \mapsto o', \texttt{z} \mapsto o''] \mid \texttt{w = } \text{f;}\}, \varnothing)\ \mathit{cog}(\text{start}, \mathit{start})\ \mathit{ob}(o, a_{\texttt{null}}, \text{idle}, \varnothing)\ \mathit{ob}(o', a_o, \text{idle}, \varnothing) \\
\qquad \mathit{ob}(o'', [\mathit{cog} \mapsto c, \texttt{x} \mapsto \texttt{null}], \text{idle}, \varnothing)\ \mathit{cog}(c, o'')\ \mathit{fut}(f, f'')\ \mathit{fut}(f', \bot)\ \mathit{fut}(f'', \bot) \qquad (\star) \\
\qquad \mathit{invoc}(o'', f', \texttt{m2}, o)\ \mathit{invoc}(o, f'', \texttt{m2}, o'') \\
\quad \longrightarrow^2 \quad \text{twice } (\textsc{Bind-Mtd}) \\
\mathit{ob}(\mathit{start}, \varepsilon, \{l[\texttt{x} \mapsto o, \texttt{y} \mapsto o', \texttt{z} \mapsto o''] \mid \texttt{w = } \text{f;}\}, \varnothing)\ \mathit{cog}(\text{start}, \mathit{start})\ \mathit{ob}(o, a_{\texttt{null}}, \text{idle}, \{l_o \mid s_o\})\ \mathit{ob}(o', a_o, \text{idle}, \varnothing) \\
\qquad \mathit{ob}(o'', [\mathit{cog} \mapsto c, \texttt{x} \mapsto \texttt{null}], \text{idle}, \{l_{o''} \mid s_{o''}\})\ \mathit{cog}(c, o'')\ \mathit{fut}(f, f'')\ \mathit{fut}(f', \bot)\ \mathit{fut}(f'', \bot) \\
\quad \longrightarrow^+ \quad \text{twice } (\textsc{Activate}) \text{ and twice } (\textsc{Async-Call}) \\
\mathit{ob}(\mathit{start}, \varepsilon, \{l[\texttt{x} \mapsto o, \texttt{y} \mapsto o', \texttt{z} \mapsto o'', \texttt{w} \mapsto f] \mid \text{idle}\}, \varnothing)\ \mathit{cog}(\text{start}, o)\ \mathit{ob}(o, a_{\texttt{null}}, \{l_o[\texttt{h} \mapsto f'''] \mid \texttt{h.get;} s'_o\}, \varnothing) \\
\qquad \mathit{ob}(o', a_o, \text{idle}, \varnothing)\ \mathit{ob}(o'', [\mathit{cog} \mapsto c, \texttt{x} \mapsto \texttt{null}], \{l_{o''}[\texttt{h} \mapsto f''''] \mid \texttt{h.get;} s'_{o''}\}, \varnothing)\ \mathit{cog}(c, o'')\ \mathit{fut}(f, f'')\ \mathit{fut}(f', \bot) \\
\qquad \mathit{fut}(f'', \bot)\ \mathit{invoc}(o'', f''', \texttt{m3}, \varepsilon)\ \mathit{fut}(f', \bot)\ \mathit{fut}(f''', \bot)\ \mathit{invoc}(o, f'''', \texttt{m3}, \varepsilon)\ \mathit{fut}(f'', \bot)\ \mathit{fut}(f'''', \bot) \\
\quad \longrightarrow^2 \quad \text{twice } (\textsc{Bind-Mtd}) \\
\mathit{ob}(\mathit{start}, \varepsilon, \{l[\texttt{x} \mapsto o, \texttt{y} \mapsto o', \texttt{z} \mapsto o'', \texttt{w} \mapsto f] \mid \text{idle}\}, \varnothing)\ \mathit{cog}(\text{start}, o)\ \mathit{ob}(o, a_{\texttt{null}}, \{l_o[\texttt{h} \mapsto f'''] \mid \texttt{h.get;} s'_o\}, \{l'_o \mid \texttt{skip;}\}) \\
\qquad \mathit{ob}(o', a_o, \text{idle}, \varnothing)\ \mathit{ob}(o'', [\mathit{cog} \mapsto c, \texttt{x} \mapsto \texttt{null}], \{l_{o''}[\texttt{h} \mapsto f''''] \mid \texttt{h.get;} s'_{o''}\}, \{l'_{o''} \mid \texttt{skip;}\})\ \mathit{cog}(c, o'') \\
\qquad \mathit{fut}(f, f'')\ \mathit{fut}(f', \bot)\ \mathit{fut}(f'', \bot)\ \mathit{fut}(f', \bot)\ \mathit{fut}(f''', \bot)\ \mathit{fut}(f'', \bot)\ \mathit{fut}(f'''', \bot)
\end{array}
$$

Fig. 7 Reduction of Example 2

3. (get-idle deadlock)

$$
\begin{array}{l}
\mathit{ob}(o_1, a_1, \{l_1 | x_1 = e_1.\texttt{get}; s_1\}, q_1) \\
\mathit{ob}(o_2, a_2, \text{idle}, q_1 \cup \{l_2 | s_2\}) \\
\mathit{ob}(o_3, a_3, \{l_3 | x_3 = e_3.\texttt{get}; s_3\}, q_3) \\
\mathit{ob}(o_4, a_4, \text{idle}, q_4 \cup \{l_4 | s_4\}) \\
\mathit{ob}(o_5, a_5, \{l_5 | x_5 = e_5.\texttt{get}; s_5\}, q_5) \\
\mathit{fut}(f_1, \bot),\ \mathit{fut}(f_2, \bot),\ \mathit{fut}(f_4, \bot)
\end{array}
$$

where $f_2 = l_2(\mathit{destiny}) = [\![e_1]\!]_{a_1+l_1}$, $f_4 = l_4(\mathit{destiny}) = [\![e_3]\!]_{a_3+l_3}$, $f_1 = l_1(\mathit{destiny}) = [\![e_5]\!]_{a_5+l_5}$ and $a_2(\mathit{cog}) = a_3(\mathit{cog})$ and $a_4(\mathit{cog}) = a_5(\mathit{cog})$.

A deadlocked configuration has at least one object that is stuck (the one performing the get instruction). This means that the configuration may progress, but future configurations will still have one object stuck.

Proposition 2 *If cn is deadlocked and $cn \longrightarrow cn'$ then cn' is deadlocked as well.*

Definition 1 is about runtime entities that have no static counterpart. Therefore we consider a notion weaker than deadlocked configuration. This last notion will be used in the Appendices to demonstrate the correctness of the inference system in Section 4.

Definition 2 A configuration *cn* has

(i) a *dependency* (c, c') if

$$\mathit{ob}(o, a, \{l | x = e.\texttt{get}; s\}, q), \mathit{ob}(o', a', p', q') \in \mathit{cn}$$

with $[\![e]\!]_{(a+l)} = f$ and $a(\mathit{cog}) = c$ and $a'(\mathit{cog}) = c'$ and

(a) either $\mathit{fut}(f, \bot) \in \mathit{cn}$, $l'(\text{destiny}) = f$ and $\{l'|s'\} \in p' \cup q'$;

(b) or $\mathit{invoc}(o', f, \texttt{m}, \overline{v}) \in \mathit{cn}$.

(ii) a *dependency* $(c, c')^{\texttt{w}}$ if

$$\mathit{ob}(o, a, p, q), \mathit{ob}(o', a', p', q') \in \mathit{cn}$$

and $\{l|\texttt{await}\ e?; s\} \in p \cup q$ and $[\![e]\!]_{(a+l)} = f$ and

(a) either $\mathit{fut}(f, \bot) \in \mathit{cn}$, $l'(\text{destiny}) = f$ and $\{l'|s'\} \in p' \cup q'$;

(b) or $\mathit{invoc}(o', f, \texttt{m}, \overline{v}) \in \mathit{cn}$.

Given a set A of dependencies, let the get-*closure* of A, noted $A^{\texttt{get}}$, be the least set such that

1. $A \subseteq A^{\texttt{get}}$;
2. if $(c, c') \in A^{\texttt{get}}$ and $(c', c'')^{[\texttt{w}]} \in A^{\texttt{get}}$ then $(c, c'') \in A^{\texttt{get}}$, where $(c', c'')^{[\texttt{w}]}$ denotes either the pair (c', c'') or the pair $(c', c'')^{\texttt{w}}$.

A configuration contains a *circularity* if the get-closure of its set of dependencies has a pair (c, c).

Proposition 3 *If a configuration is deadlocked then it has a circularity. The converse is false.*

Proof The statement is a straightforward consequence of the definition of deadlocked configuration. To show that the converse is false, consider the configuration

$$\begin{array}{l} ob(o_1, a_1, \{l_1 | x_1 = e_1.\texttt{get}; s_1\}, q_1) \\ ob(o_2, a_2, \text{idle}, q_2 \cup \{l_2 | \texttt{await}\ e_2?; s_2\}) \\ ob(o_3, a_3, \{l_3 | \texttt{return}\ e_3\}, q_3) \quad cn \end{array}$$

where $l_3(destiny) = [\![e_1]\!]_{a_1+l_1}$, $l_1(destiny) = [\![e_2]\!]_{a_2+l_2}$, $c_2 = a_2(cog) = a_3(cog)$ and $c_1 = a_1(cog) \neq c_2$. This configuration has the dependencies

$$\{(c_1, c_2), (c_2, c_1)^{\mathtt{w}}\}$$

whose `get`-closure contains the circularity (c_1, c_1). However the configuration is not deadlocked. □

Example 3 The final configuration of Figure 7 is *deadlocked* according to Definition 1. In particular, there are two objects o and o'' running on different cogs whose active processes have a `get`-synchronisation on the result of process in the other object: o is performing a `get` on a future f''' which is $l'_{o''}$(destiny), and o'' is performing a `get` on a future f'''' which is l'_o(destiny) and $fut(f''', \bot)$ and $fut(f'''', \bot)$. We notice that, if in the configuration $(\star)$ we choose to evaluate $invoc(o'', f', \mathtt{m2}, o)$ when the evaluation of $invoc(o, f'', \mathtt{m2}, o'')$ has been completed (or conversely) then no deadlock is manifested.

3 Restrictions of `core ABS` in the current release of the contract inference system

The contract inference system we describe in the next section has been prototyped. To verify its feasibility, the current release of the prototype addresses a subset of `core ABS` features. These restrictions permit to ease the initial development of the inference system and do not jeopardise its extension to the full language. Below we discuss the restrictions and, for each of them, either we explain the reasons why they will be retained in the next release(s) or we detail the techniques that will be used to remove them. (It is also worth to notice that, notwithstanding the following restrictions, it is still possible to verify large commercial cases, such as a core component of FAS discussed in this paper.)

Returns. `core ABS` syntax admits `return` statements with continuations – see Figure 1 – that, according to the semantics, are executed *after the return value has been delivered to the caller.* These continuations can be hardly controlled by programmers and usually cause unpredictable behaviours, in particular as regards deadlocks. To increase the precision of our analysis we assume that `core ABS` programs have empty continuations of `return` statements. We observe that this constraint has an exception at run-time: in order to define the semantics of synchronous method invocation, rules (Cog-Sync-Call) and (Self-Sync-Call) append a `cont` f continuation to statements in order to let the right caller be scheduled. Clearly this is the `core ABS` definition of synchronous invocation and it does not cause any misbehaviour.

Fields assignments. Assignments in `core ABS` (as usual in object-oriented languages) may update the fields of objects that are accessed concurrently by other threads, thus could lead to indeterminate behaviour. In order to simplify the analysis, we constrain field assignments as follows. If the field is *not of future type* then we keep field's record structure unchanging. For instance, if a field contains an object of cog a, then that field may be only updated with objects belonging to a (and this correspondence must hold recursively with respect to the fields of objects referenced by a). When the field is of a primitive type (`Int`, `Bool`, etc.) this constraint is equivalent to the standard type-correctness. It is possible to be more liberal as regards fields assignments. In [20] an initial study for covering full-fledged field assignments was undertaken using so-called union types (that is, by extending the syntax of future records with a + operator, as for contracts, see below) and collecting all records in the inference rule of the field assignment (and the conditional). When the field is *of future type* then we disallow assignments. In fact, assignments to such fields allow a programmer to define unexpected behaviours. Consider for example the following class `Foo` implementing `I_Foo`:

```
class Foo(){
	Fut<T> x ;
	Unit foo_m () {
		Fut<T> local ;
		I_Foo y = new cog Foo() ;
		I_Foo z = new cog Foo() ;
		local = y!foo_n(this) ;
		x = z!foo_n(this) ;
		await local? ;
		await x?
	}
	T foo_n(I_Foo x){ . . . }
}
```

If the main function is

```
	{ I_Foo x ;
	  Fut<Unit> u ;
	  Fut<Unit> v ;
	  x = new cog Foo() ;
	  u = x!foo_m() ;
	  v = x!foo_m() ;  }
```

then the two invocations in lines 18 and 19 run in parallel. Each invocation of `foo_m` invokes `foo_n` twice that apparently terminate when `foo_m` returns (with the two final `await` statements). However this may be not the case because the invocation of `foo_n` line 8 is stored in a field: consider that the first invocation of `foo_m` (line 18) starts executing, sets the field `x` with its own future f, and then, with the `await` statement in line 9, the second invocation of `foo_m` (line 19) starts. That second invocation *replaces* the content of the field `x` with its own future f': at that point, the second invocation (line 19) will synchronise with f' before terminating, then the first invocation (line 18) will resume and also synchronised with f' before terminating. Hence, even after both invocations (line 18 and 19) are finished, the invocation of `foo_n` in line 8 may still be running. It is not too difficult to trace such residual behaviours in the inference system (for instance, by grabbing them using a function like $unsync(\Gamma)$). However, this extension will entangle the inference system and for this reason we decided to deal with generic field assignments in a next release.

It is worth to recall that these restrictions does not apply to local variables of methods, as they can only be accessed by the method in which they are declared. Actually, the foregoing inference algorithm tracks changes of local variables.

Interfaces. In `core ABS` objects are typed with interfaces, which may have several implementations. As a consequence, when a method is invoked, it is in general not possible to statically determine which method will be executed at runtime (dynamic dispatch). This is problematic for our technique because it breaks the association of a unique abstract behaviour with a method invocation. In the current inference system this issue is avoided by constraining codes to have interfaces implemented by at most one class. This restriction will be relaxed by admitting that methods have multiple contracts, one for every possible implementation. In turn, method invocations are defined as the *union* of the possible contracts a method has.

Synchronisation on booleans. In addition to synchronisation on method invocations, `core ABS` permits synchronisations on Booleans, with the statement `await` e. When e is `False`, the execution of the method is suspended, and when it becomes `True`, the `await` terminates and the execution of the method may proceed. It is possible that the expression e refers to a field of an object that can be modified by another method. In this case, the `await` becomes synchronised with any method that may set the field to `true`. This subtle synchronisation pattern is difficult to infer and, for this reason, we have restricted the current release of `DF4ABS`.

Nevertheless, the current release of `DF4ABS` adopts a naive solution for `await` statements on booleans, namely let programmers annotate them with the dependencies they create. For example, consider the annotated code:

```
 class ClientJob(...) {
  Schedules schedules = EmptySet;
  ConnectionThread thread;
  ...
  Unit executeJob() {
    thread = ...;
    thread!command(ListSchedule);
    [thread] await schedules != EmptySet;
    ...
  }
}
```

The statement `await` compels the task to wait for `schedules` to be set to something different from the empty set. Since `schedules` is a field of the object, any concurrent thread (on that object) may update it. In the above case, the object that will modify the boolean guard is stored in the variable `thread`. Thus the annotation `[thread]` in the `await` statement. The current inference system of `DF4ABS` is extended with rules dealing with `await` on boolean guard and, of course, the correctness of the result depends on the correctness of the `await` annotations. A data-flow analysis of boolean guards in `await` statements may produce a set of cog dependencies that can be used in the inference rule of the corresponding statement. While this is an interesting issue, it will not be our primary concern in the near future.

Recursive object structures. In `core ABS`, like in any other object-oriented language, it is possible to define circular object structures, such as an object storing a pointer to itself in one of its fields. Currently, the contract inference system cannot deal with recursive structures, because the semi-unification process associates each object with a finite tree structure. In this way, it is not possible to capture circular definitions, such as the recursive ones. Note that this restriction still allows recursive definition of classes. We will investigate whether it is possible to extend the semi-unification process by associating *regular terms* [9] to objects in the semi-unification process. These regular terms might be derived during the inference by extending the `core ABS` code with annotations, as done for letting syntonisations on booleans.

Discussion. The above restrictions do not severely restrict both programming and the precision of our analysis. As we said, despite these limitations, we were able to apply our tool to the industrial-sized case study FAS from SDL Fredhopper and detect that it was dead-lock-free. It is also worth to observe that most of our restrictions can be removed with a simple extension of the current implementation. The restriction that may be challenging to remove is the one about recursive object structures, which requires the extension of semi-unification to such structures. We finally observe that other deadlock analysis tools have restrictions similar to those discussed in this section. For instance, `DECO` doesn't allow futures to be passed around (i.e. futures cannot be returned or put in an object's field) and constraints interfaces to be implemented by at most one class [13]. Therefore, while `DECO` supports the analysis in presence of field updates, our tool supports futures to be returned.

4 Contracts and the contract inference system

The deadlock detection framework we present in this paper relies on abstract descriptions, called *contracts*, that are extracted from programs by an inference system. The syntax of these descriptions, which is defined in Figure 8, uses *record names* X, Y, Z, $\cdots$, and *future names* f, f', $\cdots$. Future records $\mathbb{r}$, which encode the values of expressions in contracts, may be one of the following:

– a dummy value $\_$ that models primitive types,
– a record name X that represents a place-holder for a value and can be instantiated by substitutions,
– $[cog{:}c, \overline{x}{:}\overline{\mathbb{r}}]$ that defines an object with its cog name c and the values for fields and parameters of the object,
– and $c \rightsquigarrow \mathbb{r}$, which specifies that accessing $\mathbb{r}$ requires control of the cog c (and that the control is to be released once the method has been evaluated). The future record $c \rightsquigarrow \mathbb{r}$ is associated with method invocations: c is the cog of the object on which the method is invoked. The name c in $[cog{:}c, \overline{x}{:}\overline{\mathbb{r}}]$ and $c \rightsquigarrow \mathbb{r}$ will be called *root* of the future record.

Contracts $\mathbb{c}$ collect the method invocations and the dependencies inside statements. In addition to 0, $0.(c, c')$, and $0.(c, c')^{\mathtt{w}}$ that respectively represent the empty behaviour, the dependencies due to a `get` and an `await` operation, we have basic contracts that deal with method invocations. There are several possibilities:

– $\mathtt{C.m}\,\mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{r}'$ (resp. $\mathtt{C!m}\,\mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{r}'$) specifies that the method `m` of class `C` is going to be invoked *synchronously* (resp. *asynchronously*) on an object $\mathbb{r}$, with arguments $\overline{\mathbb{r}}$, and an object $\mathbb{r}'$ will be returned;
– $\mathtt{C!m}\,\mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{r}'.(c, c')$ indicates that the current method execution requires the termination of method `C!m` running on an object of cog c' in order to release the object of the cog c. This is the contract of an asynchronous method invocation followed by a `get` operation on the same future name.
– $\mathtt{C!m}\,\mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{r}'.(c, c')^{\mathtt{w}}$, indicating that the current method execution requires the termination of method `C.m` running on an object of cog c' in order to progress. This is the contract of an asynchronous method invocation followed by an `await` operation, and, possibly but not necessarily, by a `get` operation. In fact, a `get` operation on the same future name does not add any dependency, since it is guaranteed to succeed because of the preceding `await`.

The composite contracts $\mathbb{c} \fatsemi \mathbb{c}'$ and $\mathbb{c} + \mathbb{c}'$ define the abstract behaviour of sequential compositions and conditionals, respectively. The contract $\mathbb{c} \parallel \mathbb{c}'$ require a separate discussion because it models parallelism, which is not explicit in `core ABS` syntax. We will discuss this issue later on.

Example 4 As an example of contracts, let us discuss the terms:

(a) $\mathtt{C.m}[cog{:}c_1, x{:}[cog{:}c_2]]() \to [cog{:}c_1', x{:}[cog{:}c_2]] \fatsemi$
$\mathtt{C.m}[cog{:}c_3, x{:}[cog{:}c_4]]() \to [cog{:}c_3', x{:}[cog{:}c_4]]$;

(b) $\mathtt{C!m}[cog{:}c_1, x{:}[cog{:}c_2]]() \to [cog{:}c_1', x{:}[cog{:}c_2]].(c, c_1) \fatsemi$
$\mathtt{C!m}[cog{:}c_3, x{:}[cog{:}c_4]]() \to [cog{:}c_3', x{:}[cog{:}c_4]].(c, c_3)^{\mathtt{w}}$.

The contract (a) defines a sequence of two synchronous invocations of method `m` of class `C`. We notice that the cog names c_1' and c_3' are free: this indicates that `C.m` returns an object of a new cog. As we will see below, a `core ABS` expression with this contract is `x.m() ; y.m() ;`.

The contract (b) defines an asynchronous invocation of `C.m` followed by a `get` statement and an asynchronous one followed by an `await`. The cog c is the one of the caller. A `core ABS` expression retaining this contract is `u = x!m() ; w = u.get ; v = y!m() ; await v? ;`.

The inference of contracts uses two additional syntactic categories: $\mathbb{x}$ of future record values and $\mathbb{z}$ of typing values. The former one extends future records with *future names*, which are used to carry out the *alias analysis*. In particular, every local variable of methods and every object field and parameter of future type is associated to a future name. Assignments between these terms, such as $x = y$, amounts to copying future names instead of the corresponding values (x and y become aliases). The category $\mathbb{z}$ collects the typing values of future names, which are either $(\mathbb{r}, \mathbb{c})$, for *unsynchronised futures*, or $(\mathbb{r}, 0)^{\checkmark}$, for *synchronised ones* (see the comments below).

$$\begin{array}{llr}
\mathbb{r} ::= \_ \mid X \mid [cog{:}c, \overline{x}{:}\overline{\mathbb{r}}] \mid c \rightsquigarrow \mathbb{r} & & \text{future record} \\[1ex]
\mathbb{c} ::= 0 \mid 0.(c,c') \mid 0.(c,c')^{\mathtt{w}} \mid \mathtt{C.m}\ \mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{r}' \mid \mathtt{C!m}\ \mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{r}' \mid \mathtt{C!m}\ \mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{r}'.(c,c') & & \text{contract} \\
\quad \mid \mathtt{C!m}\ \mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{r}'.(c,c')^{\mathtt{w}} \mid \mathbb{c} \fatsemi \mathbb{c} \mid \mathbb{c} + \mathbb{c} \mid \mathbb{c} \parallel \mathbb{c} & & \\[1ex]
\mathbb{x} ::= \mathbb{r} \mid f & & \text{extended future record} \\[1ex]
\mathbb{z} ::= (\mathbb{r},\mathbb{c}) \mid (\mathbb{r},0)^{\checkmark} & & \text{future reference values}
\end{array}$$

Fig. 8 Syntax of future records and contracts.

The abstract behaviour of methods is defined by *method contracts* $\mathbb{r}(\overline{\mathbb{s}})\ \{\langle \mathbb{c}, \mathbb{c}' \rangle\}\ \mathbb{r}'$, where $\mathbb{r}$ is the future record of the receiver of the method, $\overline{\mathbb{s}}$ are the future records of the arguments, $\langle \mathbb{c}, \mathbb{c}' \rangle$ is the abstract behaviour of the body, where $\mathbb{c}$ is called *synchronised contract* and $\mathbb{c}'$ is called *unsynchronised contract*, and $\mathbb{r}'$ is the future record of the returned object.

Let us explain why method contracts use pairs of contracts. In `core ABS`, invocations in method bodies are of two types: (*i*) *synchronised*, that is the asynchronous invocation has a subsequent `await` or `get` operation in the method body and (*ii*) *unsynchronised*, the asynchronous invocation has no corresponding `await` or `get` in the same method body. (Synchronous invocations can be regarded as asynchronous invocations followed by a `get`.) For example, let

```
x = u!m() ;
await x? ;
y = v!m() ;
```

be the main function of a program (the declarations are omitted). In this statement, the invocation `u!m()` is synchronised before the execution of `v!m()`, which is unsynchronised. `core ABS` semantics tells us that the body of `u!m()` is *performed before* the body of `v!m()`. However, while this ordering holds for the synchronised part of `m`, it may not hold for the unsynchronised part. In particular, the unsynchronised part of `u!m()` may run *in parallel* with the body of `v!m()`. For this reason, in this case, our inference system returns the pair

$$\langle \mathtt{C!m}\ [cog{:}c']() \to \_.(c,c')^{\mathtt{w}}, \mathtt{C!m}\ [cog{:}c'']() \to \_ \rangle$$

where c, c' and c'' being the cog of the caller, of `u` and `v`, respectively. Letting $\mathtt{C!m}\ [cog{:}c']() \to \_ = \langle \mathbb{c}_u, \mathbb{c}'_u \rangle$ and $\mathtt{C!m}\ [cog{:}c'']() \to \_ = \langle \mathbb{c}_v, \mathbb{c}'_v \rangle$, one has (see Sections 5 and 6)

$$\begin{array}{l}
\langle \mathtt{C!m}\ [cog{:}c']() \to \_.(c,c')^{\mathtt{w}}, \mathtt{C!m}\ [cog{:}c'']() \to \_ \rangle \\
\quad = \quad \langle \mathbb{c}_u.(c,c')^{\mathtt{w}}, \mathbb{c}'_u \parallel (\mathbb{c}_v \fatsemi \mathbb{c}'_v) \rangle
\end{array}$$

that adds the dependency $(c,c')^{\mathtt{w}}$ to the synchronised contract of `u!m()` and makes the parallel (the $\parallel$ operator) of the unsynchronised contract of `u!m()` and the contract of `v!m()`. Of course, in alternative to separating *synchronised* and *unsynchronised contracts*, one might collect all the dependencies in a unique contract. This will imply that the dependencies in different configurations will be gathered in the same set, thus significantly reducing the precision of the analyses in Sections 5 and 6.

The above discussion also highlights the need of contracts $\mathbb{c} \parallel \mathbb{c}'$. In particular, this operator models *parallel behaviours*, which is not a first class operator in `core ABS`, while it is central in the semantics (the objects in the configurations). We illustrate the point with a statement similar to the above one, where we have swapped the second and third instruction

```
x = u!m() ;
y = v!m() ;
await x? ;
```

According to `core ABS` semantics, it is possible that the bodies of `u!m()` and of `v!m()` run in parallel by interleaving their executions. In fact, in this case, our inference system returns the pair of contracts (we keep the same notations as before)

$$\begin{array}{ll}
\langle\ \mathtt{C!m}\ [cog{:}c']() \to \_.(c,c')^{\mathtt{w}} \parallel \mathtt{C!m}\ [cog{:}c'']() \to \_\ , & \\
\quad \mathtt{C!m}\ [cog{:}c'']() \to \_ & \rangle
\end{array}$$

which turns out to be equivalent to

$$\mathtt{C!m1}\ [cog{:}c']() \to \_.(c,c')^{\mathtt{w}} \parallel \mathtt{C!m2}\ [cog{:}c']() \to \_$$

(see Sections 5 and 6).

The subterm $\mathbb{r}(\overline{\mathbb{s}})$ of the method contract is called *header*; $\mathbb{r}'$ is called *returned future record*. We assume that cog and record names in the header occur linearly. Cog and record names in the header *bind* the cog and record names in $\mathbb{c}$ and in $\mathbb{r}'$. The header and the returned future record, written $\mathbb{r}(\overline{\mathbb{s}}) \to \mathbb{r}'$, are called *contract signature*. In a method contract $\mathbb{r}(\overline{\mathbb{s}})\ \{\langle \mathbb{c}, \mathbb{c}' \rangle\}\ \mathbb{r}'$, cog and record names occurring in $\mathbb{c}$ or $\mathbb{c}'$ or $\mathbb{r}'$ may be *not bound* by header. These *free names* correspond to `new cog` instructions and will be replaced by fresh cog names during the analysis.

expressions and addresses

$$\text{(T-Var)}\ \frac{\Gamma(x) = \mathbb{x}}{\Gamma \vdash_c x : \mathbb{x}} \qquad \text{(T-Fut)}\ \frac{\Gamma(f) = \mathbb{z}}{\Gamma \vdash_c f : \mathbb{z}} \qquad \text{(T-Field)}\ \frac{x \notin \mathrm{dom}(\Gamma) \qquad \Gamma(\mathtt{this}.x) = \mathbb{r}}{\Gamma \vdash_c x : \mathbb{r}} \qquad \text{(T-Value)}\ \frac{\Gamma \vdash_c e : f \qquad \Gamma \vdash_c f : (\mathbb{r}, \mathbb{c})^{[\checkmark]}}{\Gamma \vdash_c e : \mathbb{r}}$$

$$\text{(T-Val)}\ \frac{e \quad \textit{primitive value or arithmetic-and-bool-exp}}{\Gamma \vdash_c e : \_} \qquad \text{(T-Pure)}\ \frac{\Gamma \vdash_c e : \mathbb{r}}{\Gamma \vdash_c e : \mathbb{r}, 0 \rhd \mathtt{true} \mid \Gamma}$$

expressions with side effects

$$\text{(T-Get)}\ \frac{\Gamma \vdash_c x : f \qquad \Gamma \vdash_c f : (\mathbb{r}, \mathbb{c}) \qquad X, c' \text{ fresh} \qquad \Gamma' = \Gamma[f \mapsto (\mathbb{r}, 0)^{\checkmark}]}{\Gamma \vdash_c x.\mathtt{get} : X, \mathbb{c}.(c, c') \parallel \mathit{unsync}(\Gamma') \rhd \mathbb{r} = c' \leadsto X \mid \Gamma'}$$

$$\text{(T-Get-tick)}\ \frac{\Gamma \vdash_c x : f \qquad \Gamma \vdash_c f : (\mathbb{r}, \mathbb{c})^{\checkmark} \qquad X, c' \text{ fresh}}{\Gamma \vdash_c x.\mathtt{get} : X, 0 \rhd \mathbb{r} = c' \leadsto X \mid \Gamma}$$

$$\text{(T-NewCog)}\ \frac{\begin{array}{c}\Gamma \vdash_c \overline{e} : \overline{\mathbb{r}} \\ \mathit{fields}(\mathtt{C}) = \overline{T\ x} \qquad \mathit{param}(\mathtt{C}) = \overline{T'\ x'} \qquad \overline{X}, c' \text{ fresh}\end{array}}{\Gamma \vdash_c \mathtt{new\ cog\ C}(\overline{e}) : [\mathit{cog}{:}c', \overline{x{:}X}, \overline{x'{:}\mathbb{r}}], 0 \rhd \mathtt{true} \mid \Gamma}$$

$$\text{(T-New)}\ \frac{\begin{array}{c}\Gamma \vdash_c \overline{e} : \overline{\mathbb{r}} \\ \mathit{fields}(\mathtt{C}) = \overline{T\ x} \qquad \mathit{param}(\mathtt{C}) = \overline{T'\ x'} \qquad \overline{X} \text{ fresh}\end{array}}{\Gamma \vdash_c \mathtt{new\ C}(\overline{e}) : [\mathit{cog}{:}c, \overline{x{:}X}, \overline{x'{:}\mathbb{r}}], 0 \rhd \mathtt{true} \mid \Gamma}$$

$$\text{(T-AInvk)}\ \frac{\begin{array}{c}\Gamma \vdash_c e : \mathbb{r} \qquad \Gamma \vdash_c \overline{e} : \overline{\mathbb{s}} \\ \mathit{class}(\mathit{types}(e)) = \mathtt{C} \qquad \mathit{fields}(\mathtt{C}) \cup \mathit{param}(\mathtt{C}) = \overline{T\ x} \qquad X, \overline{X}, c', f \text{ fresh}\end{array}}{\Gamma \vdash_c e!\mathtt{m}(\overline{e}) : f, 0 \rhd [\mathit{cog}{:}c', \overline{x{:}X}] = \mathbb{r} \wedge \mathtt{C.m} \preceq \mathbb{r}(\overline{\mathbb{s}}) \to X \mid \Gamma[f \mapsto (c' \leadsto X,\ \mathtt{C!m}\ \mathbb{r}(\overline{\mathbb{s}}) \to X)]}$$

$$\text{(T-SInvk)}\ \frac{\begin{array}{c}\Gamma \vdash_c e : \mathbb{r} \qquad \Gamma \vdash_c \overline{e} : \overline{\mathbb{s}} \\ \mathit{class}(\mathit{types}(e)) = \mathtt{C} \qquad \mathit{fields}(\mathtt{C}) \cup \mathit{param}(\mathtt{C}) = \overline{T\ x} \qquad X, \overline{X} \text{ fresh}\end{array}}{\Gamma \vdash_c e.\mathtt{m}(\overline{e}) : X, \mathtt{C.m}\ \mathbb{r}(\overline{\mathbb{s}}) \to X \parallel \mathit{unsync}(\Gamma) \rhd [\mathit{cog}{:}c', \overline{x{:}X}] = \mathbb{r} \wedge \mathtt{C.m} \preceq \mathbb{r}(\overline{\mathbb{s}}) \to X \mid \Gamma}$$

Fig. 9 Contract inference for expressions and expressions with side effects.

4.1 *Inference of contracts*

Contracts are extracted from core ABS programs by means of an inference algorithm. Figures 9 and 11 illustrate the set of rules. The following auxiliary operators are used:

– *fields*(C) and *param*(C) respectively return the sequence of fields and parameters and their types of a class C. Sometime we write $\mathit{fields}(\mathtt{C}) = \overline{T\,x},\ \overline{\mathtt{Fut<}T'\mathtt{>}\,x'}$ to separate fields with a non-future type by those with future types. Similarly for parameters;
– *types*(e) returns the type of an expression e, which is either an interface (when e is an object) or a data type;
– *class*(I) returns the unique (see the restriction *Interfaces* in Section 3) class implementing I; and
– *mname*($\overline{M}$) returns the sequence of method names in the sequence $\overline{M}$ of method declarations.

The inference algorithm uses constraints $\mathcal{U}$, which are defined by the following syntax

$$\mathcal{U} ::= \mathtt{true} \ \mid\ c = c' \ \mid\ \mathbb{r} = \mathbb{r}' \ \mid\ \mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{s} \preceq \mathbb{r}'(\overline{\mathbb{r}}') \to \mathbb{s}' \ \mid\ \mathcal{U} \wedge \mathcal{U}$$

where true is the constraint that is always true; $\mathbb{r} = \mathbb{r}'$ is a classic unification constraint between terms; $\mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{s} \preceq \mathbb{r}'(\overline{\mathbb{r}}') \to \mathbb{s}'$ is a *semi-unification* constraint; the constraint $\mathcal{U} \wedge \mathcal{U}'$ is the conjunction of $\mathcal{U}$ and $\mathcal{U}'$. We use *semi-unification* constraints [21] to deal with method invocations: basically, in $\mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{s} \preceq \mathbb{r}'(\overline{\mathbb{r}}') \to \mathbb{s}'$, the left hand side of the constraint corresponds to the method's formal parameter, $\mathbb{r}$ being the record of this, $\overline{\mathbb{r}}$ being the records of the parameters and $\mathbb{r}'$ being the record of the returned value, while the right hand side corresponds to the actual parameters of the call, and the actual returned value. The meaning of this constraint is that the actual parameters and returned value must match the specification given by the formal parameters, like in a standard unification: the necessity of semi-unification appears when we call several times the same method. Indeed, there, unification would require that the actual parameters of the different calls must all have the same records, while with semi-unification all method calls are managed independently.

The judgments of the inference algorithm have a typing context Γ mapping variables to extended future records, future names to future name values and methods to their signatures. They have the following form:

– $\Gamma \vdash_c e : \mathbb{x}$ for pure expressions e and $\Gamma \vdash_c f : \mathbb{z}$ for future names f, where c is the cog name of the statements

$$\text{T-SKIP} \quad \Gamma \vdash_c \texttt{skip} : 0 \rhd \texttt{true} \mid \Gamma$$

$$\text{(T-FIELD-RECORD)} \quad \frac{x \notin \mathrm{dom}(\Gamma) \qquad \Gamma(\texttt{this}.x) = \mathbb{r} \qquad \Gamma \vdash_c z : \mathbb{r}', \mathbb{c} \rhd \mathcal{U} \mid \Gamma'}{\Gamma \vdash_c x = z : \mathbb{c} \rhd \mathcal{U} \wedge \mathbb{r} = \mathbb{r}' \mid \Gamma'}$$

$$\text{(T-VAR-RECORD)} \quad \frac{\Gamma(x) = \mathbb{r} \qquad \Gamma \vdash_c z : \mathbb{r}', \mathbb{c} \rhd \mathcal{U} \mid \Gamma'}{\Gamma \vdash_c x = z : \mathbb{c} \rhd \mathcal{U} \mid \Gamma'[x \mapsto \mathbb{r}']}$$

$$\text{(T-VAR-FUTURE)} \quad \frac{\Gamma(x) = f \qquad \Gamma \vdash_c z : f', \mathbb{c} \rhd \mathcal{U} \mid \Gamma'}{\Gamma \vdash_c x = z : \mathbb{c} \rhd \mathcal{U} \mid \Gamma'[x \mapsto f']}$$

$$\text{(T-AWAIT)} \quad \frac{\Gamma \vdash_c e : f \qquad \Gamma \vdash_c f : (\mathbb{r}, \mathbb{c}) \qquad X, c' \text{ fresh} \qquad \Gamma' = \Gamma[f \mapsto (\mathbb{r}, 0)^{\checkmark}]}{\Gamma \vdash_c \texttt{await}\ e? : \mathbb{c}.(c, c')^{\mathtt{w}} \parallel \mathit{unsync}(\Gamma') \rhd \mathbb{r} = c' \rightsquigarrow X \mid \Gamma'}$$

$$\text{(T-AWAIT-TICK)} \quad \frac{\Gamma \vdash_c e : f \qquad \Gamma \vdash_c f : (\mathbb{r}, \mathbb{c})^{\checkmark} \qquad X, c' \text{ fresh}}{\Gamma \vdash_c \texttt{await}\ e? : 0 \rhd \mathbb{r} = c' \rightsquigarrow X \mid \Gamma}$$

$$\text{(T-IF)} \quad \frac{\begin{array}{c} \Gamma \vdash_c e : \texttt{Bool} \qquad \Gamma \vdash_c s_1 : \mathbb{c}_1 \rhd \mathcal{U}_1 \mid \Gamma_1 \qquad \Gamma \vdash_c s_2 : \mathbb{c}_2 \rhd \mathcal{U}_2 \mid \Gamma_2 \\ \mathcal{U} = \Big(\bigwedge_{x \in \mathrm{dom}(\Gamma)} \Gamma_1(x) = \Gamma_2(x) \Big) \wedge \Big(\bigwedge_{x \in \texttt{Fut}(\Gamma)} \Gamma_1(\Gamma_1(x)) = \Gamma_2(\Gamma_2(x)) \Big) \qquad \Gamma' = \Gamma_1 + \Gamma_2|_{\{ f \mid f \notin \Gamma_2(\texttt{Fut}(\Gamma))\}} \end{array}}{\Gamma \vdash_c \texttt{if}\ e\ \{\, s_1 \,\}\ \texttt{else}\ \{\, s_2 \,\} : \mathbb{c}_1 + \mathbb{c}_2 \rhd \mathcal{U}_1 \wedge \mathcal{U}_2 \wedge \mathcal{U} \mid \Gamma'}$$

$$\text{(T-SEQ)} \quad \frac{\Gamma \vdash_c s_1 : \mathbb{c}_1 \rhd \mathcal{U}_1 \mid \Gamma_1 \qquad \Gamma_1 \vdash_c s_2 : \mathbb{c}_2 \rhd \mathcal{U}_2 \mid \Gamma_2}{\Gamma \vdash_c s_1; s_2 : \mathbb{c}_1 \fatsemi \mathbb{c}_2 \rhd \mathcal{U}_1 \wedge \mathcal{U}_2 \mid \Gamma_2}$$

$$\text{(T-RETURN)} \quad \frac{\Gamma \vdash_c e : \mathbb{r} \qquad \Gamma(\texttt{destiny}) = \mathbb{r}'}{\Gamma \vdash_c \texttt{return}\ e : 0 \rhd \mathbb{r} = \mathbb{r}' \mid \Gamma}$$

Fig. 10 Contract inference for statements.

object executing the expression and $\mathbb{x}$ and $\mathbb{z}$ are their inferred values.

– $\Gamma \vdash_c z : \mathbb{r}, \mathbb{c} \rhd \mathcal{U} \mid \Gamma'$ for expressions with side effects z, where c, and $\mathbb{x}$ are as for pure expressions e, $\mathbb{c}$ is the contract for z created by the inference rules, $\mathcal{U}$ is the generated constraint, and Γ' is the environment Γ *with updates* of variables and future names. We use the same judgment for pure expressions; in this case $\mathbb{c} = 0$, $\mathcal{U} = \texttt{true}$ and $\Gamma' = \Gamma$.
– for statements s: $\Gamma \vdash_c s : \mathbb{c} \rhd \mathcal{U} \mid \Gamma'$ where c, $\mathbb{c}$ and $\mathcal{U}$ are as before, and Γ' is the environment obtained after the execution of the statement. The environment may change because of variable updates.

Since Γ is a function, we use the standard predicates $x \in \mathrm{dom}(\Gamma)$ or $x \notin \mathrm{dom}(\Gamma)$. Moreover, given a function Γ, we define $\Gamma[x \mapsto \mathbb{x}]$ to be the following function

$$\Gamma[x \mapsto \mathbb{x}](y) = \begin{cases} \mathbb{x} & \text{if } y = x \\ \Gamma(y) & \text{otherwise} \end{cases}$$

We also let $\Gamma|_{\{x_1, \cdots, x_n\}}$ be the function

$$\Gamma|_{\{x_1, \cdots, x_n\}}(y) = \begin{cases} \Gamma(y) & \text{if } y \in \{x_1, \cdots, x_n\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Moreover, provided that $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma') = \varnothing$, the environment $\Gamma + \Gamma'$ be defined as follows

$$(\Gamma + \Gamma')(x) \stackrel{def}{=} \begin{cases} \Gamma(x) & \text{if } x \in \mathrm{dom}(\Gamma) \\ \Gamma'(x) & \text{if } x \in \mathrm{dom}(\Gamma') \end{cases}$$

Finally, we write $\Gamma(\texttt{this}.x) = \mathbb{x}$ whenever $\Gamma(\texttt{this}) = [\mathit{cog}{:}c, x : \mathbb{x}, \overline{x} : \overline{\mathbb{x}'}]$ and we let

$$\texttt{Fut}(\Gamma) \stackrel{def}{=} \{x \mid \Gamma(x) \text{ is a future name}\}$$

$$\mathit{unsync}(\Gamma) \stackrel{def}{=} \mathbb{c}_1 \parallel \cdots \parallel \mathbb{c}_n$$

where $\{\mathbb{c}_1, \cdots, \mathbb{c}_n\} = \{\mathbb{c}' \mid \text{there are } f, \mathbb{r} : \ \Gamma(f) = (\mathbb{r}, \mathbb{c}')\}$.

The inference rules for expressions and future names are reported in Figure 9. They are straightforward, except for (T-VALUE) that performs the dereference of variables and return the future record stored in the future name of the variable. (T-PURE) lifts the judgment of a pure expression to a judgment similar to those for expressions with side-effects. This expedient allows us to simplify rules for statements.

Figure 9 also reports inference rules for expressions with side effects. Rule (T-GET) deals with the $x.\texttt{get}$ synchronisation primitive and returns the contract $\mathbb{c}.(c, c') \parallel \mathit{unsync}(\Gamma)$, where $\mathbb{c}$ is stored in the future name of x and (c, c') represents a dependency between the cog of the object executing the expression and the root of the expression. The constraint $\mathbb{r} = c' \rightsquigarrow X$ is used to extract the root c' of $\mathbb{r}$. The contract $\mathbb{c}$ may have two shapes: either (i) $\mathbb{c} = \mathtt{C!m}\,\mathbb{r}(\overline{\mathbb{s}}) \to \mathbb{r}'$ or (ii) $\mathbb{c} = 0$. The subterm $\mathit{unsync}(\Gamma)$ lets us collect all the contracts in Γ *that are stored in future names that are not check-marked.* In fact, these contracts correspond to previous asynchronous invocations without any corresponding synchronisation (`get` or `await` operation) in the body. The evaluations of these invocations may interleave with the evaluation of the expression x.`get`. For this reason, the intended meaning of *unsync*(Γ) is that the dependencies generated by the invocations must be collected *together* with those generated by $\mathbb{c}.(c, c')$. We also observe that the rule updates the environment by check-marking the value of the future name of x and by replacing the contract with 0 (because the synchronisation has been already performed). This allows subsequent `get` (and `await`) operations on the same future name not to modify the contract (in fact, in this case they are operationally equivalent to the `skip` statement) – see (T-Get-Tick).

Rule (T-NewCog) returns a record with a new cog name. This is in contrast with (T-New), where the cog of the returned record is the *same* of the object executing the expression [2].

Rule (T-AInvk) derives contracts for asynchronous invocations. Since the dependencies created by these invocations influence the dependencies of the synchronised contract only if a subsequent `get` or `await` operation is performed, the rule stores the invocation into a fresh future name of the environment and returns the contract 0. This models `core ABS` semantics that lets asynchronous invocations be synchronised by explicitly getting or awaiting on the corresponding future variable, see rules (T-Get) and (T-Await). The future name storing the invocation is returned by the judgment. On the contrary, in rule (T-SInvk), which deals with synchronous invocations, the judgement returns a contract that is the invocation (because the corresponding dependencies must be added to the current ones) in parallel with the unsynchronised asynchronous invocations stored in Γ.

The inference rules for statements are collected in Figure 10. The first three rules define the inference of contracts for assignment. There are two types of assignments: those updating fields and parameters of the `this` object and the other ones. For every type, we need to address the cases of updates with values that are expressions (with side effects) (rules (T-Field-Record) and (T-Var-Record)), or future names (rule (T-Var-Future)). Rules for fields and parameters updates enforce that their future records are unchanging, as discussed in Section 3. Rule (T-Var-Future), define the management of aliases: future variables are always updated with future names and never with future names' values.

Rule (T-Await) and (T-AwaitTick) deal with the `await` synchronisation when applied to a simple future lookup x?. They are similar to the rules (T-Get) and (T-Get-Tick).

Rule (T-If) defines contracts for conditionals. In this case we collect the contracts $\mathbb{c}_1$ and $\mathbb{c}_2$ of the two branches, with the intended meaning that the dependencies defined by $\mathbb{c}_1$ and $\mathbb{c}_2$ are always kept separated. As regards the environments, the rule constraints the two environments Γ_1 and Γ_2 produced by typing of the two branches to *be the same* on variables in $\mathrm{dom}(\Gamma)$ *and* on the values of future names bound to variables in $\mathtt{Fut}(\Gamma)$. However, the two branches may have different unsynchronised invocations that are not bound to any variable. The environment $\Gamma_1 + \Gamma_2|_{\{f \mid f \notin \Gamma_2(\mathtt{Fut}(\Gamma))\}}$ allows us to collect all them.

Rule (T-Seq) defines the sequential composition of contracts. Rule (Return) constrains the record of `destiny`, which is an identifier introduced by (T-Method), shown in Figure 11, for storing the return record.

The rules for method and class declarations are defined in Figure 11. Rule (T-Method) derives the method contract of T `m` $(\overline{T}\ \overline{x})\{\overline{T'}\,\overline{u}; s\}$ by typing s in an environment extended with `this`, `destiny` (that will be set by `return` statements, see (T-Return)), the arguments $\overline{x}$, and the local variables $\overline{u}$. In order to deal with alias analysis of future variables, we separate fields, parameters, arguments and local variables with future types from the other ones. In particular, we associate future names to the former ones and bind future names to record variables. As discussed above, the abstract behaviour of the method body is a pair of contracts, which is $\langle \mathbb{c}, \textit{unsync}(\Gamma'')\rangle$ for (T-Method). This term *unsync*(Γ'') collects all the contracts in Γ'' *that are stored in future names that are not check-marked.* In fact, these contracts correspond to asynchronous invocations without any synchronisation (`get` or `await` operation) in the body. These invocations will be evaluated *after* the termination of the body – they are the *unsynchronised contract.*

The rule (T-Class) yields an *abstract class table* that associates a method contract with every method name. It is this abstract class table that is used by our analysers in Sections 5 and 6. The rule (T-Program) derives the contract of a `core ABS` program by typing the main function in the same way as it was a body of a method.

[2] It is worth to recall that, in `core ABS`, the creation of an object, either with a `new` or with a `new cog`, amounts to executing the method `init` of the corresponding class, whenever defined (the `new` performs a synchronous invocation, the `new cog` performs an asynchronous one). In turn, the termination of `init` triggers the execution of the method `run`, if present. The method `run` is asynchronously invoked when `init` is absent. Since `init` may be regarded as a method in `core ABS`, the inference system in our tool explicitly introduces a synchronous invocation to `init` in case of `new` and an asynchronous one in case of `new cog`. However, for simplicity, we overlook this (simple) issue in the rules (T-New) and (T-NewCog), acting as if `init` and `run` are always absent.

(T-Method)

$$\frac{\begin{array}{c} \mathit{fields}(\mathtt{C}) \cup \mathit{param}(\mathtt{C}) = \overline{T_f\ x}\ \overline{\mathtt{Fut<}T'_f\mathtt{>}\ x'} \qquad c, \overline{X}, \overline{X'}, \overline{Y}, \overline{Y'}, \overline{W}, \overline{W'}, \overline{f}, \overline{f'}, \overline{f''}, Z\ \mathit{fresh} \\ \Gamma' = \overline{y{:}Y} + \overline{y'{:}f'} + \overline{w{:}W} + \overline{w'{:}f''} + \overline{f{:}(\overline{X'},0)} + \overline{f'{:}(\overline{Y'},0)} + \overline{f''{:}(\overline{W'},0)} \\ \Gamma + \mathtt{this} : [\mathit{cog}{:}c,\ \overline{x{:}X}, \overline{x{:}f}] + \Gamma' + \mathtt{destiny} : Z \vdash_c s : \mathbb{c} \rhd \mathcal{U} \,|\, \Gamma'' \qquad \overline{T},\ \overline{T_f},\ \overline{T_l} \text{ are not future types} \end{array}}{\mathtt{C}, \Gamma \vdash T\ \mathtt{m}\ (\overline{T\ y}, \overline{\mathtt{Fut<}T'\mathtt{>}\ y'})\{\overline{T_l\ w}; \overline{\mathtt{Fut<}T'_l\mathtt{>}\ w'};\ s\}\ :\ \begin{array}{c} [\mathit{cog} : c, \overline{x{:}X},\ \overline{x'{:}X'}](\overline{Y}, \overline{Y'})\{\langle \mathbb{c}, \mathit{unsync}(\Gamma'')\rangle\}\ Z \\ \rhd \mathcal{U} \wedge [\mathit{cog} : c, \overline{x{:}X},\ \overline{x'{:}X'}](\overline{Y}, \overline{Y'}) \to Z = \mathtt{C.m} \end{array}}$$

(T-Class)

$$\frac{\mathtt{C}, \Gamma \vdash \overline{M} : \overline{\mathbb{C}} \rhd \overline{\mathcal{U}}}{\Gamma \vdash \mathtt{class}\ \mathtt{C}(\overline{T\ x})\ \{\overline{T'\ x'};\quad \overline{M}\} : \overline{\mathtt{C}.\mathit{mname}(M) \mapsto \mathbb{C}} \rhd \overline{\mathcal{U}}}$$

(T-Program)

$$\frac{\Gamma \vdash \overline{C} : \overline{S} \rhd \overline{\mathcal{U}} \qquad \overline{X}, \overline{X'}, \overline{f}\ \mathit{fresh} \qquad \Gamma + \overline{x{:}X} + \overline{x'{:}f} + \overline{f{:}(X',0)} \vdash_{\mathsf{start}} s : \mathbb{c} \rhd \mathcal{U} \,|\, \Gamma' \qquad \overline{T} \text{ are not future types}}{\Gamma \vdash \overline{I}\ \overline{C}\ \{\overline{T\ x};\ \overline{\mathtt{Fut<}T'\mathtt{>}\ x'};\ s\} : \overline{S}, \langle \mathbb{c}, \mathit{unsync}(\Gamma')\rangle \rhd \mathcal{U} \wedge \overline{\mathcal{U}}}$$

Fig. 11 Contract rules of method and class declarations and programs.

The contract class tables of the classes in a program derived by the rule (Class), will be noted CCT. We will address the contract of `m` of class `C` by CCT(`C.m`). In the following, we assume that every `core ABS` program is a triple $(\text{CT}, \{\overline{T\ x\ ;\ s}\}, \text{CCT})$, where CT is the class table, $\{\overline{T\ x\ ;\ s}\}$ is the main function, and CCT is its contract class table. By rule (Program), analysing (the deadlock freedom of) a program, amounts to verifying the contract of the main function with a record for `this` which associates a special cog name called start to the *cog* field (start is intended to be the cog name of the object *start*).

Example 5 The methods of `Math` in Figure 5 have the following contracts, once the constraints are solved (we always simplify $\mathbb{c} \fatsemi 0$ into $\mathbb{c}$):

– `fact_g` has contract

$$[\mathit{cog}{:}c](\_)\ \{\langle 0 + \mathtt{Math!fact\_g}\ [\mathit{cog}{:}c](\_) \to \_.(c,c), 0\rangle\}\ \_.$$

The name c in the header refers to the cog name associated with `this` in the code, and binds the occurrences of c in the body. The contract body has a recursive invocation to `fact_g`, which is performed on an object in the same cog c and followed by a `get` operation. This operation introduces a dependency (c, c). We observe that, if we replace the statement `Fut<Int> x = this!fact_g(n-1)` in `fact_g` with `Math z = new Math() ; Fut<Int> x = z!fact_g(n-1)`, we obtain the same contract as above because the new object is in the same cog as `this`.

– `fact_ag` has contract

$$[\mathit{cog}{:}c](\_)\ \{\langle 0 + \mathtt{Math!fact\_ag}\ [\mathit{cog}{:}c](\_) \to \_.(c,c)^{\mathtt{w}}, 0\rangle\}\ \_.$$

In this case, the presence of an `await` statement in the method body produces a dependency $(c, c)^{\mathtt{w}}$. The subsequent `get` operation does not introduce any dependency because the future name has a check-marked value in the environment. In fact, in this case, the success of `get` is guaranteed, provided the success of the `await` synchronisation.

– `fact_nc` has contract

$$[\mathit{cog}{:}c](\_)\ \{\langle 0 + \mathtt{Math!fact\_nc}\ [\mathit{cog}{:}c'](\_) \to \_.(c,c'), 0\rangle\}\ \_.$$

This method contract differs from the previous ones in that the receiver of the recursive invocation is a free name (i.e., it is not bound by c in the header). This because the recursive invocation is performed on an object of a new cog (which is therefore different from c). As a consequence, the dependency added by the `get` relates the cog c of `this` with the new cog c'.

Example 6 Figure 12 displays the contracts of the methods of class `CpxSched` in Figure 6.

According to the contract of the main function, the two invocations of `m2` are second arguments of $\|$ operators. This will give rise, in the analysis of contracts, to the union of the corresponding cog relations.

We notice that the inference system of contracts discussed in this section is modular because, when programs are organised in different modules, it partially supports the separate contract inference of modules with a well-founded ordering relation (for example, if there are two modules, classes in the second module use definitions or methods in the first one, but not conversely). In this case, if a module `B` includes a module `A` then a patch to a class of `B` amounts to inferring contracts for `B` only. On the contrary, a patch to a class of `A` may also require a new contract inference of `B`.

– method `m1` has contract

$$[\mathit{cog}{:}c, \mathtt{x} : [\mathit{cog}{:}c', \mathtt{x} : X]]([\mathit{cog}{:}c'', \mathtt{x} : Y])\ \{\langle 0, \mathbb{c}\rangle\}\ c' \rightsquigarrow \_.$$

where $\mathbb{c} = \mathtt{CpxSched!m2}\ [\mathit{cog}{:}c'', \mathtt{x} : Y]([\mathit{cog}{:}c', \mathtt{x} : X]) \rightarrow c'' \rightsquigarrow \_ \parallel \mathtt{CpxSched!m2}\ [\mathit{cog}{:}c', \mathtt{x} : X]([\mathit{cog}{:}c'', \mathtt{x} : Y]) \rightarrow c' \rightsquigarrow \_$

– method `m2` has contract

$$[\mathit{cog}{:}c, \mathtt{x} : X]([\mathit{cog}{:}c', \mathtt{x} : Y])\ \{\langle \mathtt{CpxSched!m3}\ [\mathit{cog}{:}c', \mathtt{x} : Y](\_) \rightarrow \_.(c, c'), 0\rangle\}\ \_.$$

– method `m3` has contract

$$[\mathit{cog}{:}c, \mathtt{x} : X](\,)\ \{\langle 0, 0\rangle\}\ \_.$$

Fig. 12 Contracts of `CpxSched`.

4.2 Correctness results

In our system, the ill-typed programs are those manifesting a failure of the semiunification process, which does not address misbehaviours. In particular, a program may be well-typed and still manifest a deadlock. In fact, in systems with *behavioural types*, one usually demonstrates that

1. in a well-typed program, every configuration *cn* has a behavioural type, let us call it `bt`(*cn*);
2. if $cn \to cn'$ then there is a relationship between `bt`(*cn*) and `bt`(*cn′*);
3. the relationship in 2 preserves a given property (in our case, deadlock-freedom).

Item 1, in the context of the inference system of this section, means that the program has a contract class table. Its proof needs a contract system for configurations, which we have defined in Appendix A. The theorem corresponding to this item is Theorem 3.

Item 2 requires the definition of a relation between contracts, called *later stage relation* in Appendix A. This later stage relation is a syntactic relationship between contracts whose basic law is that a method invocation is larger than the instantiation of its method contract (the other laws, except $0 \trianglelefteq \mathbb{c}$ and $\mathbb{c}_i \trianglelefteq \mathbb{c}_1 + \mathbb{c}_2$, are congruence laws).

The statement that relates the later stage relationship to `core ABS` reduction is Theorem 4. It is worth to observe that all the theoretical development up-to this point are useless if the later stage relation conveyed no relevant property. This is the purpose of item 3, which requires the definition of *contract models* and the proof that deadlock-freedom is preserved by the models of contracts in later stage relation. The reader can find the proofs of these statements in the Appendices B and C (they correspond to the two analysis techniques that we study).

5 The fix-point analysis of contracts

The first algorithm we define to analyse contracts uses the standard Knaster-Tarski fixpoint technique. We first give an informal introduction of the notion used in the analysis, and start to formally define our algorithm in Subsection 5.1 (a simplified version of the algorithm may be found in [17], see also Section 8).

Based on a contract class table and a main contract (both produced by the inference system in Section 4), our fixpoint algorithm generates models that encode the dependencies between cogs that may occur during the program's execution. These models, called *lams* (an acronym for deadLock Analysis Models [18,19] are sets of relations between cog names, each relation representing a possible configuration of program's execution. Consider for instance the main function:

```
{
  I x ; I y ; Fut<Unit> f ;
  x = new cog C() ;
  y = new cog C() ;
  f = x!m() ;
  await f? ;
  f = y!m() ;
  await f? ;
}
```

In this case, the configurations of the program may be represented by two relations: one containing a dependency between the cog name *start* and the cog name of x and the other containing a dependency between *start* and the cog name of y. This would be represented by the following lam (where c_x and c_y respectively being the cog names of x and y):

$$[\![(c, c_x)^{\mathtt{w}}]\!],\ [\![(c, c_y)^{\mathtt{w}}]\!]$$

(in order to ease the parsing of the formula, we are representing relations with the notation $[\![\cdot]\!]$ and we have removed the outermost curly brackets). Our algorithm, being a fixpoint analysis, returns *the* lam of a program by computing a sequence of approximants. In particular, the algorithm performs the following steps:

1. compute a new approximant of the lam of every method using the abstract class table and the previously computed lams;
2. reiterate step 1 till a fixed approximant – say n (if a fixpoint is found before, go to 4);
3. when the n-th approximant is computed then *saturate*, i.e. compute the next approximants by reusing the same cog names (then a fixpoint is eventually found);
4. replace the method invocations in the main contract with the corresponding values; and
5. analyse the result of 4 by looking for a circular dependency in one of the relations of the computed lam (in such case, a possible deadlock is detected).

The critical issue of our algorithm is the creation of fresh cog names at each step 1, because of free names in method contracts (that correspond to new cogs created during method's execution). For example, consider the contract of `Math.fact_nc` that has been derived in Section 4

$$\begin{array}{l}\texttt{Math.fact\_nc}\,[\mathit{cog}{:}c](\_)\{\\ \quad\langle 0 + \texttt{Math!fact\_nc}\,[\mathit{cog}{:}c'](\_) \to \_.(c,c'),0\rangle \quad \}\_\end{array}$$

According to our definitions, the cog name c' is free. In this case, our fixpoint algorithm will produce the following sequence of lams when computing the model of $\texttt{Math.fact\_nc}[\mathit{cog}{:}c_0](\_)$:

$$\begin{array}{ll}
\textit{approximant 0}: & \langle\, [\varnothing]\,, 0\rangle\\
\textit{approximant 1}: & \langle\, [(c_0,c_1)]\,, 0\rangle\\
\textit{approximant 2}: & \langle\, [(c_0,c_1),(c_1,c_2)]\,, 0\rangle\\
\cdots & \\
\textit{approximant n}: & \langle\, [(c_0,c_1),(c_1,c_2),\cdots(c_{n-1},c_n)]\,, 0\rangle\\
\cdots &
\end{array}$$

While every lam in the above sequence is strictly larger than the previous one, an upper bound element cannot be obtained by iterating the process. Technically, the lam model *is not a complete partial order* (the ascending chains of lams may have infinite length and no upper bound).

In order to circumvent this issue and to get a decision on deadlock-freedom in a finite number of steps, we use a *saturation argument.* If the n-th approximant is not a fixpoint, then the $(n+1)$-th approximant is computed by *reusing the same cog names used by the n-th approximant* (no additional cog name is created anymore). Similarly for the $(n+2)$-th approximant till a fixpoint is reached (by straightforward cardinality arguments, the fixpoint does exist, in this case). This fixpoint is called *the saturated state.*

For example, for $\texttt{Math.fact\_nc}[\mathit{cog}{:}c_0](\_)$, the n-th approximant returns the pairs of lams

$$\langle\, [(c_0,c_1),\cdots,(c_{n-1},c_n)]\,, 0\rangle\,.$$

Saturating at this stage yields the lam

$$\langle\, [(c_0,c_1),\cdots,(c_{n-1},c_n),(c_1,c_1)]\,, 0\rangle$$

that contains a circular dependency – the pair (c_1,c_1) – revealing a potential deadlock in the corresponding program. Actually, in this case, this circularity is a *false positive* that is introduced by the (over)approximation: the original code never manifests a deadlock.

Note finally that a lam is the result of the analysis of one contract. Hence, to match the structures that are generated during the type inference, our analysis uses three extensions of lams: (i) a pair of lams $\langle\mathcal{L},\mathcal{L}'\rangle$ for analysing pairs of contracts $\langle \mathbb{c},\mathbb{c}'\rangle$; (ii) *parameterised* pair of lams $\lambda\overline{c}.\langle\mathcal{L},\mathcal{L}'\rangle$ for analysing methods: here, $\overline{c}$ are the cog names in the header of the method (the `this` object and the formal parameters), and $\langle\mathcal{L},\mathcal{L}'\rangle$ is the result of the analysis of the contract pair typing the method; and (iii) lam tables $(\cdots,\lambda\overline{c_i}.\langle\mathcal{L}_i,\mathcal{L}'_i\rangle,\cdots)$ that maps each method in the program to its current approximant. We observe that $\lambda\overline{c}.\langle\mathcal{L},\mathcal{L}'\rangle$ is $\langle\mathcal{L},\mathcal{L}'\rangle$ whenever $\overline{c}$ is empty.

5.1 *Lams and lam operations*

The following definition formally introduce the notion of lam.

Definition 3 A relation on cog names is a set of pairs either of the form (c_1,c_2) or of the form $(c_1,c_2)^{\texttt{w}}$, generically represented as $(c_1,c_2)^{[\texttt{w}]}$. We denote such relation by $[(c_{i_0},c_{i_1})^{[\texttt{w}]},\cdots,(c_{i_{n-1}},c_{i_n})^{[\texttt{w}]}]$. A *lam*, ranged over $\mathcal{L}$, $\mathcal{L}'$, $\cdots$, is a set of relations on cog names. Let 0 be the lam $[\varnothing]$ and let $\mathit{cog\_names}(\mathcal{L})$ be the cog names occurring in $\mathcal{L}$.

The pre-order relation between lam, pair of lams and parameterised pair of lam, noted $\Subset$ is defined below. This pre-order is central to prove that our algorithm indeed computes a fix point.

Definition 4 Let $\mathcal{L}$ and $\mathcal{L}'$ be lams and κ be an injective function between cog names. We note $\mathcal{L}\Subset_\kappa\mathcal{L}'$ iff for every $L\in\mathcal{L}$ there is $L'\in\mathcal{L}'$ with $\kappa(L)\subseteq L'$. Let

– $\lambda\overline{c}.\langle\mathcal{L}_1,\mathcal{L}'_1\rangle \Subset_\kappa \lambda\overline{c}.\langle\mathcal{L}_2,\mathcal{L}'_2\rangle$ iff κ is the identity on $\overline{c}$ and $\langle\mathcal{L}_1,\mathcal{L}'_1\rangle \Subset_\kappa \langle\mathcal{L}_2,\mathcal{L}'_2\rangle$.

Let also $\Subset$ be the relation

– $\mathcal{L}\Subset\mathcal{L}'$ iff there is κ such that $\mathcal{L}\Subset_\kappa\mathcal{L}'$;
– $\lambda\overline{c}.\langle\mathcal{L}_1,\mathcal{L}'_1\rangle \Subset \lambda\overline{c}.\langle\mathcal{L}_2,\mathcal{L}'_2\rangle$ iff there is κ such that $\lambda\overline{c}.\langle\mathcal{L}_1,\mathcal{L}'_1\rangle \Subset_\kappa \lambda\overline{c}.\langle\mathcal{L}_2,\mathcal{L}'_2\rangle$.

$$
\begin{array}{lrl}
\text{[extension]} & \mathcal{L}\&(c,c')^{[\mathtt{w}]} \overset{def}{=} & \{L \cup \{(c,c')^{[\mathtt{w}]}\} \mid L \in \mathcal{L}\}. \\
\text{[parallel]} & \mathcal{L}\|\mathcal{L}' \overset{def}{=} & \{L \cup L' \mid L \in \mathcal{L} \text{ and } L' \in \mathcal{L}'\} \\
\text{[extension (on pairs of lams)]} & \langle \mathcal{L}, \mathcal{L}'\rangle\&(c,c')^{[\mathtt{w}]} \overset{def}{=} & \langle \mathcal{L}\&(c,c')^{[\mathtt{w}]}, \mathcal{L}'\rangle \\
\text{[parallel (on pairs of lams)]} & \langle \mathcal{L}_1, \mathcal{L}_1'\rangle \parallel \langle \mathcal{L}_2, \mathcal{L}_2'\rangle \overset{def}{=} & \langle (\mathcal{L}_1 \cup \mathcal{L}_1')\|(\mathcal{L}_2 \cup \mathcal{L}_2'), 0\rangle \\
\text{[sequence (on pairs of lams)]} & \langle \mathcal{L}_1, \mathcal{L}_1'\rangle \fatsemi \langle \mathcal{L}_2, \mathcal{L}_2'\rangle \overset{def}{=} & \begin{cases} \langle \mathcal{L}_1, \mathcal{L}_1'\|\mathcal{L}_2'\rangle & \text{if } \mathcal{L}_2 = 0 \\ \langle \mathcal{L}_1 \cup (\mathcal{L}_2\|\mathcal{L}_1'), \mathcal{L}_1'\|\mathcal{L}_2'\rangle & \text{otherwise} \end{cases} \\
\text{[plus (on pairs of lams)]} & \langle \mathcal{L}_1, \mathcal{L}_1'\rangle + \langle \mathcal{L}_2, \mathcal{L}_2'\rangle \overset{def}{=} & \langle \mathcal{L}_1 \cup \mathcal{L}_2, \mathcal{L}_1' \cup \mathcal{L}_2'\rangle.
\end{array}
$$

Fig. 13 Lam operations.

The set of lams with the $\Subset$ relation is a pre-order with a bottom element, which is either 0 or $\lambda\overline{c}.\langle 0,0\rangle$ or $(\cdots, \lambda\overline{c_i}.\langle 0,0\rangle, \cdots)$ according to the domain we are considering. In Figure 13 we define a number of basic operations on the lam model that are used in the semantics of contracts.

The relevant property for the following theoretical development is the one below. We say that an operation is monotone if, whenever it is applied to arguments in the pre-order relation $\Subset$, it returns values in the same pre-order relation $\Subset$). The proof is straightforward and therefore omitted.

Proposition 4 *The operations of extension, parallel, sequence and plus are monotone with respect to* $\Subset$. *Additionally, if* $\mathcal{L} \Subset \mathcal{L}'$ *then* $\mathcal{L}[\overline{c'}/\overline{c}] \Subset \mathcal{L}'[\overline{c'}/\overline{c}]$.

5.2 *The finite approximants of abstract method behaviours.*

As explained above, the lam model of a `core ABS` program is obtained by means of a fixpoint technique plus a saturation applied to its contract class table. In particular, the lam model of the class table is a lam table that maps each method `C.m` of the program to $\lambda\overline{c}_{\mathtt{C.m}}.\langle \mathcal{L}_{\mathtt{C.m}}, \mathcal{L}'_{\mathtt{C.m}}\rangle$ where $\overline{c}_{\mathtt{C.m}} = \lceil \mathbb{r}, \overline{\mathbb{s}}\rceil$, with $\mathbb{r}(\overline{\mathbb{s}})$ being the header of the method contract of `C.m`. The definition of $\lceil \mathbb{r}, \overline{\mathbb{s}}\rceil$ is given in Figure 14 (we recall that names in headers occur linearly). The following definition presents the algorithm used to compute the next approximant of a method class table.

Definition 5 Let CCT be a contract class table of the form

$(\cdots, \mathtt{C.m} \mapsto\ \mathbb{r}_{\mathtt{C.m}}(\overline{\mathbb{s}_{\mathtt{C.m}}})\ \{\langle \mathbb{c}_{\mathtt{C.m}}, \mathbb{c}'_{\mathtt{C.m}}\rangle\}\ \mathbb{r}'_{\mathtt{C.m}}, \cdots)$

1. the *approximant 0* is defined as

   $(\cdots, \lambda(\lceil \mathbb{r}_{\mathtt{C.m}}, \overline{\mathbb{s}_{\mathtt{C.m}}}\rceil).\langle 0,0\rangle, \cdots)$;
2. let $\mathfrak{L} = (\cdots, \lambda\lceil \mathbb{r}_{\mathtt{C.m}}, \overline{\mathbb{s}_{\mathtt{C.m}}}\rceil.\langle \mathcal{L}_{\mathtt{C.m}}, \mathcal{L}'_{\mathtt{C.m}}\rangle, \cdots)$ be the n-th approximant; the $n+1$-th approximant is defined as $(\cdots, \lambda\lceil \mathbb{r}_{\mathtt{C.m}}, \overline{\mathbb{s}_{\mathtt{C.m}}}\rceil.\langle \mathcal{L}''_{\mathtt{C.m}}, \mathcal{L}'''_{\mathtt{C.m}}\rangle, \cdots)$ where $\langle \mathcal{L}''_{\mathtt{C.m}}, \mathcal{L}'''_{\mathtt{C.m}}\rangle = \mathbb{c}_{\mathtt{C.m}}(\mathfrak{L})_c \fatsemi \mathbb{c}'_{\mathtt{C.m}}(\mathfrak{L})_c$ with c being the cog of $\mathbb{r}_{\mathtt{C.m}}$ and the function $\mathbb{c}(\mathfrak{L})_c$ being defined by structural induction in Figure 16.

It is worth to notice that there are two rules for synchronous invocations in Figure 16: (L-SInvk) dealing with synchronous invocations on the same cog name of the caller – the index c of the transformation –, (L-RSInvk) dealing with synchronous invocations on different cog names.

Let

$$\left(\cdots, \lambda\overline{c_{\mathtt{C,m}}}.\langle {\mathcal{L}_{\mathtt{C,m}}}^0, {\mathcal{L}'_{\mathtt{C,m}}}^0\rangle, \cdots\right) = \left(\cdots, \lambda\overline{c_{\mathtt{C,m}}}.\langle 0,0\rangle\rangle, \cdots\right),$$

and let

$$\left(\cdots, \lambda\overline{c_{\mathtt{C,m}}}.\langle {\mathcal{L}_{\mathtt{C,m}}}^0, {\mathcal{L}'_{\mathtt{C,m}}}^0\rangle, \cdots\right),$$
$$\left(\cdots, \lambda\overline{c_{\mathtt{C,m}}}.\langle {\mathcal{L}_{\mathtt{C,m}}}^1, {\mathcal{L}'_{\mathtt{C,m}}}^1\rangle, \cdots\right),$$
$$\left(\cdots, \lambda\overline{c_{\mathtt{C,m}}}.\langle {\mathcal{L}_{\mathtt{C,m}}}^2, {\mathcal{L}'_{\mathtt{C,m}}}^2\rangle, \cdots\right), \cdots$$

be the sequence obtained by the algorithm of Definition 5 (this is the standard Knaster-Tarski technique). This sequence is non-decreasing (according to $\Subset$) because it is defined as a composition of monotone operators, see Proposition 5. Because of the creation of new cog names at each iteration, the fixpoint of the above sequence may not exist. We have already discussed the example of `Math.fact_nc`. In order to let our analysis terminate, after a given approximant, we run the Knaster-Tarski technique *using a different semantics for the operations* (L-SInvk), (L-RSInvk), (L-AInvk), *and* (L-GAinvk) (these are the rules where cog names may be created). In particular, when these operations are used at approximants larger than n, the renaming of free variables is disallowed. That is, the substitutions $[\overline{b'_{\mathtt{D,n}}}/\overline{b_{\mathtt{D,n}}}]$ in Figure 16 are removed. It

$$\lceil \_ \rceil \stackrel{def}{=} \varepsilon \qquad \lceil X \rceil \stackrel{def}{=} \varepsilon \qquad \lceil [cog{:}c, x_1{:}\mathbb{r}_1, \cdots, x_n{:}\mathbb{r}_n] \rceil \stackrel{def}{=} c \lceil \mathbb{r}_1 \rceil \cdots \lceil \mathbb{r}_n \rceil \qquad \lceil c \rightsquigarrow \mathbb{r} \rceil \stackrel{def}{=} c \lceil \mathbb{r} \rceil \qquad \lceil \overline{\mathbb{r}}, \overline{\mathbb{s}} \rceil \stackrel{def}{=} \lceil \overline{\mathbb{r}} \rceil \lceil \overline{\mathbb{s}} \rceil$$

Fig. 14 The extraction process.

$$(\_ \curvearrowright \_) \stackrel{def}{=} \varepsilon \qquad (\mathbb{r} \curvearrowright X) \stackrel{def}{=} \varepsilon \qquad ([cog{:}c', x_1{:}\mathbb{r}'_1, \cdots, x_n{:}\mathbb{r}'_n] \curvearrowright [cog{:}c, x_1{:}\mathbb{r}_1, \cdots, x_n{:}\mathbb{r}_n]) \stackrel{def}{=} c' \, (\mathbb{r}'_1 \curvearrowright \mathbb{r}_1) \cdots (\mathbb{r}'_n \curvearrowright \mathbb{r}_n)$$

$$(c' \rightsquigarrow \mathbb{r}' \curvearrowright c \rightsquigarrow \mathbb{r}) \stackrel{def}{=} c' \, (\mathbb{r}' \curvearrowright \mathbb{r})$$

Fig. 15 The cog mapping process.

1. let $\overline{b_{\mathtt{C,m}}} = (cog\_names(\mathcal{L}_{\mathtt{C,m}}) \cup cog\_names(\mathcal{L}'_{\mathtt{C,m}})) \setminus \overline{c_{\mathtt{C,m}}}$. These are the free cog names that are replaced by *fresh* cog names at every *transformation step*;
2. the transformation $\mathbb{c}_{\mathtt{C,m}}(\cdots \lambda\overline{c_{\mathtt{C,m}}}.\langle \mathcal{L}_{\mathtt{C,m}}, \mathcal{L}'_{\mathtt{C,m}} \rangle, \cdots)_c$ is defined inductively as follows:

– $\langle 0, 0 \rangle \& (c_1, c_2)^{[\mathtt{w}]}$ (L-GAzero)
 if $\mathbb{c}_{\mathtt{C,m}} = (c_1, c_2)^{[\mathtt{w}]}$;

– $(\lambda\overline{c_{\mathtt{D,n}}}.\langle \mathcal{L}_{\mathtt{D,n}}, \mathcal{L}'_{\mathtt{D,n}} \rangle \& (c, c)^{\mathtt{w}}[\overline{b'_{\mathtt{D,n}}}/\overline{b_{\mathtt{D,n}}}])(\mathbb{r}' \curvearrowright \mathbb{r}_{\mathtt{D,n}})(\overline{\mathbb{s}'} \curvearrowright \overline{\mathbb{s}_{\mathtt{D,n}}})$ (L-SInvk)
 if $\mathbb{c}_{\mathtt{C,m}} = \mathtt{D.n}\ \mathbb{r}'(\overline{\mathbb{s}'}) \to \mathbb{r}''$, $\mathbb{r}' = [cog{:}c, \overline{x}{:}\overline{\mathbb{r}'}]$, and $\textsc{cct}(\mathtt{D})(\mathtt{n}) = \mathbb{r}_{\mathtt{D,n}}(\overline{\mathbb{s}_{\mathtt{D,n}}})\ \{\langle \mathbb{c}_{\mathtt{D,n}}, \mathbb{c}'_{\mathtt{D,n}} \rangle\}\ \mathbb{r}'_{\mathtt{D,n}}$
 and $\overline{b'_{\mathtt{D,n}}}$ are fresh cog names;

– $(\lambda\overline{c_{\mathtt{D,n}}}.\langle \mathcal{L}_{\mathtt{D,n}}, \mathcal{L}'_{\mathtt{D,n}} \rangle \& (c, c')[\overline{b'_{\mathtt{D,n}}}/\overline{b_{\mathtt{D,n}}}])(\mathbb{r}' \curvearrowright \mathbb{r}_{\mathtt{D,n}})(\overline{\mathbb{s}'} \curvearrowright \overline{\mathbb{s}_{\mathtt{D,n}}})$ (L-RSInvk)
 if $\mathbb{c}_{\mathtt{C,m}} = \mathtt{D.n}\ \mathbb{r}'(\overline{\mathbb{s}'}) \to \mathbb{r}''$, $\mathbb{r}' = [cog{:}c', \overline{x}{:}\overline{\mathbb{r}'}]$, and $c \neq c'$ and $\textsc{cct}(\mathtt{D})(\mathtt{n}) = \mathbb{r}_{\mathtt{D,n}}(\overline{\mathbb{s}_{\mathtt{D,n}}})\ \{\langle \mathbb{c}_{\mathtt{D,n}}, \mathbb{c}'_{\mathtt{D,n}} \rangle\}\ \mathbb{r}'_{\mathtt{D,n}}$
 and $\overline{b'_{\mathtt{D,n}}}$ are fresh cog names;

– $\langle 0, \mathcal{L}''_{\mathtt{D,n}} \cup \mathcal{L}'''_{\mathtt{D,n}} \rangle$ (L-AInvk)
 if $\mathbb{c}_{\mathtt{C,m}} = \mathtt{D!n}\ \mathbb{r}'(\overline{\mathbb{s}'}) \to \mathbb{r}''$ and $\textsc{cct}(\mathtt{D})(\mathtt{n}) = \mathbb{r}_{\mathtt{D,n}}(\overline{\mathbb{s}_{\mathtt{D,n}}})\ \{\langle \mathbb{c}_{\mathtt{D,n}}, \mathbb{c}'_{\mathtt{D,n}} \rangle\}\ \mathbb{r}'_{\mathtt{D,n}}$
 and $\langle \mathcal{L}''_{\mathtt{D,n}}, \mathcal{L}'''_{\mathtt{D,n}} \rangle = (\lambda\overline{c_{\mathtt{D,n}}}.\langle \mathcal{L}_{\mathtt{D,n}}, \mathcal{L}'_{\mathtt{D,n}} \rangle[\overline{b'_{\mathtt{D,n}}}/\overline{b_{\mathtt{D,n}}}])(\mathbb{r}' \curvearrowright \mathbb{r}_{\mathtt{D,n}})(\overline{\mathbb{s}'} \curvearrowright \overline{\mathbb{s}_{\mathtt{D,n}}})$ and $\overline{b'_{\mathtt{D,n}}}$ are fresh cog names;

– $(\lambda\overline{c_{\mathtt{D,n}}}.\langle \mathcal{L}_{\mathtt{D,n}}, \mathcal{L}'_{\mathtt{D,n}} \rangle \& (c_1, c_2)^{[\mathtt{w}]}[\overline{b'_{\mathtt{D,n}}}/\overline{b_{\mathtt{D,n}}}])(\mathbb{r}' \curvearrowright \mathbb{r}_{\mathtt{D,n}})(\overline{\mathbb{s}'} \curvearrowright \overline{\mathbb{s}_{\mathtt{D,n}}})$ (L-GAInvk)
 if $\mathbb{c}_{\mathtt{C,m}} = \mathtt{D!n}\ \mathbb{r}'(\overline{\mathbb{s}'}) \to \mathbb{r}''.(c_1, c_2)^{[\mathtt{w}]}$ and $\textsc{cct}(\mathtt{D})(\mathtt{n}) = \mathbb{r}_{\mathtt{D,n}}(\overline{\mathbb{s}_{\mathtt{D,n}}})\ \{\langle \mathbb{c}_{\mathtt{D,n}}, \mathbb{c}'_{\mathtt{D,n}} \rangle\}\ \mathbb{r}'_{\mathtt{D,n}}$ and $\overline{b'_{\mathtt{D,n}}}$ are fresh cog names;

– $\mathbb{c}'_{\mathtt{C,m}}(\cdots, \lambda\overline{c_{\mathtt{C,m}}}.\langle \mathcal{L}_{\mathtt{C,m}}, \mathcal{L}'_{\mathtt{C,m}} \rangle, \cdots)_c \fatsemi \mathbb{c}''_{\mathtt{C,m}}(\cdots, \lambda\overline{c_{\mathtt{C,m}}}.\langle \mathcal{L}_{\mathtt{C,m}}, \mathcal{L}'_{\mathtt{C,m}} \rangle, \cdots)_c$ (L-Seq)
 if $\mathbb{c}_{\mathtt{C,m}} = \mathbb{c}'_{\mathtt{C,m}} \fatsemi \mathbb{c}''_{\mathtt{C,m}}$;

– $\mathbb{c}'_{\mathtt{C,m}}(\cdots, \lambda\overline{c_{\mathtt{C,m}}}.\langle \mathcal{L}_{\mathtt{C,m}}, \mathcal{L}'_{\mathtt{C,m}} \rangle, \cdots)_c + \mathbb{c}''_{\mathtt{C,m}}(\cdots, \lambda\overline{c_{\mathtt{C,m}}}.\langle \mathcal{L}_{\mathtt{C,m}}, \mathcal{L}'_{\mathtt{C,m}} \rangle, \cdots)_c$ (L-Plus)
 if $\mathbb{c}_{\mathtt{C,m}} = \mathbb{c}'_{\mathtt{C,m}} + \mathbb{c}''_{\mathtt{C,m}}$;

– $\mathbb{c}'_{\mathtt{C,m}}(\cdots, \lambda\overline{c_{\mathtt{C,m}}}.\langle \mathcal{L}_{\mathtt{C,m}}, \mathcal{L}'_{\mathtt{C,m}} \rangle, \cdots)_c \parallel \mathbb{c}''_{\mathtt{C,m}}(\cdots, \lambda\overline{c_{\mathtt{C,m}}}.\langle \mathcal{L}_{\mathtt{C,m}}, \mathcal{L}'_{\mathtt{C,m}} \rangle, \cdots)_c$ (L-Par)
 if $\mathbb{c}_{\mathtt{C,m}} = \mathbb{c}'_{\mathtt{C,m}} \parallel \mathbb{c}''_{\mathtt{C,m}}$.

Fig. 16 Lam transformation of CCT.

is straightforward to verify that these operations are still monotone. It is also straightforward to demonstrate by a simple cardinality argument the existence of fixpoints in the lam domain by running the Knaster-Tarski technique with this different semantics. This method is called a *saturation technique at n*.

For example, if we compute the third approximant of

$$\begin{array}{l} \mathtt{Math.fact\_nc} \mapsto \\ \quad [cog{:}c](\_)\{\langle 0 + \mathtt{Math!fact\_nc}\ [cog{:}c'](\_) \to \_.(c, c'), 0 \rangle \\ \qquad\qquad \}\_ \end{array}$$

we get the sequence

$$\begin{array}{l} \lambda c.\langle 0, 0 \rangle \\ \lambda c.\langle\, [(c, c_0)]\, , 0 \rangle \\ \lambda c.\langle\, [(c, c_0), (c_0, c_1)]\, , 0 \rangle \\ \lambda c.\langle\, [(c, c_0), (c_0, c_1), (c_1, c_2)]\, , 0 \rangle \end{array}$$

and, if we saturate a this point, we obtain

$$\begin{array}{ll} \lambda c.\langle\, [(c, c_0)(c_0, c_0), (c_0, c_1), (c_1, c_2)]\, , 0 \rangle & \\ \lambda c.\langle\, [(c, c_0)(c_0, c_0), (c_0, c_1), (c_1, c_2)]\, , 0 \rangle & \text{fixpoint} \end{array}$$

Definition 6 Let $(\textsc{ct}, \{\overline{T\ x\ ;\ s}\}, \textsc{cct})$ be a `core ABS` program and let $\Big(\cdots \lambda\overline{c_{\mathtt{C,m}}}.\langle {\mathcal{L}_{\mathtt{C,m}}}^{n+h}, {\mathcal{L}'_{\mathtt{C,m}}}^{n+h} \rangle, \cdots\Big)$ be the fixpoint (unique up to renaming of cog names) obtained by the saturation technique at n. The *abstract class table at* n, written $\mathrm{ACT}_{[n]}$, is a map that takes `C.m` and returns $\lambda\overline{c_{\mathtt{C,m}}}.\langle \mathcal{L}_{\mathtt{C,m}}{}^{n+h}, \mathcal{L}'_{\mathtt{C,m}}{}^{n+h}\rangle$.

Let $(\mathrm{CT}, \{\overline{T\ x\ ;}\ s\}, \mathrm{CCT})$ be a `core ABS` program and

$$\Gamma \vdash_{\text{start}} \{\overline{T\ x\ ;}\ s\} : \langle \mathbb{c}, \mathbb{c}' \rangle \ \triangleright \mathcal{U} \,|\, \Gamma \ .$$

The *abstract semantics saturated at* n of $(\mathrm{CT}, \{\overline{T\ x\ ;}\ s\}, \mathrm{CCT})$ is $(\mathbb{c}(\mathrm{ACT}_{[n]})_{\text{start}}) \fatsemi (\mathbb{c}'(\mathrm{ACT}_{[n]})_{\text{start}})$.

As an example, in Figure 17 we compute the abstract semantics saturated at 2 of the class `Math` in Figure 5.

5.3 *Deadlock analysis of lams.*

Definition 7 Let L be a relation on cog names (pairs in L are either of the form (c_1, c_2) or of the form $(c_1, c_2)^{\mathtt{w}}$. L *contains a circularity* if $L^{\mathtt{get}}$ has a pair (c, c) (see Definition 2). Similarly, $\langle \mathcal{L}, \mathcal{L}' \rangle$ (or $\lambda\overline{c}.\langle \mathcal{L}, \mathcal{L}' \rangle$) has a circularity if there is $L \in \mathcal{L} \cup \mathcal{L}'$ that contains a circularity.

A `core ABS` program with an abstract class table saturated at n is *deadlock-free* if its abstract semantics $\langle \mathcal{L}, \mathcal{L}' \rangle$ does not contain a circularity.

The fixpoints for `Math.fact_g` and `Math.fact_ag` are found at the third iteration. According to the above definition of deadlock-freedom, `Math.fact_g` yields a deadlock, whilst `Math.fact_ag` is deadlock-free because $\{(c, c)^{\mathtt{w}}\}^{\mathtt{get}}$ does not contain any circularity. As discussed before, there exists no fixpoint for `Math.fact_nc`. If we decide to stop at the approximant 2 and saturate, we get

$$\lambda c.\langle\, [(c, c'), (c', c'), (c', c'')]\, , 0\rangle,$$

which contains a circularity that is a false positive.

Note that saturation might even start at the approximant 0 (where every method is $\lambda c.\langle 0, 0\rangle$). In this case, for `Math.fact_g` and `Math.fact_ag`, we get the same answer and the same pair of lams as the above third approximant. For `Math.fact_nc` we get

$$\lambda c.\langle\, [(c, c'), (c', c')]\, , 0\rangle,$$

which contains a circularity.

In general, in techniques like the one we have presented, it is possible to augment the precision of the analysis by delaying the saturation. However, assuming that pairwise different method contracts have disjoint free cog names (which is a reasonable assumption), we have not found any sample `core ABS` code where saturating at 1 gives a better precision than saturating at 0. While this issue is left open, the current version of our tool `DF4ABS` allows one to specify the saturation point; the default saturation point is 0.

The computation of the abstract class table for class `CpxSched` does not need any saturation, all methods are non-recursive and encounter their fixpoint by iteration 2. (See Figure 18.) The abstract class table shows a circularity for method `m1`, manifesting the presence of a deadlock.

The correctness of the fixpoint analysis of contracts discussed in this section is demonstrated in Appendix B. We remark that this technique is as modular as the inference system: once the contracts of a module have been computed, one may run the fixpoint analysis and attach the corresponding abstract values to the code. Analysing a program reduces to computing the lam of the main function.

6 The model-checking analysis of contracts

The second analysis technique for the contracts of Section 4 consists of computing contract models *by expanding* their invocations. We therefore begin this section by introducing a semantics of contracts that is alternative to the one of Section 5.

6.1 *Operational semantics of contracts.*

The operational semantics of a contract is defined as a reduction relation between terms that are *contract pairs* $\mathbb{cp}$, whose syntax is defined in Figure 19. These contract pairs highlight (in the operational semantics) the fact that every contract actually represents two collections of relations on cog names: those corresponding to the *present states* and those corresponding to *future states*. We have discussed this dichotomy in Section 4.

In Figure 19 we have also defined the *contract pair contexts*, noted $\mathfrak{C}[\ ]_c$, which are indexed contract pairs with a hole. The index c indicates that the hole is immediately enclosed by $\langle \cdot, \cdot \rangle_c$.

The reduction relation that defines the evaluation of contract pairs $\langle \mathbb{cp}_1, \mathbb{cp}'_1 \rangle_c \longrightarrow \langle \mathbb{cp}_2, \mathbb{cp}'_2 \rangle_c$ is defined in Figure 19. There are four reduction rules: (Red-SInvk) for synchronous invocation on the same cog name of the caller (which is stored in the index of the enclosing pair), (Red-RSInvk) for synchronous invocations on different cog name, (Red-AInvk) for asynchronous invocations, and (Red-GAInvk) for asynchronous invocations with synchronisations. We observe that every evaluation step amounts to expanding method invocations by replacing free cog names in method contracts with fresh names and *without modifying the syntax tree of contract pairs.*

method	approx. 0	approx. 1	approx.2	saturation
`Math.fact_g`	$\lambda c.\langle 0, 0\rangle$	$\lambda c.\langle \,[\!(c,c)]\!\,, 0\rangle$	$\lambda c.\langle \,[\!(c,c)]\!\,, 0\rangle$	
`Math.fact_ag`	$\lambda c.\langle 0, 0\rangle$	$\lambda c.\langle \,[\!(c,c)^{\mathtt{w}}]\!\,, 0\rangle$	$\lambda c.\langle \,[\!(c,c)^{\mathtt{w}}]\!\,, 0\rangle$	
`Math.fact_nc`	$\lambda c.\langle 0, 0\rangle$	$\lambda c.\langle \,[\!(c,c')]\!\,, 0\rangle$	$\lambda c.\langle \,[\!(c,c'),(c',c'')]\!\,, 0\rangle$	$\lambda c.\langle \,[\!(c,c'),(c',c'),(c',c'')]\!\,, 0\rangle$

Fig. 17 Abstract class table computation for class `Math`.

method	approx. 0	approx. 1	approx.2
`CpxSched.m1`	$\lambda c,c',c''.\langle 0, 0\rangle$	$\lambda c,c',c''.\langle 0, \,[\!(c',c''),(c'',c')]\!\,\rangle$	$\lambda c,c',c''.\langle 0, \,[\!(c',c''),(c'',c')]\!\,\rangle$
`CpxSched.m2`	$\lambda c,c'.\langle 0, 0\rangle$	$\lambda c,c'.\langle \,[\!(c,c')]\!\,, 0\rangle$	$\lambda c,c'.\langle \,[\!(c,c')]\!\,, 0\rangle$
`CpxSched.m3`	$\lambda c.\langle 0, 0\rangle$	$\lambda c.\langle 0, 0\rangle$	

Fig. 18 Abstract class table computation for class `CpxSched`.

contract pairs

$$\mathbb{cp} \;::=\; \mathbb{c} \;\mid\; \langle \mathbb{cp}, \mathbb{cp}\rangle_c \;\mid\; \mathbb{cp}\&(c,c')^{[\mathtt{w}]} \;\mid\; \mathbb{cp}+\mathbb{cp} \;\mid\; \mathbb{cp}\,\fatsemi\,\mathbb{cp} \;\mid\; \mathbb{cp}\parallel\mathbb{cp}$$

contract pairs contexts ($\sharp \in \{+, \fatsemi, \parallel\}$)

$$\mathfrak{D}[\,] \;::=\; [\,] \;\mid\; \mathfrak{D}[\,]\&(c',c'')^{[\mathtt{w}]} \;\mid\; \mathfrak{D}[\,]\sharp\mathbb{cp} \;\mid\; \mathbb{cp}\sharp\mathfrak{D}[\,]$$

$$\mathfrak{C}[\,]_c \;::=\; \langle \mathfrak{D}[\,], \mathbb{cp}\rangle_c \;\mid\; \langle \mathbb{cp}, \mathfrak{D}[\,]\rangle_c \;\mid\; \langle \mathfrak{C}[\,]_c, \mathbb{cp}\rangle_{c'} \;\mid\; \langle \mathbb{cp}, \mathfrak{C}[\,]_c\rangle_{c'} \;\mid\; \mathfrak{C}[\,]_c\&(c',c'')^{[\mathtt{w}]} \;\mid\; \mathfrak{C}[\,]_c\sharp\mathbb{cp} \;\mid\; \mathbb{cp}\sharp\mathfrak{C}[\,]_c$$

reduction relation

(Red-SInvk)

$$\frac{\begin{array}{c}\mathtt{C.m} = \mathbb{s}(\overline{\mathbb{s}})\{\langle \mathbb{c}, \mathbb{c}'\rangle\}\mathbb{s}' \qquad \mathbb{r} = [cog{:}c, \overline{x}{:}\overline{\mathbb{r}''}]\\ cog\_names(\langle \mathbb{c}, \mathbb{c}'\rangle) \setminus cog\_names(\mathbb{s}, \overline{\mathbb{s}}) = \widetilde{z}\\ \widetilde{w} \text{ are fresh} \qquad \langle \mathbb{c}, \mathbb{c}'\rangle[^{\widetilde{w}}/_{\widetilde{z}}][^{\mathbb{r},\overline{\mathbb{r}}}/_{\mathbb{s},\overline{\mathbb{s}}}] = \langle \mathbb{c}'', \mathbb{c}'''\rangle\end{array}}{\mathfrak{C}[\mathtt{C.m}\ \mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{r}']_c \longrightarrow \mathfrak{C}[\langle \mathbb{c}'', \mathbb{c}'''\rangle_c]_c}$$

(Red-RSInvk)

$$\frac{\begin{array}{c}\mathtt{C.m} = \mathbb{s}(\overline{\mathbb{s}})\{\langle \mathbb{c}, \mathbb{c}'\rangle\}\mathbb{s}' \qquad \mathbb{r} = [cog{:}c', \overline{x}{:}\overline{\mathbb{r}''}] \qquad c \neq c'\\ cog\_names(\langle \mathbb{c}, \mathbb{c}'\rangle) \setminus cog\_names(\mathbb{s}, \overline{\mathbb{s}}) = \widetilde{z}\\ \widetilde{w} \text{ are fresh} \qquad \langle \mathbb{c}, \mathbb{c}'\rangle[^{\widetilde{w}}/_{\widetilde{z}}][^{\mathbb{r},\overline{\mathbb{r}}}/_{\mathbb{s},\overline{\mathbb{s}}}] = \langle \mathbb{c}'', \mathbb{c}'''\rangle\end{array}}{\mathfrak{C}[\mathtt{C.m}\ \mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{r}']_c \longrightarrow \mathfrak{C}[\langle \mathbb{c}'', \mathbb{c}'''\rangle_{c'}\&(c,c')]_c}$$

(Red-AInvk)

$$\frac{\begin{array}{c}\mathtt{C.m} = \mathbb{s}(\overline{\mathbb{s}})\{\langle \mathbb{c}, \mathbb{c}'\rangle\}\mathbb{s}' \qquad \mathbb{r} = [cog{:}c', \overline{x}{:}\overline{\mathbb{r}''}]\\ cog\_names(\langle \mathbb{c}, \mathbb{c}'\rangle) \setminus cog\_names(\mathbb{s}, \overline{\mathbb{s}}) = \widetilde{z}\\ \widetilde{w} \text{ are fresh} \qquad \langle \mathbb{c}, \mathbb{c}'\rangle[^{\widetilde{w}}/_{\widetilde{z}}][^{\mathbb{r},\overline{\mathbb{r}}}/_{\mathbb{s},\overline{\mathbb{s}}}] = \langle \mathbb{c}'', \mathbb{c}'''\rangle\end{array}}{\mathfrak{C}[\mathtt{C!m}\ \mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{r}']_c \longrightarrow \mathfrak{C}[\langle \mathbb{c}'', \mathbb{c}'''\rangle_{c'}]_c}$$

(Red-GAInvk)

$$\frac{\begin{array}{c}\mathtt{C.m} = \mathbb{s}(\overline{\mathbb{s}})\{\langle \mathbb{c}, \mathbb{c}'\rangle\}\mathbb{s}' \qquad \mathbb{r} = [cog{:}c', \overline{x}{:}\overline{\mathbb{r}''}]\\ cog\_names(\langle \mathbb{c}, \mathbb{c}'\rangle) \setminus cog\_names(\mathbb{s}, \overline{\mathbb{s}}) = \widetilde{z}\\ \widetilde{w} \text{ are fresh} \qquad \langle \mathbb{c}, \mathbb{c}'\rangle[^{\widetilde{w}}/_{\widetilde{z}}][^{\mathbb{r},\overline{\mathbb{r}}}/_{\mathbb{s},\overline{\mathbb{s}}}] = \langle \mathbb{c}'', \mathbb{c}'''\rangle\end{array}}{\mathfrak{C}[\mathtt{C!m}\ \mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{r}'.(c'',c''')^{[\mathtt{w}]}]_c \longrightarrow \mathfrak{C}[\langle \mathbb{c}'', \mathbb{c}'''\rangle_{c'}\&(c'',c''')^{[\mathtt{w}]}]_c}$$

Fig. 19 Contract reduction rules.

To illustrate the operational semantics of contracts we discuss three examples:

1. Let
   $\mathtt{F.f} \;=\; [cog : c](x : [cog : c'], y : [cog : c'']) \{$
   $\langle(\mathtt{F.g}\,[cog : c'](x : [cog : c'']) \to \_).(c,c') \;+\; 0.(c',c''), 0\rangle$
   $\}_\_$
   and
   $\mathtt{F.g} \;=\; [cog : c](x : [cog : c'])\{\langle 0.(c,c') + 0.(c',c), 0\rangle\}_\_$
   Then

$$\begin{array}{l}\langle \mathtt{F!f}\,[cog : c](x : [cog : c'], y : [cog : c'']) \to \_, 0\rangle_{\mathrm{start}}\\ \longrightarrow\; \langle\langle 0, (\mathtt{F.g}\,[cog : c'](x : [cog : c'']) \to \_).(c,c') \;+\; 0.(c',c'')\rangle_c, 0\rangle_{\mathrm{start}}\\ \longrightarrow\; \langle\langle 0, \langle 0.(c',c'') + 0.(c'',c'), 0\rangle_{c'}\&(c,c') \;+\; 0.(c',c'')\rangle_c, 0\rangle_{\mathrm{start}}\end{array}$$

   The contract pair in the final state *does not contain method invocations*. This is because the above main function is not recursive. Additionally, the evaluation of $\mathtt{F.f}\,[cog : c](x : [cog : c'], y : [cog : c''])$ *has not created names*. This is because names in the bodies of `F.f` and `F.g` are bound.
2. Let

   $\mathtt{F.h} \;=\; [cog : c](\_)\{\langle 0, (\mathtt{F.h}[cog : c'](\_) \to \_)\|0.(c,c')\rangle\}_\_$

   Then

$$\begin{array}{l}\langle \mathtt{F!h}\,[cog : c](\_) \to \_, 0\rangle_{\mathrm{start}}\\ \longrightarrow\; \langle\langle 0, (\mathtt{F.h}[cog : c'](\_) \to \_)\|0.(c,c')\rangle_c, 0\rangle_{\mathrm{start}}\\ \longrightarrow\; \langle\langle 0, \langle 0, (\mathtt{F.h}[cog : c''](\_) \to \_)\|0.(c',c'')\rangle_{c'}\|0.(c,c')\rangle_c, 0\rangle_{\mathrm{start}}\\ \longrightarrow\; \cdots\end{array}$$

   where, in this case, the contract pairs grow in the number of dependencies as the evaluation progresses. This growth *is due to the presence of a free name* in the definition of `F.h` that, as said, corresponds to generating a fresh name at every recursive invocation.

3. Let

$$\texttt{F.l} = [cog : c](\,)\{\langle 0.(c,c') \fatsemi (0.(c,c') \parallel \texttt{F!l}\,[cog : c](\,) \rightarrow \_), 0\rangle\}_\_$$

Then

$$\begin{array}{l}
\langle \texttt{F!l}\,[cog : c](\,) \rightarrow \_, 0\rangle_{\mathrm{start}} \\
\longrightarrow\ \langle\langle 0, 0.(c,c') \fatsemi (0.(c,c') \parallel \texttt{F!l}\,[cog : c](\,) \rightarrow \_)\rangle_c, 0\rangle_{\mathrm{start}} \\
\longrightarrow\ \langle\langle 0, 0.(c,c') \fatsemi (0.(c,c') \parallel \\
\qquad \langle 0, 0.(c',c'') \fatsemi (0.(c',c'') \parallel \texttt{F!l}\,[cog : c''](\,) \rightarrow \_)\rangle_{c'})\rangle_c, 0\rangle_{\mathrm{start}} \\
\longrightarrow \cdots
\end{array}$$

In this case, the contract pairs grow in the number of "$\fatsemi$"-terms, which become larger and larger as the evaluation progresses.

It is clear that, in presence of recursion and of free cog names in method contracts, a technique that analyses contracts by expanding method invocations is fated to fail because the system is infinite state. However, it is possible to stop the expansions at suitable points without losing any relevant information about dependencies. In this section we highlight the technique we have developed in [19] that has been prototyped for `core ABS` in `DF4ABS`.

6.2 *Linear recursive contract class tables.*

Since contract pairs models may be infinite-state, instead of resorting to a saturation technique, which introduces inaccuracies, we exploit a generalisation of permutation theory that let us decide when stopping the evaluation with the guarantee that if no circular dependency has been found up to that moment then it will not appear afterwards. That stage corresponds to the *order* of an associated permutation. It turns out that this technique is suited for so-called *linear recursive* contract class tables.

Definition 8 A contract class table is *linear recursive* if (mutual) recursive invocations in bodies of methods have *at most one recursive invocation.*

It is worth to observe that a `core ABS` program may be linear recursive while the corresponding contract class table is not. For example, consider the following method `foo` of class `Foo` that prints integers by invoking a printer service and awaits for the termination of the printer task and for its own termination:

```
Void foo(Int n, Print x){
   Fut<Void> u, v ;
   if (n == 0) return() ;
   else { u = this!foo(n-1, x) ;
          v = x!print(n) ;
          await v? ;
          await u? ;
          return() ;
     }
}
```

While `foo` has only one recursive invocation, its contract written in Figure 20 is not. That is, the contract of `Foo.foo` displays *two* recursive invocations because, in correspondence of the `await v?` instruction, we need to collect all the effects produced by the previous unsynchronised asynchronous invocations (see rule (T-Await)) [3].

6.3 *Mutations and flashbacks*

The idea of our technique is to consider the patterns of cog names in the formal parameters and the (at most unique) recursive invocation of method contracts and to study the changes. For example, the above method contracts of `F.h` and `F.l` transform the pattern of cog names in the formal parameters, written (c) into the pattern of recursive invocation (c'). We write this transformation as

$$(c) \ \rightsquigarrow\ (c')\ .$$

In general, the transformations we consider are called *mutations.*

Definition 9 A *mutation* is a transformation of tuples of (cog) names, written

$$(x_1, \cdots, x_n) \ \rightsquigarrow\ (x'_1, \cdots, x'_n)$$

where $x_1, \cdots, x_n$ are pairwise different and x'_i *may not occur in* $\{x_1, \cdots, x_n\}$.

Applying a mutation $(x_1, \cdots, x_n) \rightsquigarrow (x'_1, \cdots, x'_n)$ to a tuple of cog names (that may contain duplications) $(c_1, \cdots, c_n)$ gives a tuple $(c'_1, \cdots, c'_n)$ where

- $c'_i = c_j$ if $x'_i = x_j$;
- c'_i is a fresh name if $x'_i \notin \{x_1, \cdots, x_n\}$;
- $c'_i = c'_j$ if they are both fresh and $x'_i = x'_j$.

We write $(c_1, \cdots, c_n) \rightarrow_{\texttt{mut}} (c'_1, \cdots, c'_n)$ when $(c'_1, \cdots, c'_n)$ is obtained by applying a mutation (which is kept implicit) to $(c_1, \cdots, c_n)$.

For example, given the mutation

$$(x, y, z, u) \rightsquigarrow (y, x, z', z') \qquad (1)$$

we obtain the following sequence of tuples:

$$\begin{array}{ll}
(c, c', c'', c''') & \rightarrow_{\texttt{mut}} (c', c, c_1, c_1) \\
& \rightarrow_{\texttt{mut}} (c, c', c_2, c_2) \\
& \rightarrow_{\texttt{mut}} (c', c, c_3, c_3) \\
& \rightarrow_{\texttt{mut}} \quad \cdots
\end{array} \qquad (2)$$

[3] It is possible to define sufficient conditions on `core ABS` programs that entail linear recursive contract class tables. For example, two such conditions are that, in (mutual) recursive methods, recursive invocations are either (i) synchronous or (ii) asynchronous followed by a `get` or `await` synchronisation on the future value, without any other `get` or `await` synchronisation or synchronous invocation in between.

$$\begin{aligned} &\texttt{Foo.foo} = [cog : c](\_, x : [cog : c'])\{ \ \langle 0 \ + \ \Big((\mathbb{c}.(c,c')^{\mathtt{w}} \parallel \mathbb{c}') \mathbin{⨾} \mathbb{c}'.(c,c)^{\mathtt{w}} \Big), 0 \rangle \ \} \to \_ \\ &\text{where } \ \mathbb{c} = \texttt{Print!print}\, [cog : c'](\_) \to \_ \\ &\qquad\quad \mathbb{c}' = \texttt{Foo!foo}\, [cog : c](\_, x : [cog : c']) \to \_ \end{aligned}$$

Fig. 20 Method contract of `Foo.foo`.

When a mutation $(x_1, \cdots, x_n) \rightsquigarrow (x'_1, \cdots, x'_n)$ is such that $\{x_1, \cdots, x_n\} = \{x'_1, \cdots, x'_n\}$ then the mutation is a *permutation* [8]. In this case, the permutation theory guarantees that, by repeatedly applying the same permutation to a tuple of names, at some point, one obtains the initial tuple. This point, which is known as the *order of the permutation*, allows one to define the following algorithm for linear recursive method contracts whose mutation is a permutation:

1. compute the order of the permutation associated to the recursive method contract and
2. correspondingly unfold the term to evaluate.

It is clear that, when method contract bodies have no free cog names, further unfoldings of the recursive method contract cannot add new dependencies. Therefore the evaluation, as far as dependencies are concerned, may stop.

When a mutation is not a permutation, as in the example above, it is not possible to get again an old tuple by applying the mutation because of the presence of fresh names. However, it is possible to demonstrate that a tuple is equal to an old one *up-to* a suitable map, called flashback.

Definition 10 A tuple of cog names $(c_1, \cdots, c_n)$ is equivalent to $(c'_1, \cdots, c'_n)$, written $(c_1, \cdots, c_n) \approx (c'_1, \cdots, c'_n)$, if there is an injection $\imath$ called *flashback* such that:

1. $(c_1, \cdots, c_n) = (\imath(c'_1), \cdots, \imath(c'_n))$
2. $\imath$ is the identity on "old names", that is, if $c'_i \in \{c_1, \cdots, c_n\}$ then $\imath(c'_i) = c'_i$.

For example, in the sequence of transitions (2), there is a flashback from the last tuple to the second one and there is (and there will be, by applying the mutation (1)) no tuple that is equivalent to the initial tuple.

It is possible to generalize the result about permutation orders:

Theorem 1 ([19]) *Let* $(x_1, \cdots, x_n) \rightsquigarrow (x'_1, \cdots, x'_n)$ *be a mutation and let*

$$(c_1, \cdots, c_n) \to_{\mathtt{mut}} (c_{n+1}, \cdots, c_{2n}) \to_{\mathtt{mut}} (c_{2n+1}, \cdots, c_{3n}) \to_{\mathtt{mut}} \cdots$$

be a sequence of applications of the mutation. Then there are $0 \le h < k$ *such that*

$$(c_{hn+1}, \cdots, c_{(h+1)n}) \approx (c_{kn+1}, \cdots, c_{(k+1)n})$$

The value k *is called* order of the mutation*.*

For example, the order of the mutation (1) is 3.

6.4 Evaluation of the main contract pair

The generalisation of permutation theory in Theorem 1 allows us to define the notion of *order of the contract of the main function* in a linear recursive contract class table. This order is the length of the evaluation of the contract obtained

1. by unfolding every recursive function as many times as *twice its ordering* [4];
2. by iteratively applying 1 to every invocation of recursive function that has been produced during the unfolding.

In order to state the theorem of the correctness of our analysis technique, we need to define the lam of a contract pair. The following functions will do the job.

Let $[\![\cdot]\!]$ be a map taking a contract pair and returning a pair of lams that is defined by

$$\begin{aligned} [\![\texttt{C.m}\ \mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{r}']\!] &= \langle 0, 0\rangle \\ [\![\texttt{C!m}\ \mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{r}']\!] &= \langle 0, 0\rangle \\ [\![\texttt{C!m}\ \mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{r}'.(c,c')^{[\mathtt{w}]}]\!] &= \langle\, [\,(c,c')^{[\mathtt{w}]}\,]\,, 0\rangle \\ [\![\langle \mathbb{cp}, \mathbb{cp}'\rangle_c]\!] &= \langle [\![\mathbb{cp}]\!], [\![\mathbb{cp}']\!]\rangle \end{aligned}$$

and it is homomorphic with respect to the operations $+$, $⨾$, $\parallel$ (whose definition on pairs of lams is in Figure 13). Let $\mathbb{t}$ be terms of the following syntax

$$\mathbb{t} \ ::= \quad \mathcal{L} \quad | \quad \langle \mathbb{t}, \mathbb{t}\rangle$$

and let $(\mathcal{L})^\flat = \mathcal{L}$ and $(\langle \mathbb{t}, \mathbb{t}'\rangle)^\flat = (\mathbb{t})^\flat, (\mathbb{t}')^\flat$.

Theorem 2 ([19]) *Let* $\langle \mathbb{cp}_1, \mathbb{cp}'_1\rangle$ *be a main function contract and let*

$$\langle \mathbb{cp}_1, \mathbb{cp}'_1\rangle_{\mathtt{start}} \longrightarrow \langle \mathbb{cp}_2, \mathbb{cp}'_2\rangle_{\mathtt{start}} \longrightarrow \langle \mathbb{cp}_3, \mathbb{cp}'_3\rangle_{\mathtt{start}} \longrightarrow \cdots$$

be its evaluation. Then there is a k*, which is the order of* $\langle \mathbb{cp}_1, \mathbb{cp}'_1\rangle_{\mathtt{start}}$ *such that if a circularity occurs in* $([\![\langle \mathbb{cp}_{k+h}, \mathbb{cp}'_{k+h}\rangle_{\mathtt{start}}]\!])^\flat$*, for every* h*, then it also occurs in* $([\![\langle \mathbb{cp}_k, \mathbb{cp}'_k\rangle_{\mathtt{start}}]\!])^\flat$*.*

Example 7 The reduction of the contract of method `Math.fact_nc` is as in Figure 21. The theory of mutations provide us with an order for this evaluation. In particular, the mutation associated to `Math.fact_nc` is

[4] The interested reader may find in [19] the technical reason for unfolding recursive methods as many times as twice the length of the order of the corresponding mutation.

$c \rightsquigarrow c'$, with order 1, such that after one step we can encounter a flashback to a previous state of the mutation. Therefore, we need to reduce our contract for a number of steps corresponding to twice the ordering of `Math.fact_nc`: after two steps we find the flashback associating the last generated pair (c', c'') to the one produced in the previous step (c, c'), by mapping c' to c and c'' to c'.

The flattening and the evaluation of the resulting contract are shown in Figure 22 and produce the pair of lams $\langle\, [\![(c', c''), (c, c')]\!]\, , 0\rangle$ which does not present any deadlock. Thus, differently from the fixpoint analysis for the same example, with this operational analysis we get a precise answer instead of a false positive. (See Figure 17 and Section 5.3.)

The correctness of the technique based on mutations is demonstrated in Appendix C.

7 The DF4ABS tool and its application to the case study

`core ABS` (actually full `ABS` [23]) comes with a suite [39] that offers a compilation framework, a set of tools to analyse the code, an Eclipse IDE plugin and Emacs mode for the language. We extended this suite with an implementation of our deadlock analysis framework (at the time of writing the suite has only the fixpoint analyser, the full framework is available at `http://df4abs.nws.cs.unibo.it`). The `DF4ABS` tool is built upon the abstract syntax tree (AST) of the `core ABS` type checker, which allows us to exploit the type information stored in every node of the tree. This simplifies the implementation of several contract inference rules. The are four main modules that comprise `DF4ABS`:

(1) *Contract and Constraint Generation.* This is performed in three steps: (i) the tool first parses the classes of the program and generates a map between interfaces and classes, required for the contract inference of method calls; (ii) then it parses again all classes of the program to generate the initial environment Γ that maps methods to the corresponding method signatures; and (iii) it finally parses the AST and, at each node, it applies the contract inference rules in Figures 9, 10, and 11.

(2) *Constraint Solving* is done by a generic semi-unification solver implemented in Java, following the algorithm defined in [21]. When the solver terminates (and no error is found), it produces a substitution that satisfies the input constraints. Applying this substitution to the generated contracts produces the abstract class table and the contract of the main function of the program.

(3) *Fixpoint Analysis* uses dynamic structures to store lams of every method contract (because lams become larger and larger as the analysis progresses). At each iteration of the analysis, a number of fresh cog names is created and the states are updated according to what is prescribed by the contract. At each iteration, the tool checks whether a fixpoint has been reached. Saturation starts when the number of iterations reaches a maximum value (that may be customised by the user). In this case, since the precision of the algorithm degrades, the tool signals that the answer may be imprecise. To detect whether a relation in the fixpoint lam contains a circular dependency, we run Tarjan algorithm [35] for connected components of graphs and we stop the algorithm when a circularity is found.

(4) *Abstract model checking* algorithm for deciding the circularity-freedom problem in linear recursive contract class tables performs the following steps. (*i*) *Find (linear) recursive methods*: by parsing the contract class table we create a graph where nodes are function names and, for every invocation of `D.n` in the body of `C.m`, there is an edge from `C.m` to `D.n`. Then a standard depth first search associates to every node a path of (mutual) recursive invocations (the paths starting and ending at that node, if any). The contract class table is linear recursive if every node has at most one associated path. (*ii*) *Computation of the orders*: given the list of recursive methods, we compute the corresponding mutations. (*iii*) *Evaluation process*: the contract pair corresponding to the main function is evaluated till every recursive function invocation has been unfolded up-to twice the corresponding order. (*iv*) *Detection of circularities*: this is performed with the same algorithm of the fixpoint analysis.

As regards the computational complexity, the contract inference system is polynomial time with respect to the length of the program in most of the cases [21]. The fixpoint analysis is is exponential in the number of cog names in a contract class table (because lams may double the size at every iteration). However, this exponential effect actually bites in practice. The abstract model checking is linear with respect to the length of the program as far as steps (i) and (ii) are concerned. Step (iv) is linear with respect to the size of the final lam. The critical step is (iii), which may be exponential with respect to the length of the program. Below, there is an overestimation of the computational complexity. Let

$$\begin{array}{ll} \multicolumn{2}{l}{\langle \texttt{Math!fact\_nc}\,[\mathit{cog} : c](\_) \to \_\mathbf{,}\, 0\rangle_{\text{start}}} \\ \longrightarrow & \langle\langle 0\mathbf{,}\, 0 + \texttt{Math!fact\_nc}[\mathit{cog} : c'](\_) \to \_.(c, c')\rangle_c\mathbf{,}\, 0\rangle_{\text{start}} \\ \longrightarrow & \langle\langle 0\mathbf{,}\, 0 + \langle 0\mathbf{,}\, 0 + \texttt{Math!fact\_nc}[\mathit{cog} : c''](\_) \to \_.(c', c'')\rangle_{c'}.(c, c')\rangle_c\mathbf{,}\, 0\rangle_{\text{start}} \end{array}$$

Fig. 21 Reduction for contract of method `Math.fact_nc`.

$$\begin{array}{l} ([\![\langle\langle 0\mathbf{,}\, 0 + \langle 0\mathbf{,}\, 0 + \texttt{Math!fact\_nc}[\mathit{cog} : c''](\_) \to \_.(c', c'')\rangle_{c'}.(c, c')\rangle_c\mathbf{,}\, 0\rangle_{\text{start}}]\!])^\flat \\ = (\langle\langle\langle 0\mathbf{,}\, 0 + \langle 0\mathbf{,}\, 0 + \langle\, \mathbf{[}(c', c'')\mathbf{]}\mathbf{,}\, 0\rangle\rangle.(c, c')\rangle\mathbf{,}\, 0\rangle)^\flat \\ = \langle 0 + 0 + \mathbf{[}(c', c'')\mathbf{]}\,\&(c, c')\mathbf{,}\, 0\rangle \\ = \langle\, \mathbf{[}(c', c''), (c, c')\mathbf{]}\mathbf{,}\, 0\rangle \end{array}$$

Fig. 22 Flattening and evaluation of resulting contract of method `Math.fact_nc`.

$\mathfrak{o}_{max}$ be the largest order of a recursive method contract (without loss of generality, we assume there is no mutual recursion).

m_{max} be the maximal number of function invocations in a body or in the contract of the main function.

An upper bound to the length of the evaluation till the saturated state is

$$\sum_{0 \le i \le \ell} (2 \times \mathfrak{o}_{max} \times m_{max})^i,$$

where ℓ is the number of methods in the program. Let k_{max} be the maximal number of dependency pairs in a body. Then the size of the saturated state is $O(k_{max} \times (\mathfrak{o}_{max} \times m_{max})^\ell)$, which is also the computational complexity of the abstract model checking.

7.1 *Assessments*

We tested `DF4ABS` on a number of medium-size programs written for benchmarking purposes by `core ABS` programmers and on an industrial case study based on the Fredhopper Access Server (FAS)[5] developed by SDL Fredhopper [10], which provides search and merchandising IT services to e-Commerce companies. The (leftmost two columns of the) table in Figure 23 reports the experiments: for every program we display the number of lines, whether the analysis has reported a deadlock (`D`) or not (✓), the time in seconds required for the analysis. Concerning time, we only report the time of the analysis of `DF4ABS` (and not the one taken by the inference) when they run on a QuadCore 2.4GHz and Gentoo (Kernel 3.4.9).

The rightmost column of the table in Figure 23 reports the results of another tool that has also been developed for the deadlock analysis of `core ABS` programs: `DECO` [13]. This technique integrates a point-to analysis with an analysis returning (an over-approximation of) program points that may be running in parallel. As highlighted by the above table, the three tools return the results as regards deadlock analysis, but are different as regards performance. In particular the fixpoint and model-checking analysis of `DF4ABS` are comparable on small/mid-size programs, `DECO` appears less performant (except for `PeerToPeer`, where our model-checking analysis is quite slow because of the number of dependencies produced by the underlying algorithm). On the `FAS module`, our two analysis are again comparable, while `DECO` has a better performance (`DECO` worst case complexity is cubic in the size of the input).

Few remarks about the precision of the techniques follow. `DF4ABS`/model-check is the most powerful tool we are aware of for linear recursive contract class table. For instance, it correctly detect the deadlock-freedom of the method `Math.fact_nc` (previously defined in Figure 5) while `DF4ABS`/fixpoint signals a false positive. Similarly, `DECO` signals a false positive deadlock for the following program, whereas `DF4ABS`/model-check returns its deadlock-freedom.

```
class C implements C {
       Unit m(C c){ C w ;
              w = new cog C() ;
              w!m(this) ;
              c!n(this) ;
       }
       Unit n(C a){ Fut<Unit> x ;
              x = a!q() ;
       x.get ;
       }
       Unit q(){ }
}
{  C a; C b ;
   Fut<Unit> x ;
   a = new cog C() ;
   b = new cog C() ;
   x = a!m(b) ;
}
```

However, `DF4ABS`/model-check is not defined on non-linear recursive contract class tables. Non-linear recursive contract class tables can easily be defined, as shown

[5] Actually, the FAS module has been written in `ABS` [10], and so, we had to adapt it in order to conform with `core ABS` restrictions (see Section 3). This adaptation just consisted of purely syntactic changes, and only took half-day work (see also the comments in [15]).

program	lines	`DF4ABS`/fixpoint result	time	`DF4ABS`/model-check result	time	`DECO` result	time
`PingPong`	61	✓	0.311	✓	0.046	✓	1.30
`MultiPingPong`	88	`D`	0.209	`D`	0.109	`D`	1.43
`BoundedBuffer`	103	✓	0.126	✓	0.353	✓	1.26
`PeerToPeer`	185	✓	0.320	✓	6.070	✓	1.63
`FAS Module`	2645	✓	31.88	✓	39.78	✓	4.38

Fig. 23 Assessments of `DF4ABS`.

with the following two contracts:

$$\begin{aligned}\texttt{C.m} &= [\mathit{cog} : c]\ ()\ \{\langle 0, (\texttt{C!m}\,[\mathit{cog} : c]() \to \_).(c, c') \\ &\quad + \texttt{C!n}\,[\mathit{cog} : c''](\,[\mathit{cog} : c]) \to \_\rangle\} \to \_ \\ \texttt{C.n} &= [\mathit{cog} : c]\ ([\mathit{cog} : c']) \\ &\quad \{\langle(\texttt{C!m}\,[\mathit{cog} : c]() \to \_).(c, c'), 0\rangle\} \to \_\end{aligned}$$

Here, `DF4ABS`/model-check fails to analyse `C.m` while `DF4ABS`/fixpoint and `DECO` successfully recognise as deadlock-free[6]. We conclude this section with a remark about the proportion between programs with linear recursive contract class tables and those with nonlinear ones. While this proportion is hard to assess, our preliminary analyses strengthen the claim that nonlinear recursive programs are rare. We have parsed the three case-studies developed in the European project HATS [10]. The case studies are the FAS module, a Trading System (TS) modelling a supermarket handling sales, and a Virtual Office of the Future (VOF) where office workers are enabled to perform their office tasks seamlessly independent of their current location. FAS has 2645 code-lines, TS has 1238 code-lines, and VOF has 429 code-lines. In none of them we found a nonlinear recursion in the corresponding contract class table, TS and VOF have respectively 2 and 3 linear recursive method contracts (there are recursions in functions on data-type values that have nothing to do with locks and control). This substantiates the usefulness of our technique in these programs; the analysis of a wider range of programs is matter of future work.

8 Related works

A preliminary theoretical study was undertaken in [17], where (*i*) the considered language is a functional subset of `core ABS`; (*ii*) contracts are not inferred, they are provided by the programmer and type-checked; (*iii*) the deadlock analysis is less precise because it is not iterated as in this contribution, but stops at the first approximant, and (*iv*), more importantly, method contracts are not pairs of lams, which led it to discard dependencies (thereby causing the analysis, in some cases, to erroneously yield false negatives). This system has been improved in [15] by modelling method contracts as pairs of lams, thus supporting a more precise fixpoint technique. The contract inference system of [15] has been extended in this contribution with the management of aliases of futures and with the dichotomy of present contract and future contract in the inference rules of statements.

The proposals in the literature that statically analyse deadlocks are largely based on (behavioural) types. In [1, 4, 12, 36] a type system is defined that computes a partial order of the locks in a program and a subject reduction theorem demonstrates that tasks follow this order. Similarly to these techniques, the tool `Java PathFinder` [37] computes a tree of lock orders for every method and searches for mismatches between such orderings. On the contrary, our technique does not compute any ordering of locks during the inference of contracts, thus being more flexible: a computation may acquire two locks in different order at different stages, being correct in our case, but incorrect with the other techniques. The Extended Static Checking for Java [11] is an automatic tool for contract-based programming: annotation are used to specify loop invariants, pre and post conditions, and to catch deadlocks. The tool warns the programmer if the annotations cannot be validated. This techniques requires that annotations are explicitly provided by the programmer, while they are inferred in `DF4ABS`.

A well-known deadlock analyser is TyPiCal, a tool that has been developed for pi-calculus by Kobayashi [22, 26–28]. TyPiCal uses a clever technique for deriving inter-channel dependency information and is able to deal with several recursive behaviours and the creation of new channels without committing to any pre-defined order of channel names. Nevertheless, since TyPiCal is based on an inference system, there are recursive behaviours that escape its accuracy. For instance, it returns false positives when recursion create networks with arbitrary numbers of nodes. To illustrate the issue

[6] In [19], we have defined a source-to-source transformation taking nonlinear recursive contract class tables and returning linear recursive ones. This transformation introduces fake cog dependencies that returns a false positive when applying `DF4ABS`/model-check on the example above.

we consider the following deadlock-free program computing factorial

```
class Math implements Math {
   Int fact(Int n, Int r){
        Math y ;
        Fut<Int> v ;
        if (n == 0) return r ;
        else { y = new cog Math() ;
               v = y!fact(n-1, n*r) ;
               w = v.get ;
               return w ;
        }
   }
}
{
        Math x ; Fut<Int> fut ; Int r ;
        x = new cog Math();
        fut = x!fact(6,1);
        r = fut.get ;
}
```

that is a variation of the method `Math.fact_ng` in Figure 5. This code is deadlock free according to `DF4ABS`/model-check, however, its implementation in pi-calculus [7] is not deadlock-free according to TyPiCal. The extension of TyPiCal with a technique similar to the one in Section 6, but covering the whole range of lam programs, has been recently defined in [16].

Type-based deadlock analysis has also been studied in [34]. In this contribution, types define objects' states and can express acceptability of messages. The exchange of messages modifies the state of the objects. In this context, a deadlock is avoided by setting an ordering on types. With respect to our technique, [34] uses a deadlock prevention approach, rather than detection, and no inference system for types is provided.

In [33], the author proposes two approaches for a type and effect-based deadlock analysis for a concurrent extension of ML. The first approach, like our ones, uses a type and effect inference algorithm, followed by an analysis to verify deadlock freedom. However, their analysis approximates infinite behaviours with a chaotic behaviour that non-deterministically acquires and releases locks, thus becoming imprecise. For instance, the previous example should be considered a potential deadlock in their approach. The second approach is an initial result on a technique for reducing deadlock analysis to data race analysis.

Model-theoretic techniques for deadlock analysis have also been investigated. defined. In [5], circular dependencies among processes are detected as erroneous configurations, but dynamic creation of names is not treated. Similarly in [2] (see the discussion below).

Works that specifically tackle the problem of deadlocks for languages with the same concurrency model as that of `core ABS` are the following: [38] defines an approach for deadlock prevention (as opposed to our deadlock detection) in SCOOP, an Eiffel-based concurrent language. Different from our approach, they annotate classes with the used *processors* (the analogue of cogs in `core ABS`), while this information is inferred by our technique. Moreover each method exposes preconditions representing required lock ordering of processors (processors obeys an order in which to take locks), and this information must be provided by the programmer. [2] studies a Petri net based analysis, reducing deadlock detection to a reachability problem in Petri nets. This technique is more precise in that it is thread based and not just object based. Since the model is finite, this contribution does not address the feature of object creation and it is not clear how to scale the technique. We plan to extend our analysis in order to consider finer-grained thread dependencies instead of just object dependencies. [25] offers a design pattern methodology for CoJava to obtain deadlock-free programs. CoJava, a Java dialect where data-races and data-based deadlocks are avoided by the type system, prevents threads from sharing mutable data. Deadlocks are excluded by a programming style based on ownership types and *promise* (i.e. future) objects. The main differences with our technique are (*i*) the needed information must be provided by the programmer, (*ii*) deadlock freedom is obtained through ordering and timeouts, and (*iii*) no guarantee of deadlock freedom is provided by the system.

The relations with the work by Flores-Montoya *et al.* [13] has been largely discussed in Section 7. Here we remark that, as regards the design, `DECO` is a monolithic code written in Prolog. On the contrary, `DF4ABS` is a highly modular Java code. Every module may be replaced by another; for instance one may rewrite the inference system for another language and plug it easily in the tool, or one may use a different/refined contract analysis algorithm, in particular one used in `DECO` (see Conclusions).

[7] The pi-calculus factorial program is

```
*factorial?(n,(r,s)).
    if n=0 then r?m. s!m else new t in
        (r?m. t!(m*n)) | factorial!(n-1,(t,s))
```

In this code, `factorial` returns the value (on the channel `s`) by *delegating* this task to the recursive invocation, if any. In particular, the initial invocation of `factorial`, which is `r!1 | factorial!(n,(r,s))`, performs a synchronisation between `r!1` and the input `r?m` in the continuation of `factorial?(n,(r,s))`. In turn, this may delegate the computation of the factorial to a subsequent synchronisation on a new channel `t`. TyPiCal signals a deadlock on the two inputs `r?m` because it fails in connecting the output `t!(m*n)` with them.

9 Conclusions

We have developed a framework for detecting deadlocks in `core ABS` programs. The technique uses (i) an inference algorithm to extract abstract descriptions of methods, called contracts, (ii) an evaluator of contracts, which computes an over-approximated fixpoint semantics, (iii) a model checking algorithm that evaluates contracts by unfolding method invocations.

This study can be extended in several directions. As regards the prototype, the next release will provide indications about *how* deadlocks have been produced by pointing out the elements in the code that generated the detected circular dependencies. This way, the programmer will be able to check whether or not the detected circularities are actual deadlocks, fix the problem in case it is a verified deadlock, or be assured that the program is deadlock-free.

`DF4ABS`, being modular, may be integrated with other analysis techniques. In fact, in collaboration with Kobayashi [16], we have recently defined a variant of the model checking algorithm that has no linearity restriction. For the same reason, another direction of research is to analyse contracts with the point-to analysis technique of `DECO` [13]. We expect that such analyser will be simpler than `DECO` because, after all, contracts are simpler than `core ABS` programs.

Another direction of research is the application of our inference system to other languages featuring asynchronous method invocation, possibly after removing or adapting or adding rules. One such language that we are currently studying is `ASP` [7]. While we think that our framework and its underlying theory are robust enough to support these applications, we observe that a necessary condition for demonstrating the results of correctness of the framework is that the language has a formal semantics.

References

1. Abadi, M., Flanagan, C., Freund, S.N.: Types for safe locking: Static race detection for java. ACM Trans. Program. Lang. Syst. **28** (2006)
2. de Boer, F., Bravetti, M., Grabe, I., Lee, M., Steffen, M., Zavattaro, G.: A petri net based analysis of deadlocks for active objects and futures. In: Proc. of Formal Aspects of Component Software - 9th International Workshop, FACS 2012, *Lecture Notes in Computer Science*, vol. 7684, pp. 110–127. Springer (2012)
3. de Boer, F., Clarke, D., Johnsen, E.: A complete guide to the future. In: Progr. Lang. and Systems, *LNCS*, vol. 4421, pp. 316–330. Springer (2007)
4. Boyapati, C., Lee, R., Rinard, M.: Ownership types for safe program.: preventing data races and deadlocks. In: Proc. OOPSLA '02, pp. 211–230. ACM (2002)
5. Carlsson, R., Millroth, H.: On cyclic process dependencies and the verification of absence of deadlocks in reactive systems (1997)
6. Caromel, D.: Towards a method of object-oriented concurrent programming. Commun. ACM **36**(9), 90–102 (1993)
7. Caromel, D., Henrio, L., Serpette, B.P.: Asynchronous and deterministic objects. In: Proc. POPL'04, pp. 123–134. ACM (2004)
8. Comtet, L.: Advanced Combinatorics: The Art of Finite and Infinite Expansions. Dordrecht, Netherlands (1974)
9. Coppo, M.: Type Inference with Recursive Type Equations. In: Proc. FoSSaCS, *LNCS*, vol. 2030, pp. 184–198. Springer (2001)
10. Requirement elicitation (2009). Deliverable 5.1 of project FP7-231620 (HATS), available at `http://www.hats-project.eu/sites/default/files/Deliverable51_rev2.pdf`
11. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for java. SIGPLAN Not. **37**(5), 234–245 (2002)
12. Flanagan, C., Qadeer, S.: A type and effect system for atomicity. In: In PLDI 03: Programming Language Design and Implementation, pp. 338–349. ACM (2003)
13. Flores-Montoya, A., Albert, E., Genaim, S.: May-happen-in-parallel based deadlock analysis for concurrent objects. In: Proc. FORTE/FMOODS 2013, *Lecture Notes in Computer Science*, vol. 7892, pp. 273–288. Springer (2013)
14. Gay, S., Hole, M.: Subtyping for session types in the π-calculus. Acta Informatica **42**(2-3), 191–225 (2005)
15. Giachino, E., Grazia, C.A., Laneve, C., Lienhardt, M., Wong, P.Y.H.: Deadlock analysis of concurrent objects: Theory and practice. In: iFM'13, *LNCS*, vol. 7940, pp. 394–411. Springer-Verlag (2013)
16. Giachino, E., Kobayashi, N., Laneve, C.: Deadlock detection of unbounded process networks. In: Proceedings of CONCUR 2014, *LNCS*, vol. 8704, pp. 63–77. Springer-Verlag (2014)
17. Giachino, E., Laneve, C.: Analysis of deadlocks in object groups. In: FMOODS/FORTE, *Lecture Notes in Computer Science*, vol. 6722, pp. 168–182. Springer-Verlag (2011)
18. Giachino, E., Laneve, C.: A beginner's guide to the deadLock Analysis Model. In: Trustworthy Global Computing - 7th International Symposium, TGC 2012, Revised Selected Papers, *Lecture Notes in Computer Science*, vol. 8191, pp. 49–63. Springer (2013)
19. Giachino, E., Laneve, C.: Deadlock detection in linear recursive programs. In: Proceedings of SFM-14:ESM, *LNCS*, vol. 8483, pp. 26–64. Springer-Verlag (2014)
20. Giachino, E., Lascu, T.A.: Lock Analysis for an Asynchronous Object Calculus. In: Proc. 13th ICTCS (2012)
21. Henglein, F.: Type inference with polymorphic recursion. ACM Trans. Program. Lang. Syst. **15**(2), 253–289 (1993)
22. Igarashi, A., Kobayashi, N.: A generic type system for the pi-calculus. Theor. Comput. Sci. **311**(1-3), 121–163 (2004)
23. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A core language for abstract behavioral specification. In: B. Aichernig, F.S. de Boer, M.M. Bonsangue (eds.) Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010), *LNCS*, vol. 6957, pp. 142–164. Springer-Verlag (2011)
24. Johnsen, E.B., Owe, O.: An asynchronous communication model for distributed concurrent objects. Software and System Modeling **6**(1), 39–58 (2007)

25. Kerfoot, E., McKeever, S., Torshizi, F.: Deadlock freedom through object ownership. In: T. Wrigstad (ed.) 5rd International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO), in conjunction with ECOOP 2009 (2009)
26. Kobayashi, N.: A partially deadlock-free typed process calculus. TOPLAS **20**(2), 436–482 (1998)
27. Kobayashi, N.: A new type system for deadlock-free processes. In: Proc. CONCUR 2006, *LNCS*, vol. 4137, pp. 233–247. Springer (2006)
28. Kobayashi, N.: TyPiCal (2007). At `kb.ecei.tohoku.ac.jp /~koba/typical/`
29. Laneve, C., Padovani, L.: The *must* preorder revisited. In: Proc. CONCUR 2007, *LNCS*, vol. 4703, pp. 212–225. Springer (2007)
30. Milner, R.: A Calculus of Communicating Systems. Springer (1982)
31. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, ii. Inf. and Comput. **100**, 41–77 (1992)
32. Naik, M., Park, C.S., Sen, K., Gay, D.: Effective static deadlock detection. In: IEEE 31st International Conference on Software Engineering, 2009. ICSE 2009., pp. 386–396 (2009)
33. Pun, K.I.: behavioural static analysis for deadlock detection. Ph.D. thesis, Faculty olf Mathematics and Natural Sciences, University of Oslo, Norway (2013)
34. Puntigam, F., Peter, C.: Types for active objects with static deadlock prevention. Fundam. Inform. **48**(4), 315–341 (2001)
35. Tarjan, R.E.: Depth-first search and linear graph algorithms. SIAM J. Comput. **1**(2), 146–160 (1972)
36. Vasconcelos, V.T., Martins, F., Cogumbreiro, T.: Type inference for deadlock detection in a multithreaded polymorphic typed assembly language. In: Proc. PLACES'09, *EPTCS*, vol. 17, pp. 95–109 (2009)
37. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. Automated Software Engineering **10**(2), 203–232 (2003)
38. West, S., Nanz, S., Meyer, B.: A modular scheme for deadlock prevention in an object-oriented programming model. In: ICFEM, pp. 597–612 (2010)
39. Wong, P.Y.H., Albert, E., Muschevici, R., Proença, J., Schäfer, J., Schlatte, R.: The ABS tool suite: modelling, executing and analysing distributed adaptable object-oriented systems. Journal on Software Tools for Technology Transfer **14**(5), 567–588 (2012)
40. Yonezawa, A., Briot, J.P., Shibayama, E.: Object-oriented concurrent programming in ABCL/1. In: Proc. OOPSLA'86, pp. 258–268 (1986)

A Properties of Section 4

The *initial configuration* of a well-typed `core ABS` program is

$$ob(start, \varepsilon, \{[\text{destiny} \mapsto f_{start}, \overline{x} \mapsto \bot] \mid s\}, \varnothing)\ cog(\text{start}, start)$$

where the activity $\{[\text{destiny} \mapsto f_{start}, \overline{x} \mapsto \bot] \mid s\}$ corresponds to the activation of the main function. A *computation* is a sequence of reductions starting at the initial configuration according to the operational semantics. We show in this appendix that such computations keep configurations well-typed; in particular, we show that the sequence of contracts corresponding to the configurations of the computations is in the *later-stage relationship* (see Figure 28).

Runtime contracts. In order to type the configurations we use a *runtime type system*. To this aim we extend the syntax of contracts in Figure 8 and define *extended futures* F, *extended contracts* that, with an abuse of notation, we still denote $\mathbb{c}$ and *runtime contracts* $\mathbb{k}$ as follows:

$$F ::= f \mid \imath_f$$

$$\mathbb{c} ::= \textit{as in Figure 8} \mid f \mid f.(c, c') \mid f.(c, c')^{\mathtt{w}} \mid \langle \mathbb{c}, \mathbb{c} \rangle^{c}$$

$$\mathbb{k} ::= 0 \mid \langle \mathbb{c}, \mathbb{c} \rangle_f^{c} \mid [\mathtt{C!m}\ \mathbb{r}(\overline{\mathbb{r}}) \rightarrow \mathbb{r}]_f \mid \mathbb{k} \parallel \mathbb{k}$$

As regards F, they are introduced for distinguishing two kind of future names: i) f that has been used in the contract inference system as a *static time* representation of a future, but is now used as its *runtime* representation; ii) $\imath_f$ now replacing f in its role of *static time* future (it's typically used to reference a future that isn't created yet).

As regards $\mathbb{c}$ and $\mathbb{k}$, the extensions are motivated by the fact that, at runtime, the informations about contracts are scattered in all the configuration. However, when we plug all the parts to type the whole configuration, we can merge the different informations to get a runtime contract $\mathbb{k}'$ such that every contract $\mathbb{c} \in \mathbb{k}'$ does not contain any reference to futures anymore. This merging is done using a set of rewriting rules $\Rightarrow$ defined in Figure 24 that let one replace the occurrences of runtime futures in runtime contracts $\mathbb{k}$ with the corresponding contract of the future. We write $f \in \mathit{names}(\mathbb{k})$ whenever f occurs in $\mathbb{k}$ not as an index. The substitution $\mathbb{k}[^{\mathbb{C}}/_{f}]$ replaces the occurrences of f in contracts $\mathbb{c}''$ of $\mathbb{k}$ (by definition of our configurations, in these cases f can never occur as index in $\mathbb{k}$). It is easy to demonstrate that the merging process always terminates and is confluent for non-recursive contracts and, in the following, we let $(\!|\mathbb{k}|\!)$ be the *normal form* of $\mathbb{k}$ with respect to $\Rightarrow$:

Definition 11 A runtime contract $\mathbb{k}$ is *non-recursive* if:

- all futures $f \in \mathit{names}(\mathbb{k})$ are declared once in $\mathbb{k}$
- all futures $f \in \mathit{names}(\mathbb{k})$ are not recursive, i.e. for all $\langle \mathbb{c}, \mathbb{c}' \rangle_f^{c} \in \mathbb{k}$, we have $f \notin \mathit{names}(\langle \mathbb{c}, \mathbb{c}' \rangle_f^{c})$

Typing Runtime Configurations. The typing rules for the runtime configuration are given in Figures 25, 26 and 27. Except for few rules (in particular, those in Figure 25 which type the runtime objects of a configuration), all the typing rules have a corresponding one in the contract inference system defined in Section 4. Additionally, the typing judgments are identical to the corresponding one in the inference system, with three minor differences:

i) the typing environment, that now contains a reference to the contract class table and mappings object names to pairs $(\mathtt{C}, \mathbb{r})$, is called Δ;
ii) the typing rules do not collect constraints;
iii) the $\mathit{rt\_unsync}(\cdot)$ function on environments Δ is similar to $\mathit{unsync}(\cdot)$ in Section 4, except that it now grabs all $\imath_f$ and all futures f' that was created by the current thread f. More precisely

$$\mathit{rt\_unsync}(\Delta, f) \overset{def}{=} \mathbb{c}_1 \parallel \cdots \parallel \mathbb{c}_n \parallel f_1 \parallel \cdots \parallel f_m$$

where $\{\mathbb{c}_1, \cdots, \mathbb{c}_n\} = \{\mathbb{c}' \mid \exists \imath_f, \mathbb{r} :\ \Delta(\imath_f) = (\mathbb{r}, \mathbb{c}')\}$ and $\{f_1, \ldots, f_m\} = \{f' \mid \Delta(f') = (\mathbb{r}', f)\}$.

Finally, few remarks about the auxiliary functions:

$$\frac{f \in \mathit{names}(\mathbb{k})}{\mathbb{k} \| \langle \mathbb{c}, \mathbb{c}' \rangle_f^c \Rightarrow \mathbb{k}[{}^{\langle \mathbb{c}, \mathbb{c}' \rangle^c}/_f]} \qquad \frac{f \in \mathit{names}(\mathbb{k})}{\mathbb{k} \| [\texttt{C!m}\ \mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{r}]_f \Rightarrow \mathbb{k}[{}^{\texttt{C!m}\ \mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{r}}/_f]}$$

Fig. 24 Definition of $\Rightarrow$

– $\mathit{init}(\texttt{C}, o)$ is supposed to return the init activity of the class C. However, we have assumed that these activity is always empty, see Footnote 2. Therefore the corresponding contract will be $\langle 0, 0 \rangle$.
– $\mathit{atts}(\texttt{C}, \overline{v}, o, c)$ returns a substitution provided that $\overline{v}$ have records $\overline{\mathbb{r}}$ and o and c are object and cog identifiers, respectively.
– $\mathit{bind}(o, f, \texttt{m}, \overline{v'}, \texttt{C})$ returns the activity corresponding to the method C.m with the parameters $\overline{v'}$ provided that f has type $c \rightsquigarrow \mathbb{r}$ and $\overline{v'}$ have the types $\overline{\mathbb{r}'}$.

Theorem 3 *Let* $P = \overline{I}\ \overline{C}\ \{\overline{T\ x}; s\}$ *be a* `core ABS` *program and let* $\Gamma \vdash P : \textsc{cct}, \langle \mathbb{c}, \mathbb{c}' \rangle \rhd \mathcal{U}$. *Let also* σ *be a substitution satisfying* $\mathcal{U}$ *and*

$$\Delta = \sigma(\Gamma + \textsc{cct}) + \mathit{start} : [\mathit{cog} : \text{start}] + f_{\mathit{start}} : (\text{start} \rightsquigarrow \_, 0)$$

Then

$$\Delta \vdash_R \mathit{ob}(\mathit{start}, \varepsilon, \{l \mid s\}, \varnothing)\ \mathit{cog}(\text{start}, \mathit{start}) : \sigma(\langle \mathbb{c}, \mathbb{c}' \rangle)^{\text{start}}_{f_{\mathit{start}}}$$

where $l = [\mathit{destiny} \mapsto f_{\mathit{start}}, \overline{x \mapsto \bot}]$.

Proof By (TR-Configuration) and (TR-Object) we are reduced to prove:

$$\Delta \vdash_R^{\text{start}, \mathit{start}} \{\text{destiny} \mapsto f_{\mathit{start}}, \overline{x \mapsto \bot} | s\} : \sigma(\langle \mathbb{c}, \mathbb{c}' \rangle)^{\text{start}}_{f_{\mathit{start}}} \quad (3)$$

To this aim, let $\overline{X}$ be the variables used in the inference rule of (T-Program).

To demonstrate (3) we use (TR-Process). Therefore we need to prove:

$$\Delta[\text{destiny} \mapsto f_{\mathit{start}}, \overline{x \mapsto \sigma(X)}] \vdash_R^{\text{start}, \mathit{start}} s : \sigma(\mathbb{c}) \mid \Delta'$$

with $\mathit{rt\_unsync}(\Delta') = \sigma(\mathbb{c}'')$. This proof is done by a standard induction on s, using a derivation tree identical to the one used for the inference (with the minor exception of replacing the fs used in the inference with corresponding $\imath_f$s). This is omitted because straightforward. □

Definition 12 A runtime contract $\mathbb{k}$ is *well-formed* if it is non recursive and if futures and method calls in $\mathbb{k}$ are placed as described by the typing rules: i.e. in a sequence $\mathbb{c}_1 \fatsemi \ldots \fatsemi \mathbb{c}_n$, they are present in all $\mathbb{c}_i$, $i_1 \leq i \leq i_k$ with $\mathbb{c}_{i_1}$ being when the method is called, and $\mathbb{c}_{i_k}$ being when the method is synchronised with. Formally, for all $\langle \mathbb{c}, \mathbb{c}' \rangle_f^c \in \mathbb{k}$, we can derive $\varnothing \vdash \mathbb{c} : \mathbb{c}'$ with the following rules:

$$0 \vdash 0 : 0 \qquad 0 \vdash \texttt{C.m}\ \mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{r}' : 0 \qquad \frac{\mathbb{c}' = 0 \vee \mathbb{c}' = f}{\mathbb{c}' \vdash f : f}$$

$$\frac{\begin{array}{c}\mathbb{c} = \texttt{C!m}\ \mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{r}' \\ \mathbb{c}' = 0 \vee \mathbb{c}' = \mathbb{c}\end{array}}{\mathbb{c}' \vdash \mathbb{c} : \mathbb{c}} \qquad \frac{\mathbb{c}' = 0 \vee \mathbb{c}' = f}{\mathbb{c}' \vdash f\centerdot(c, c')^{[\texttt{w}]} \vdash 0} \qquad \frac{\begin{array}{c}\mathbb{c} = \texttt{C!m}\ \mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{r}' \\ \mathbb{c}' = 0 \vee \mathbb{c}' = \mathbb{c}\end{array}}{\mathbb{c}' \vdash \mathbb{c}\centerdot(c, c')^{[\texttt{w}]} : 0}$$

$$\frac{\mathbb{c}' \vdash \mathbb{c}_1 : \mathbb{c}'' \qquad \mathbb{c}'' \vdash \mathbb{c}_2 : \mathbb{c}'''}{\mathbb{c}' \vdash \mathbb{c}_1 \fatsemi \mathbb{c}_2 : \mathbb{c}'''} \qquad \frac{\mathbb{c}' \vdash \mathbb{c}_1 : \mathbb{c}''}{\mathbb{c}' \vdash \mathbb{c}_1 \fatsemi 0 : \mathbb{c}''}$$

$$\frac{\mathbb{c}' \vdash \mathbb{c}_1 : \mathbb{c}'' \qquad \mathbb{c}' \vdash \mathbb{c}_2 : \mathbb{c}''}{\mathbb{c}' \vdash \mathbb{c}_1 + \mathbb{c}_2 : \mathbb{c}''} \qquad \frac{\mathbb{c}_1' \vdash \mathbb{c} : \mathbb{c}_1'' \qquad \mathbb{c}_2' \vdash \mathbb{c}' : \mathbb{c}_2''}{\mathbb{c}_1' \parallel \mathbb{c}_2' \vdash \mathbb{c} \parallel \mathbb{c}' : \mathbb{c}_1'' \parallel \mathbb{c}_2''}$$

Lemma 1 *If* $\Delta \vdash \mathit{cn} : \mathbb{k}$ *is a valid statement, then* $\mathbb{k}$ *is well-formed.*

Proof The result is given by the way $\mathit{rt\_unsync}(\cdot)$ is used in the typing rules. □

In the following theorem we use the so-called *later-stage relation* $\trianglerighteq$ that has been defined in Figure 28 on runtime contracts.

We observe that the later-stage relation uses a substitution process that *also performs a pattern matching operation* – therefore it is partial because the pattern matching may fail. In particular, $[{}^{\mathbb{s}}/_{\mathbb{r}}]$ (i) extracts the cog names and terms $\mathbb{s}'$ in $\mathbb{s}$ that corresponds to occurrences of cog names and record variables in $\mathbb{r}$ and (ii) returns the corresponding substitution.

Theorem 4 (Subject Reduction) *Let* $\Delta \vdash_R \mathit{cn} : \mathbb{k}$ *and* $\mathit{cn} \to \mathit{cn}'$. *Then there exist* Δ', $\mathbb{k}'$, *and an injective renaming of cog names* $\imath$ *such that*
– $\Delta' \vdash_R \mathit{cn}' : \mathbb{k}'$ *and*
– $\imath(\mathbb{k}) \trianglerighteq \mathbb{k}'$.

Proof The proof is a case analysis on the reduction rule used in $\mathit{cn} \to \mathit{cn}'$ and we assume that the evaluation of an expression $[\![e]\!]_\sigma$ always terminates. We focus on the most interesting cases. We remark that the injective renaming $\imath$ is used to identify fresh cog names that are created by the static analysis with fresh cog names that are created by the operational semantics. In fact, the renaming is not the identity only in the case of cog creation (second case below).

– *Skip Statement.*

(Skip)
$$\mathit{ob}(o, a, \{l \mid \texttt{skip}; s\}, q) \to \mathit{ob}(o, a, \{l \mid s\}, q)$$

By (TR-Object), (TR-Process), (TR-Seq) and (TR-Skip), there exists Δ'' and $\mathbb{c}$ such that $\Delta'' \vdash_R^{c,o} \texttt{skip}; s : 0 \fatsemi \mathbb{c} \,|\, \Delta''$. It is easy to see that $\Delta'' \vdash_R^{c,o} s : \mathbb{c} \,|\, \Delta''$. Moreover, by (LS-Delete), we have $0 \fatsemi \mathbb{c} \trianglerighteq_{\texttt{cog}(o)} \mathbb{c}$ which proves that $\mathbb{k} \trianglerighteq \mathbb{k}'$.

– *Object creation.*

(New-Object)
$$\frac{o' = \text{fresh}(\texttt{C}) \quad p = \text{init}(\texttt{C}, o') \quad a' = \text{atts}(\texttt{C}, [\![\overline{e}]\!]_{(a+l)}, c)}{\begin{array}{c}\mathit{ob}(o, a, \{l \mid x = \texttt{new C}(\overline{e}); s\}, q)\ \mathit{cog}(c, o) \\ \to \mathit{ob}(o, a, \{l \mid x = o'; s\}, q)\ \mathit{cog}(c, o) \quad \mathit{ob}(o', a', \text{idle}, \{p\})\end{array}}$$

By (TR-Object) and (TR-Process), there exists Δ'' that extends Δ such that $\Delta'' \vdash_R^{c,o} \texttt{new C}(\overline{e}) : \mathbb{r}, 0 \mid \Delta''$. Let $\Delta' = \Delta[o' \mapsto \mathbb{r}]$. The theorem follows by the assumption that p is empty (see Footnote 2).

– *Cog creation.*

(New-Cog-Object)
$$\frac{\begin{array}{c}c' = \text{fresh}(\,) \quad o' = \text{fresh}(\texttt{C}) \quad p = \text{init}(\texttt{C}, o') \\ a' = \text{atts}(\texttt{C}, [\![\overline{e}]\!]_{(a+l)}, c')\end{array}}{\begin{array}{c}\mathit{ob}(o, a, \{l \mid x = \texttt{new cog C}(\overline{e}); s\}, q) \\ \to \mathit{ob}(o, a, \{l \mid x = o'; s\}, q) \quad \mathit{ob}(o', a', p, \varnothing) \quad \mathit{cog}(c', o')\end{array}}$$

By (TR-Object) and (TR-Process), there exists Δ'' that extends Δ such that $\Delta'' \vdash_R^{c,o} \texttt{new C}(\overline{e}) : [\mathit{cog} : c'', \overline{x : \mathbb{r}}],\ 0 \mid \Delta''$ for some c'' and records $\overline{\mathbb{r}}$. Let $\Delta' = \Delta[o' \mapsto [\mathit{cog} : c', \overline{x : \mathbb{r}}], c' \mapsto \mathit{cog}]$ and $\imath(c'') = c'$, where $\imath$ is an injective renaming on cog names. The theorem follows by the assumption that p is empty (see Footnote 2).

$$\text{(TR-FUTURE-TICK)}\quad \frac{\Delta(f) = (c \rightsquigarrow \mathbb{r}, \mathbb{c})^{\checkmark} \qquad \Delta \vdash \mathit{val} : \mathbb{r}}{\Delta \vdash_R \mathit{fut}(f, \mathit{val}) : 0}$$

$$\text{(TR-FUTURE)}\quad \frac{\Delta(f) = (c \rightsquigarrow \mathbb{r}, \mathbb{c})}{\Delta \vdash_R \mathit{fut}(f, \bot) : 0}$$

$$\text{(TR-INVOC)}\quad \frac{\begin{array}{c}\Delta(f) = (c \rightsquigarrow \mathbb{r}', \mathbb{c}) \\ \Delta \vdash_R \overline{v} = \overline{\mathbb{r}} \qquad \Delta(o) = [\mathit{cog} : c, \overline{x{:}\mathbb{r}}]\end{array}}{\Delta \vdash_R \mathit{invoc}(o, f, m, \overline{v}) : [\mathtt{C!m}\ [\mathit{cog} : c, \overline{x{:}\mathbb{r}}](\overline{\mathbb{s}}) \to \mathbb{r}']_f}$$

$$\text{(TR-OBJECT)}\quad \frac{\begin{array}{c}\Delta(o) = [\mathit{cog} : c, \overline{x{:}\mathbb{r}}] \qquad \Delta \vdash_R^{c,o} \overline{\mathit{val}} : \overline{\mathbb{r}} \\ \Delta \vdash_R^{c,o} p : \mathbb{k} \qquad \Delta \vdash_R^{c,o} \overline{p} : \overline{\mathbb{k}}\end{array}}{\Delta \vdash_R \mathit{ob}(o, [\mathit{cog} \mapsto c; \overline{x \mapsto \mathit{val}}], p, \overline{p}) : \mathbb{k} \| \overline{\mathbb{k}}}$$

$$\text{(TR-PROCESS)}\quad \frac{\begin{array}{c}\Delta \vdash_R^{c,o} \overline{\mathit{val} : \mathbb{x}} \qquad \Delta(f) = (c \rightsquigarrow \mathbb{r}', \overline{f'})^{[\checkmark]} \\ \Delta[\text{destiny} \mapsto f, \overline{x \mapsto \mathbb{x}}] \vdash_R^{c,o} s : \mathbb{c} \mid \Delta''\end{array}}{\Delta \vdash_R^{c,o} \{\text{destiny} \mapsto f, \overline{x \mapsto \mathit{val}} \mid s\} : \langle \mathbb{c}, \mathit{rt\_unsync}(\Delta'', f)\rangle_f^c}$$

$$\text{(TR-IDLE)}\quad \Delta \vdash_R^{c,o} \mathtt{idle} : 0$$

$$\text{(TR-PARALLEL)}\quad \frac{\Delta \vdash_R \mathit{cn}_1 : \mathbb{k}_1 \qquad \Delta \vdash_R \mathit{cn}_2 : \mathbb{k}_2}{\Delta \vdash_R \mathit{cn}_1\ \mathit{cn}_2 : \mathbb{k}_1 \parallel \mathbb{k}_2}$$

Fig. 25 The typing rules for runtime configurations.

– *Asynchronous calls.*

$$\text{(ASYNC-CALL)}\quad \frac{o' = [\![e]\!]_{(a+l)} \quad \overline{v} = [\![\overline{e}]\!]_{(a+l)} \quad f = \text{fresh}(\ )}{\begin{array}{c}\mathit{ob}(o, a, \{l \mid x = e!\mathtt{m}(\overline{e}); s\}, q) \\ \to \mathit{ob}(o, a, \{l \mid x = f; s\}, q)\ \mathit{invoc}(o', f, \mathtt{m}, \overline{v})\ \mathit{fut}(f, \bot)\end{array}}$$

By (TR-OBJECT) and (TR-PROCESS), there exist $\overline{\mathbb{r}}$, Δ'_1, $\mathbb{c}$ and $\mathbb{k}''$ such that (let $f' = l(\text{destiny})$)

- $\mathbb{k} = \langle \mathbb{c}, \mathit{rt\_unsync}(\Delta'_1, f')\rangle_{\mathtt{cog}(o)}^{f'} \| \mathbb{k}''$
- $\Delta \vdash_R^{c,o} \overline{v} : \overline{\mathbb{x}}$ (with $l = [\overline{y \mapsto v}]$)
- $\Delta \vdash_R^{c,o} q : \mathbb{k}''$
- $\Delta[\overline{y \mapsto \mathbb{r}}] \vdash_R^{c,o} x = e!m(\overline{e}); s : \mathbb{c} \mid \Delta'_1$

Let $\Delta_1 = \Delta[\overline{y \mapsto \mathbb{r}}]$: by either (TR-VAR-RECORD) or (TR-FIELD-RECORD) and (TR-AINVK), there exist $\mathbb{r} = c' \rightsquigarrow \mathbb{r}'$ (where c' is the cog of the record of e), $\imath_f$ and $\mathbb{c}_{\imath_f}$ such that $\Delta_1 \vdash_R^{c,o} e!m(\overline{e}) : \imath_f, 0 \mid \Delta_1[\imath_f \mapsto (\mathbb{r}, \mathbb{c}_{\imath_f})]$. By construction of the type system (in particular, the rules (TR-GET*) and (TR-AWAIT*)), there exists a term t such that $\mathbb{c} = t[^{\mathbb{c}_{\imath_f}}/_{\imath_f}]$ and such that $\Delta_1[f \mapsto (\mathbb{r}, f')] \vdash_R^{c,o} x = f; s : t[^{f}/_{\imath_f}] \mid \Delta'_2$ (with $\Delta'_2 \triangleq \Delta'_1 \setminus \{\imath_f\}[f \mapsto (\mathbb{r}, f')^{[\checkmark]}]$ and $[\checkmark] = \checkmark$ iff $\Delta'_1(\imath_f)$ is checked). By construction of the *rt_unsync* function, there exist a term t' such that $\mathit{rt\_unsync}(\Delta'_1) = t'[^{\mathbb{c}_{\imath_f}}/_{\imath_f}]$ and $\mathit{rt\_unsync}(\Delta'_2) = t'[^{f}/_{\imath_f}]$. Finally, if we note $\Delta' \triangleq \Delta[f \mapsto (\mathbb{r}, f')]$, we can type the invocation message with $[\mathbb{c}_{\imath_f}]_f$ (as c' is the cog of the record of `this` in $\mathbb{c}'$), we have

- $\Delta' \vdash_R \mathit{cn}' : \langle t[^{f}/_{\imath_f}], t'[^{f}/_{\imath_f}]\rangle_c^{f'} \parallel [\mathbb{c}_{\imath_f}]_f \parallel \mathbb{k}''$
- the rule (LS-AINVK) gives us that

$$\mathbb{k} \trianglerighteq \langle t[^{f}/_{\imath_f}], t'[^{f}/_{\imath_f}]\rangle_c^{f'} \parallel [\mathbb{c}_{\imath_f}]_f \parallel \mathbb{k}''$$

– *Method instantiations.*

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

By assumption and rules (TR-PARALLEL) and (R-INVOC) we have $\Delta(o) = (\mathtt{C}, \mathbb{r})$, $\Delta(f) = (c \rightsquigarrow \mathbb{r}', 0)$, $c = \mathtt{cog}(\mathbb{r})$ and $\mathbb{k} = [C!\mathtt{m}\ \mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{r}']_f \parallel \mathbb{k}'$ with $\Delta \vdash_R \mathit{invoc}(o, f, m, \overline{v}) : [\mathtt{C!m}\ \mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{r}']_f$ and $\Delta \vdash_R \mathit{ob}(o, a, p, q) : \mathbb{k}'$. Let $\overline{x}$ be the formal parameters of `m` in `C`. The auxiliary function $\mathit{bind}(o, f, m, \overline{v}, \mathtt{C})$ returns a process $\{[\text{destiny} \mapsto f, \overline{x} \mapsto \overline{v}] \mid s\}$. It is possible to demonstrate that $\Delta \vdash_R^{c,o} \{l[\text{destiny} \mapsto f, \overline{x} \mapsto \overline{v}] | s\} : \langle \mathbb{c}_{\mathtt{m}}, \mathbb{c}'_{\mathtt{m}}\rangle_c^f$, where $\Delta(\mathtt{C.m}) = \mathbb{s}(\overline{\mathbb{s}})\{\langle \mathbb{c}_0, \mathbb{c}'_0\rangle\}\mathbb{s}'$ and $\mathbb{c}_{\mathtt{m}} = \mathbb{c}_0[^{\overline{c}}/_{\overline{c'}}][^{\mathbb{r}, \overline{\mathbb{r}}}/_{\mathbb{s}, \overline{\mathbb{s}}}]$ and $\overline{c'} \in \mathbb{s}' \setminus (\mathbb{s} \cup \overline{\mathbb{s}})$ with $\overline{c}$ fresh and $\mathbb{c}'_{\mathtt{m}} = \mathbb{c}'_0[^{\overline{c}}/_{\overline{c'}}][^{\mathbb{r}, \overline{\mathbb{r}}}/_{\mathbb{s}, \overline{\mathbb{s}}}]$.

By rules (TR-PROCESS) and (TR-OBJECT), it follows that $\Delta \vdash_R \mathit{ob}(o, a, p, q \cup \{\mathit{bind}(o, f, m, \overline{v}, \mathtt{C})\}) : \mathbb{k}' \| \langle \mathbb{c}_{\mathtt{m}}, \mathbb{c}'_{\mathtt{m}}\rangle_c^f$. Moreover, by applying the rule (LS-BIND), we have that $[\mathtt{C!m}\ \mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{r}']_f \trianglerighteq \langle \mathbb{c}_{\mathtt{m}}, \mathbb{c}'_{\mathtt{m}}\rangle_c^f$ which implies with the rule (LS-GLOBAL) that $\mathbb{k} \trianglerighteq \mathbb{k}'$.

– *Getting the value of a future.*

$$\text{(READ-FUT)}\quad \frac{f = [\![e]\!]_{(a+l)} \quad v \neq \bot}{\begin{array}{c}\mathit{ob}(o, a, \{l \mid x = e.\mathtt{get}; s\}, q)\ \mathit{fut}(f, v) \to \\ \mathit{ob}(o, a, \{l \mid x = v; s\}, q)\ \mathit{fut}(f, v)\end{array}}$$

By assumption and rules (TR-PARALLEL), (TR-OBJECT) and (TR-FUTURE-TICK), there exists Δ'', $\mathbb{c}$, $\mathbb{k}''$ such that (let f' =l[destiny])

- $\Delta \vdash_R^{\mathtt{cog}(o),o} \{l | x = e.\mathtt{get}; s\} : \langle \mathbb{c}, \mathit{rt\_unsync}(\Delta'', f')\rangle_{\mathtt{cog}(o)}^{f'}$
- $\Delta \vdash_R^{\mathtt{cog}(o),o} q : \mathbb{k}''$
- $\Delta \vdash_R \mathit{fut}(f, v) : 0$
- $\mathbb{k} = \langle \mathbb{c}, \mathit{rt\_unsync}(\Delta'', f')\rangle_{\mathtt{cog}(o)}^{f'} \parallel \mathbb{k}''$, and
- $[\![e]\!]_{a \circ l} = f$.

Moreover, as $\mathit{fut}(f, v)$ is typable and contains a value, we know that $\mathbb{c} = 0 \,⨟\, \mathbb{c}'$ ($e.\mathtt{get}$ has contract 0). With the rule (TR-PURE), have that $\Delta \vdash_R^{\mathtt{cog}(o),o} \{l | x = v; s\} : \langle \mathbb{c}, \mathit{rt\_unsync}(\Delta'', f')\rangle_{\mathtt{cog}(o)}^{f'}$, and with $\mathbb{k}' = \mathbb{k}$, we have the result.

– *Remote synchronous call.* Similar to the cases of asynchronous call with a `get`-synchronisation. The result follows, in particular, from rule (LS-RSINVK) of Figure 28.

– *Cog-local synchronous call.* Similar to case of asynchronous call. The result follows, in particular, from rules (LS-SIMPLENULL) of Figure 28 and from the Definition of $\mathtt{C.m}\ \mathbb{r}(\overline{\mathbb{r}}) \to \mathbb{s}$.

– *Local Assignment.*

$$\text{(ASSIGN-LOCAL)}\quad \frac{x \in \text{dom}(l) \quad v = [\![e]\!]_{(a+l)}}{\begin{array}{c}\mathit{ob}(o, a, \{l \mid x = e; s\}, q) \\ \to \mathit{ob}(o, a, \{l[x \mapsto v] \mid s\}, q)\end{array}}$$

By assumption and rules (TR-OBJECT), (TR-PROCESS), (TR-SEQ), (TR-VAR-RECORD) and (TR-PURE), there exists Δ'', $\mathbb{c}$, $\mathbb{k}''$ such that (we note Δ_1 for $\Delta[\overline{y : \mathbb{x}}]$ and f for l[destiny])

- $\Delta \vdash_R^{\mathtt{cog}(o),o} \{l | x = e; s\} : \langle 0 \,⨟\, \mathbb{c}, \mathit{rt\_unsync}(\Delta'', f)\rangle_{\mathtt{cog}(o)}^{f}$
- $\Delta \vdash_R^{\mathtt{cog}(o),o} q : \mathbb{k}''$

runtime expressions

(TR-Obj)
$$\frac{\Delta(o) = (\mathtt{C}, \mathbb{r})}{\Delta \vdash^{c,o}_{R} o : \mathbb{r}}$$

(TR-Fut)
$$\frac{\Delta(F) = \mathbb{z}}{\Delta \vdash^{c,o}_{R} F : \mathbb{z}}$$

(TR-Var)
$$\frac{\Delta(x) = \mathbb{x}}{\Delta \vdash^{c,o}_{R} x : \mathbb{x}}$$

(TR-Field)
$$\frac{x \notin \mathrm{dom}(\Delta) \qquad \Delta(o.x) = \mathbb{r}}{\Delta \vdash^{c,o}_{R} x : \mathbb{r}}$$

(TR-Value)
$$\frac{\Delta \vdash^{c,o}_{R} e : F \qquad \Delta \vdash^{c,o}_{R} F : (\mathbb{r}, \mathbb{c})^{[\checkmark]}}{\Delta \vdash^{c,o}_{R} e : \mathbb{r}}$$

(TR-Val)
$$\frac{e \quad \textit{primitive value or arithmetic-and-bool-exp}}{\Delta \vdash^{c,o}_{R} e : \_}$$

(TR-Pure)
$$\frac{\Delta \vdash^{c,o}_{R} e : \mathbb{r}}{\Delta \vdash^{c,o}_{R} e : \mathbb{r}, 0 \,|\, \Delta}$$

expressions with side-effects

(TR-Get)
$$\frac{\begin{array}{c}\Delta \vdash^{c,o}_{R} x : \imath_f \qquad \Delta \vdash^{c,o}_{R} \imath_f : (c' \rightsquigarrow \mathbb{r}', \mathbb{c}) \\ \Delta[\mathit{destiny}] = f \qquad \Delta' = \Delta[\imath_f \mapsto (\mathbb{r}, 0)^{\checkmark}]\end{array}}{\Delta \vdash^{c,o}_{R} x.\mathtt{get} : \mathbb{r}', \mathbb{c}.(c, c') \parallel \mathit{rt\_unsync}(\Delta', f) \,|\, \Delta'}$$

(TR-Get-Runtime)
$$\frac{\begin{array}{c}\Delta \vdash^{c,o}_{R} x : f \qquad \Delta \vdash^{c,o}_{R} f : (c' \rightsquigarrow \mathbb{r}', \mathbb{c}) \\ \Delta[\mathit{destiny}] = f' \qquad \Delta' = \Delta[f \mapsto (\mathbb{r}, 0)^{\checkmark}]\end{array}}{\Delta \vdash^{c,o}_{R} x.\mathtt{get} : \mathbb{r}', f.(c, c') \parallel \mathit{rt\_unsync}(\Delta', f') \,|\, \Delta'}$$

(TR-Get-tick)
$$\frac{\Delta \vdash^{c,o}_{R} x : F \qquad \Delta \vdash^{c,o}_{R} F : (c' \rightsquigarrow \mathbb{r}', \mathbb{c})^{\checkmark}}{\Delta \vdash^{c,o}_{R} x.\mathtt{get} : \mathbb{r}', 0 \,|\, \Delta}$$

(TR-NewCog)
$$\frac{\Delta \vdash^{c,o}_{R} \overline{e} : \overline{\mathbb{r}} \qquad \mathit{param}(\mathtt{C}) = \overline{T\ x} \qquad \mathit{fields}(\mathtt{C}) = \overline{T'\ x'} \qquad c' \text{ fresh}}{\Delta \vdash^{c,o}_{R} \mathtt{new\ cog\ C}(\overline{e}) : [\mathit{cog}{:}c', \overline{x{:}\mathbb{r}}, \overline{x'}{:}\overline{\mathbb{r}'}], 0 \,|\, \Delta}$$

(TR-New)
$$\frac{\Delta \vdash^{c,o}_{R} \overline{e} : \overline{\mathbb{r}} \qquad \mathit{param}(\mathtt{C}) = \overline{T\ x} \qquad \mathit{fields}(\mathtt{C}) = \overline{T'\ x'}}{\Delta \vdash^{c,o}_{R} \mathtt{new\ C}(\overline{e}) : [\mathit{cog}{:}c, \overline{x{:}\mathbb{r}}, \overline{x'}{:}\overline{\mathbb{r}'}], 0 \,|\, \Delta}$$

(TR-AInvk)
$$\frac{\begin{array}{c}\Delta \vdash^{c,o}_{R} e : [\mathit{cog}{:}c', \overline{x{:}\mathbb{r}}] \qquad \mathit{class}(\mathit{types}(e)) = \mathtt{C} \qquad \Delta \vdash^{c,o}_{R} \overline{e} : \overline{\mathbb{s}} \qquad \mathit{fields}(\mathtt{C}), \mathit{param}(\mathtt{C}) = \overline{T\ x} \\ \Delta(\mathtt{C.m}) = \mathbb{r}'(\overline{\mathbb{s}'})\{\langle \mathbb{c}, \mathbb{c}' \rangle\}\mathbb{r}'' \qquad \overline{c'} = \mathit{cog\_names}(\mathbb{r}'') \setminus \mathit{cog\_names}(\mathbb{r}', \overline{\mathbb{s}'}) \qquad \overline{c}, \imath_f \text{ fresh} \qquad \mathbb{s}'' = \mathbb{r}''[^{\overline{c}}/_{\overline{c'}}][^{\mathbb{r}, \overline{\mathbb{s}}}/_{\mathbb{r}', \overline{\mathbb{s}'}}]\end{array}}{\Delta \vdash^{c,o}_{R} e!\mathtt{m}(\overline{e}) : \imath_f, 0 \,|\, \Delta[\imath_f \mapsto (c' \rightsquigarrow \mathbb{s}'', \mathtt{C!m}\ \mathbb{r}(\overline{\mathbb{s}}) \to \mathbb{s}'')]}$$

(TR-SInvk)
$$\frac{\begin{array}{c}\Delta \vdash^{c,o}_{R} e : [\mathit{cog}{:}c', \overline{x{:}\mathbb{r}}] \qquad \mathit{class}(\mathit{types}(e)) = \mathtt{C} \qquad \Delta \vdash^{c,o}_{R} \overline{e} : \overline{\mathbb{s}} \qquad \mathit{fields}(\mathtt{C}), \mathit{param}(\mathtt{C}) = \overline{T\ x} \\ \Delta(\mathtt{C.m}) = \mathbb{r}'(\overline{\mathbb{s}'})\{\langle \mathbb{c}, \mathbb{c}' \rangle\}\mathbb{r}'' \qquad \overline{c'} = \mathit{cog\_names}(\mathbb{r}'') \setminus \mathit{cog\_names}(\mathbb{r}', \overline{\mathbb{s}'}) \qquad \overline{c} \text{ fresh} \qquad \mathbb{s}'' = \mathbb{r}''[^{\overline{c}}/_{\overline{c'}}][^{\mathbb{r}, \overline{\mathbb{s}}}/_{\mathbb{r}', \overline{\mathbb{s}'}}]\end{array}}{\Delta \vdash^{c,o}_{R} e.\mathtt{m}(\overline{e}) : \mathbb{s}'', \mathtt{C.m}\ \mathbb{r}(\overline{\mathbb{s}}) \to \mathbb{s}'' \parallel \mathit{rt\_unsync}(\Delta) \,|\, \Delta}$$

Fig. 26 Runtime typing rules for expressions

– $\mathbb{k} = \langle \mathbb{c}, \mathit{rt\_unsync}(\Delta'', f)\rangle^{f}_{\mathtt{cog}(o)} \parallel \mathbb{k}''$, and
– $[\![e]\!]_{a \circ l} = v$.

We have

$$\Delta \vdash^{\mathtt{cog}(o),o}_{R} \{l[x \mapsto [\![e]\!]_{(a+l)}]|s\} : \langle \mathbb{c}, \mathit{rt\_unsync}(\Delta'', f)\rangle^{f}_{\mathtt{cog}(o)}$$

which gives us the result with $\mathbb{k}' = \langle \mathbb{c}, \mathit{rt\_unsync}(\Delta'', f)\rangle^{\mathbb{c}}_{f} \parallel \mathbb{k}'$.

□

B Properties of Section 5

In this section, we will prove that the statements given in Section 5 are correct, i.e. that the fixpoint analysis does detect deadlocks. To prove that statement, we first need to define the dependencies generated by the runtime contract of a running program. Then, our proof works in three steps: i) first, we show that our analysis (performed at static time) contains all the dependencies of the runtime contract of the program; ii) second that the dependencies in a program at runtime are contained in the dependencies of its runtime contract; and finally iii) when *cn* (typed with $\mathbb{k}$) reduces to *cn*′ (typed with $\mathbb{k}'$), we prove that the dependencies of $\mathbb{k}'$ are contained in $\mathbb{k}$. Basically, we prove that the following diagram holds:

$$\begin{tikzcd}
\mathrm{P} \arrow[rr] \arrow[d, dashed] & & cn_1 \arrow[dl, dashed] \arrow[dr, dashed] \arrow[rr, dash] & & \cdots \arrow[rr] & & cn_n \arrow[dl, dashed] \arrow[dr, dashed] & \\
\langle \mathbb{c}, \mathbb{c}' \rangle \arrow[d] & \mathbb{k}_1 \arrow[d] & & A_1 \arrow[dll, phantom, "\Supset" sloped] & \cdots & \mathbb{k}_n \arrow[d] & & A_n \arrow[dll, phantom, "\Supset" sloped] \\
\langle \mathcal{L}, \mathcal{L}' \rangle \arrow[r, phantom, "\Supset"] & \langle \mathcal{L}_1, \mathcal{L}'_1 \rangle \arrow[rrr, phantom, "\Supset"] & & & \cdots \arrow[r, phantom, "\Supset"] & \langle \mathcal{L}_n, \mathcal{L}'_n \rangle & &
\end{tikzcd}$$

Hence, the analysis $\langle \mathcal{L}, \mathcal{L}' \rangle$ contains all the dependencies A_i that the program can have at runtime, and thus, if the program has a deadlock, the analysis would have a circularity.

In the following, we introduce how we compute the dependencies of a runtime contract. This computation is difficult in general, but in case the runtime contract is as we constructed it in the subject-reduction theorem, then the definition is very simple. First, let say that a contract $\mathbb{c}$ that does not contain any future is *closed*. It is clear that we can compute $\mathbb{c}(\textsc{act}_{[n]})$ when $\mathbb{c}$ is closed.

Proposition 5 *Let $\Delta \vdash cn : \mathbb{k}$ be a typing derivation constructed as in the proof of Theorem 4. Then $\mathbb{k}$ is well formed and $(\!|\mathbb{k}|\!) = \langle \mathbb{c}, \mathbb{c}' \rangle^{\mathtt{start}}_{f_{start}}$ where $\mathbb{c}$ and $\mathbb{c}'$ are closed.*

Proof The first property is already stated in Lemma 1. The second property comes from the fact that when we create a new future f (in the *Asynchronous calls* case for instance), we statements

(TR-Var-Record)
$$\frac{\Delta \vdash_R^{c,o} x : \mathbb{x} \qquad \Delta \vdash_R^{c,o} z : \mathbb{x}', \mathbb{c} \mid \Delta'}{\Delta \vdash_R^{c,o} x = z : \mathbb{c} \mid \Delta'[x \mapsto \mathbb{x}']}$$

(TR-Field-Record)
$$\frac{x \notin \mathrm{dom}(\Delta) \qquad \Delta(\mathtt{this}.x) = \mathbb{r} \qquad \Delta \vdash_R^{c,o} z : \mathbb{r}, \mathbb{c} \mid \Delta'}{\Delta \vdash_R^{c,o} x = z : \mathbb{c} \mid \Delta'}$$

(TR-Var-Future)
$$\frac{\Delta \vdash_R^{c,o} x : F}{\Delta \vdash_R^{c,o} x = f : 0 \mid \Delta[x \mapsto f]}$$

(TR-Await)
$$\frac{\begin{array}{c}\Delta \vdash_R^{c,o} x : \imath_f \qquad \Delta \vdash_R^{c,o} \imath_f : (c' \rightsquigarrow \mathbb{r}, \mathbb{c}) \\ \Delta[\mathit{destiny}] = f \qquad \Delta' = \Delta[\imath_f \mapsto (c' \rightsquigarrow \mathbb{r}, 0)^{\checkmark}]\end{array}}{\Delta \vdash_R^{c,o} \mathtt{await}\ x? : \mathbb{c}.(c, c')^{\mathtt{w}} \parallel \mathit{rt\_unsync}(\Delta', f) \mid \Delta'}$$

(TR-Await-Runtime)
$$\frac{\begin{array}{c}\Delta \vdash_R^{c,o} x : f \qquad \Delta \vdash_R^{c,o} f : (c' \rightsquigarrow \mathbb{r}, \mathbb{c}) \\ \Delta[\mathit{destiny}] = f' \qquad \Delta' = \Delta[f \mapsto (c' \rightsquigarrow \mathbb{r}, 0)^{\checkmark}]\end{array}}{\Delta \vdash_R^{c,o} \mathtt{await}\ x? : f.(c, c')^{\mathtt{w}} \parallel \mathit{rt\_unsync}(\Delta', f') \mid \Delta'}$$

(TR-Await-Tick)
$$\frac{\Delta \vdash_R^{c,o} x : F \qquad \Delta \vdash_R^{c,o} F : (c' \rightsquigarrow \mathbb{r}, \mathbb{c})^{\checkmark}}{\Delta \vdash_R^{c,o} \mathtt{await}\ x? : 0 \mid \Delta}$$

(TR-If)
$$\frac{\begin{array}{c}\Delta \vdash_R^{c,o} e : \mathtt{Bool} \qquad \Delta \vdash_R^{c,o} s_1 : \mathbb{c}_1 \mid \Delta_1 \qquad \Delta \vdash_R^{c,o} s_2 : \mathbb{c}_2 \mid \Delta_2 \\ x \in \mathrm{dom}(\Delta) \Longrightarrow \Delta_1(x) = \Delta_2(x) \\ x \in \mathtt{Fut}(\Delta) \Longrightarrow \Delta_1(\Delta_1(x)) = \Delta_2(\Delta_2(x)) \\ \Delta' = \Delta_1 + (\Delta_2 \setminus (\mathrm{dom}(\Delta) \cup \{\Delta_2(x) \mid x \in \mathtt{Fut}(\Delta_2)\}))\end{array}}{\Delta \vdash_R^{c,o} \mathtt{if}\ e\ \{\, s_1 \,\}\ \mathtt{else}\ \{\, s_2 \,\} : \mathbb{c}_1 + \mathbb{c}_2 \mid \Delta'}$$

(TR-Skip)
$$\Delta \vdash_R^{c,o} \mathtt{skip} : 0 \mid \Delta$$

(TR-Seq)
$$\frac{\begin{array}{c}\Delta \vdash_R^{c,o} s_1 : \mathbb{c}_1 \mid \Delta_1 \\ \Delta_1 \vdash_R^{c,o} s_2 : \mathbb{c}_2 \mid \Delta_2\end{array}}{\Delta \vdash_R^{c,o} s_1; s_2 : \mathbb{c}_1 \fatsemi \mathbb{c}_2 \mid \Delta_2}$$

(TR-Return)
$$\frac{\begin{array}{c}\Delta \vdash_R^{c,o} e : \mathbb{r} \\ \Delta(\mathrm{destiny}) = f \qquad \Delta(f) = (c \rightsquigarrow \mathbb{r}, \mathbb{c})\end{array}}{\Delta \vdash_R^{c,o} \mathtt{return}\ e : 0 \mid \Delta}$$

(TR-Cont)
$$\frac{\Delta(f) = \mathbb{z}}{\Delta \vdash_R^{c,o} \mathtt{cont}(f) : 0 \mid \Delta}$$

Fig. 27 Runtime typing rules for statements

map it in Δ' to its father process, which will then reference f because of the $\mathit{rt\_unsync}(\cdot)$ function. Hence, if we consider the relation of which future references which other future in $\mathbb{k}$, we get a dependency graph in the shape of a directed tree, where the root is f_{start}. So, $(\!|\mathbb{k}|\!)$ reduces to a simple pair of contract of the form $\langle \mathbb{c}, \mathbb{c}' \rangle^{\mathtt{start}}_{f_{start}}$ where $\mathbb{c}$ and $\mathbb{c}'$ are closed. □

In the following, we will suppose that all runtime contracts $\mathbb{k}$ come from a type derivation constructed as in Theorem 4.

Definition 13 The *semantics* of a closed runtime pair (unique up to remaning of cog names) for the saturation at i, noted $[\![\langle \mathbb{c}, \mathbb{c}' \rangle^{c}_{f}]\!]_n$, is defined as $[\![\langle \mathbb{c}, \mathbb{c}' \rangle^{c}_{f}]\!]_n = (\mathbb{c}(\text{ACT}_{[n]})_c) \fatsemi (\mathbb{c}'(\text{ACT}_{[n]})_c)$. We extend that definition for any runtime contract with $[\![\mathbb{k}]\!]_n \triangleq [\![(\!|\mathbb{k}|\!)]\!]_n$.

Now that we can compute the dependencies of a runtime contract, we can prove our first property: the analysis performed at static time contains all the dependencies of the initial runtime contract of the program (note that $\sigma(\mathbb{c})(\text{ACT}_{[n]}) \fatsemi \sigma(\mathbb{c}')(\text{ACT}_{[n]})$ is the analysis performed at static time, and $[\![\sigma(\langle \mathbb{c}, \mathbb{c}' \rangle^{\mathtt{start}}_{f_{start}})]\!]_n$ is the set of dependencies of the initial runtime contract of the program):

Proposition 6 *Let $P = \overline{I}\ \overline{C}\ \{\overline{T\ x}; s\}$ be a* core ABS *program and let $\Gamma \vdash P$:* CCT*, $\langle \mathbb{c}, \mathbb{c}' \rangle \rhd \mathcal{U}$. Let also σ be a substitution satisfying $\mathcal{U}$. Then we have that $[\![\sigma(\langle \mathbb{c}, \mathbb{c}' \rangle^{\mathtt{start}}_{f_{start}})]\!]_n \Subset \sigma(\mathbb{c})(\text{ACT}_{[n]}) \fatsemi \sigma(\mathbb{c}')(\text{ACT}_{[n]})$.*

Proof The result is direct with an induction on $\mathbb{c}$ and $\mathbb{c}'$, and with the fact that $+$, $\fatsemi$ and $\parallel$ are monotone with respect to $\Subset$. □

We now prove the second property: all the dependencies of a program at a given time is included in the dependencies generated from its contract.

Proposition 7 *Let suppose $\Delta \vdash_R cn : \mathbb{k}$ and let A be the set of dependencies of cn. Then, with $[\![\mathbb{k}]\!]_n = \langle \mathcal{L}, \mathcal{L}' \rangle$, we have $A \subset \mathcal{L}$.*

Proof By Definition 2, if cn has a dependency (c, c'), then there exist $cn_1 = ob(o, a, \{l | x = e.\mathtt{get}; s\}, q) \in cn$, $cn_2 = fut(f, \bot) \in cn$ and $cn_3 = ob(o', a', p', q') \in cn$ such that $[\![e]\!]_{(a+l)} = l'(\text{destiny}) = f$, $\{l' \mid s'\} \in p' \cup q'$ and $a(cog) = c$ and $a'(cog) = c'$. By runtime typing rules (TR-Object), (TR-Process), (TR-Seq) and (TR-Get-Runtime), the contract of cn_1 is

$$\langle f.(c, c') \fatsemi \mathbb{c}_s, \mathbb{c}'_s \rangle^{a(\mathtt{cog})}_{l(destiny)} \parallel \mathbb{k}_q$$

we indeed know that the dependency in the contract is toward c' because of (TR-Invoc) or (TR-Process). Hence $\mathbb{k} = \langle f.(c, c') \fatsemi \mathbb{c}_s, \mathbb{c}'_s \rangle^{a(\mathtt{cog})}_{l(destiny)} \parallel \mathbb{k}'$. It follows, with the lam transformation rule (L-GAInvk), that (c, c') is in $\mathcal{L}$. □

Proposition 8 *Given two runtime contracts $\mathbb{k}$ and $\mathbb{k}'$ with $\mathbb{k} \trianglerighteq \mathbb{k}'$, we have that $[\![\mathbb{k}']\!]_n \Subset [\![\mathbb{k}]\!]_n$.*

Proof We refer to the rules (LS-*) of the later-stage relation defined in Figure 28 and to the lam transformation rules (L-*) defined in Figure 16 . The result is clear for the rules (LS-Global), (LS-Fut), (LS-Empty), (LS-Delete) and (LS-Plus). The result for the rule (LS-Bind) is a consequence of (L-AInvk). The result for the rule (LS-AInvk) is a consequence of the definition of $\Rightarrow$. The result for the rule (LS-SInvk) is a consequence of the definition of $\Rightarrow$ and (L-SInvk). The result for the rule (LS-RSInvk) is a consequence of the

the substitution process

$$\begin{aligned}
[\text{-}/\_] &\overset{def}{=} \varepsilon \\
[{}^{\mathbb{r}}/X] &\overset{def}{=} [{}^{\mathbb{r}}/X] \\
[{[cog{:}c', x_1{:}\mathbb{r}'_1, \cdots, x_n{:}\mathbb{r}'_n]}/{[cog{:}c, x_1{:}\mathbb{r}_1, \cdots, x_n{:}\mathbb{r}_n]}] &\overset{def}{=} [{}^{c'}/{}_{c}]\,[{}^{\mathbb{r}'_1}/{}_{\mathbb{r}_1}] \cdots [{}^{\mathbb{r}'_n}/{}_{\mathbb{r}_n}] \\
[{}^{c' \rightsquigarrow \mathbb{r}'}/{}_{c \rightsquigarrow \mathbb{r}}] &\overset{def}{=} [{}^{c'}/{}_{c}]\,[{}^{\mathbb{r}'}/{}_{\mathbb{r}}]
\end{aligned}$$

the later-stage relation is the least congruence with respect to runtime contracts that contains the rules

$$\textsc{LS-Global}\quad \frac{\mathbb{k}_1 \trianglerighteq \mathbb{k}'_1 \qquad \mathbb{k}_2 \trianglerighteq \mathbb{k}'_2}{\mathbb{k}_1 \parallel \mathbb{k}_2 \trianglerighteq \mathbb{k}'_1 \parallel \mathbb{k}'_2}$$

$$\textsc{LS-Bind}\quad \frac{\begin{array}{cc}\Delta(\mathtt{C.m}) = \mathbb{r}_{\mathtt{this}}\ (\overline{\mathbb{r}_{\mathtt{this}}})\ \{\langle \mathbb{c}, \mathbb{c}' \rangle\}\ \mathbb{r}'_{\mathtt{this}} & \overline{c} = fn(\langle \mathbb{c}, \mathbb{c}' \rangle) \setminus fn(\mathbb{r}_{\mathtt{this}}, \overline{\mathbb{r}_{\mathtt{this}}}, \mathbb{r}'_{\mathtt{this}}) \\ \mathbb{r}_{\mathtt{p}} = [cog : c, \overline{x{:}\mathbb{r}}] & \overline{c'} \cap fn(\mathbb{r}_{\mathtt{p}}, \overline{\mathbb{r}_{\mathtt{p}}}, \mathbb{r}'_{\mathtt{p}}) = \varnothing\end{array}}{[\mathtt{C!m}\ \mathbb{r}_{\mathtt{p}}(\overline{\mathbb{r}_{\mathtt{p}}}) \to \mathbb{r}'_{\mathtt{p}}]_f \trianglerighteq \langle \mathbb{c}, \mathbb{c}' \rangle^{c}_{f} [{}^{\overline{c'}}/{}_{\overline{c}}][{}^{\mathbb{r}_{\mathtt{p}}, \overline{\mathbb{r}_{\mathtt{p}}}, \mathbb{r}'_{\mathtt{p}}}/{}_{\mathbb{r}_{\mathtt{this}}, \overline{\mathbb{r}_{\mathtt{this}}}, \mathbb{r}'_{\mathtt{this}}}]}$$

$$\textsc{LS-AInvk}\quad \frac{f' \in fn(\langle \mathbb{c}, \mathbb{c}' \rangle)}{\langle \mathbb{c}, \mathbb{c}' \rangle^{c}_{f} [{}^{\mathtt{C!m}\ \mathbb{r}_{\mathtt{p}}(\overline{\mathbb{r}_{\mathtt{p}}}) \to \mathbb{r}'_{\mathtt{p}}}/{}_{f'}] \trianglerighteq \langle \mathbb{c}, \mathbb{c}' \rangle^{c}_{f} \parallel [\mathtt{C!m}\ \mathbb{r}_{\mathtt{p}}(\overline{\mathbb{r}_{\mathtt{p}}}) \to \mathbb{r}'_{\mathtt{p}}]_{f'}}$$

$$\textsc{LS-SInvk}\quad \frac{f' \in fn(\langle \mathbb{c}, \mathbb{c}' \rangle) \qquad \mathbb{r}_{\mathtt{p}} = [cog : c, \overline{x{:}\mathbb{r}}]}{\langle (\mathtt{C.m}\ \mathbb{r}(\overline{\mathbb{s}}) \to \mathbb{r}' \parallel \mathbb{c}) \fatsemi \mathbb{c}', \mathbb{c}'' \rangle^{c}_{f} \trianglerighteq \langle (f'.(c,c)^{\mathtt{w}} \parallel \mathbb{c}) \fatsemi \mathbb{c}', \mathbb{c}'' \rangle^{c}_{f} \parallel [\mathtt{C!m}\ \mathbb{r}_{\mathtt{p}}(\overline{\mathbb{r}_{\mathtt{p}}}) \to \mathbb{r}'_{\mathtt{p}}]_{f'}}$$

$$\textsc{LS-RSInvk}\quad \frac{f' \in fn(\langle \mathbb{c}, \mathbb{c}' \rangle) \qquad \mathbb{r}_{\mathtt{p}} = [cog : c', \overline{x{:}\mathbb{r}}] \qquad c' \neq c}{\langle (\mathtt{C.m}\ \mathbb{r}(\overline{\mathbb{s}}) \to \mathbb{r}' \parallel \mathbb{c}) \fatsemi \mathbb{c}', \mathbb{c}'' \rangle^{c}_{f} \trianglerighteq \langle (f'.(c,c') \parallel \mathbb{c}) \fatsemi \mathbb{c}', \mathbb{c}'' \rangle^{c}_{f} \parallel [\mathtt{C!m}\ \mathbb{r}_{\mathtt{p}}(\overline{\mathbb{r}_{\mathtt{p}}}) \to \mathbb{r}'_{\mathtt{p}}]_{f'}}$$

$$\textsc{LS-DepNull}\quad \langle \mathbb{c}, \mathbb{c}' \rangle^{c}_{f} \parallel \langle 0, 0 \rangle^{c'}_{f'} \trianglerighteq \langle \mathbb{c}[{}^{0}/{}_{f'}], \mathbb{c}'[{}^{0}/{}_{f'}] \rangle^{c}_{f} \parallel \langle 0, 0 \rangle^{c'}_{f'}$$

$$\textsc{LS-Fut}\ \ f \trianglerighteq 0 \qquad \textsc{LS-Empty}\ \ 0.(c,c')^{[\mathtt{w}]} \trianglerighteq 0 \qquad \textsc{LS-Delete}\ \ 0 \fatsemi \mathbb{c} \trianglerighteq \mathbb{c} \qquad \textsc{LS-Plus}\ \ \mathbb{c}_1 + \mathbb{c}_2 \trianglerighteq \mathbb{c}_i$$

Fig. 28 The later-stage relation

definition of $\Rightarrow$ and (L-RSInvk). Finally, the result for the rule (LS-DepNull) is a consequence of the definition of $\Rightarrow$. □

We can finally conclude by putting all these results together:

Theorem 5 *If a program P has a deadlock at runtime, then its* abstract semantics saturated at n *contains a circle.*

Proof This property is a direct consequence of Propositions 6, 7 and 8. □

C Properties of Section 6

The next theorem states the correctness of our model-checking technique.

Below we write $[\![cn]\!]_{[n]} = ([\![\langle \mathbb{cp}_n \boldsymbol{,} \mathbb{cp}'_n \rangle_{\text{start}}]\!])^{\flat}$, if $\Delta \vdash_R cn : \langle \mathbb{cp}_1 \boldsymbol{,} \mathbb{cp}'_1 \rangle$ and n is the order of $\langle \mathbb{cp}_1 \boldsymbol{,} \mathbb{cp}'_1 \rangle_{\text{start}}$.

Theorem 6 *Let* $(\text{CT}, \{\overline{T\ x\ ;\ s}\}, \text{CCT})$ *be a* `core ABS` *program and cn be a configuration of its operational semantics.*

1. *If cn has a circularity, then a circularity occurs in* $[\![cn]\!]_{[n]}$*;*
2. *if* $cn \to cn'$ *and* $[\![cn']\!]_{[n]}$ *has a circularity, then a circularity is already present in* $[\![cn]\!]_{[n]}$*;*
3. *let* $\imath$ *be an injective renaming of cog names;* $[\![cn]\!]_{[n]}$ *has a circularity if and only if* $[\![\imath(cn)]\!]_{[n]}$ *has a circularity.*

Proof To demonstrate item 1, let

$$[\![cn]\!]_{[n]} = ([\![\langle \mathbb{cp}_n \boldsymbol{,} \mathbb{cp}'_n \rangle_{\text{start}}]\!])^{\flat}\ .$$

We prove that every dependencies occurring in cn is also contained in one state of $([\![\langle \mathbb{cp}_n \boldsymbol{,} \mathbb{cp}'_n \rangle_{\text{start}}]\!])^{\flat}$. By Definition 2, if cn has a dependency (c, c') then it contains $cn'' = ob(o, a, \{l | x = e.\mathtt{get}; s\}, q)\ \ fut(f, \bot)$, where $f = [\![e]\!]_{(a+l)}$, $a(cog) = c$ and there is $ob(o', a', \{l'|s'\}, q') \in cn$ such that $a'(cog) = c'$ and $l'(\text{destiny}) = f$. By the typing rules, the contract of cn' is $f.(c,c') \fatsemi \mathbb{c}_s$, where, by typing rule (T-Configurations), f is actually replaced by a $\mathtt{C!m}\,\mathbb{r}(\overline{\mathbb{s}}) \to \mathbb{s}$ produced by a concurrent *invoc* configuration, or by the contract pair $\langle \mathbb{c}_{\mathtt{m}} \boldsymbol{,} \mathbb{c}'_{\mathtt{m}} \rangle$ corresponding to the method body.

As a consequence $[\![cn'']\!]_{[n]} = ([\![\mathfrak{C}[\langle \mathbb{c}'' \& (c,c') \boldsymbol{,} \mathbb{c}''' \rangle_c]_{c''}]\!])^{\flat}$.

Let $[\![ob(o', a', \{l'|s'\}, q')]\!]_{[n]} = ([\![\mathfrak{C}'[\langle \mathbb{c}_{\mathtt{m}} \boldsymbol{,} \mathbb{c}'_{\mathtt{m}} \rangle_c]_{c''}]\!])^{\flat}$, with $[\![e]\!]_{(a+l)} = l'(\text{destiny})$, then

$$\begin{aligned}&[\![ob(o', a', \{l'|s'\}, q')\quad cn'']\!]_{[n]} = \\ &\qquad ([\![\mathfrak{C}[\langle \mathbb{c}'' \& (c,c') \boldsymbol{,} \mathbb{c}''' \rangle_c]_{c''} \parallel \mathfrak{C}'[\langle \mathbb{c}_{\mathtt{m}} \boldsymbol{,} \mathbb{c}'_{\mathtt{m}} \rangle_{c'}]_{c'''}]\!])^{\flat}\ .\end{aligned}$$

In general, if k dependencies occur in a state cn, then there is $cn'' \subseteq cn$ that collects all the tasks manifesting the dependencies.

$$\begin{aligned}&[\![cn'']\!]_{[n]}\ = \\ &\quad ([\![\mathfrak{C}_1[\langle \mathbb{c}''_1 \& (c_1, c'_1) \boldsymbol{,} \mathbb{c}'''_1 \rangle_{c_1}]_{c''_1} \parallel \mathfrak{C}'_1[\langle \mathbb{c}_{\mathtt{m}_1} \boldsymbol{,} \mathbb{c}'_{\mathtt{m}_1} \rangle_{c'_1}]_{c'''_1}]\!])^{\flat} \\ &\qquad \parallel \cdots \parallel\ ([\![\mathfrak{C}_k[\langle \mathbb{c}''_k \& (c_k, c'_k) \boldsymbol{,} \mathbb{c}'''_k \rangle_{c_k}]_{c''_k} \parallel \mathfrak{C}'_k[\langle \mathbb{c}_{\mathtt{m}_k} \boldsymbol{,} \mathbb{c}'_{\mathtt{m}_k} \rangle_{c'_k}]_{c'''_k}]\!])^{\flat}\end{aligned}$$

By definition of $\|$ composition in Section 5, the initial state contains all the above pairs (c_i, c'_i).

Let us prove the item 2. We show that the transition $cn \longrightarrow cn'$ does not produce new dependencies. That is, the set of dependencies in the states of $[\![cn']\!]_{[n]}$ is equal or smaller than the set of dependencies in the states of $[\![cn]\!]_{[n]}$.

By Theorem 4, if $\Delta \vdash_R cn : \mathbb{k}$ then $\Delta' \vdash_R cn' : \mathbb{k}'$, with $\mathbb{k} \trianglerighteq \mathbb{k}'$. We refer to the rules (LS-*) of the later-stage relation defined in Figure 28 and to the contract reduction rules (Red-*) defined in Figure 19 . The result is clear for the rules (LS-Global), (LS-Fut), (LS-Empty), (LS-Delete) and (LS-Plus). The result for the rule (LS-Bind) is a consequence of (Red-AInvk). The result for the rule (LS-AInvk) is a consequence of the definition of $\Rrightarrow$. The result for the rule (LS-SInvk) is a consequence of the definition of $\Rrightarrow$ and (Red-SInvk). The result for the rule (LS-RSInvk) is a consequence of the definition of $\Rrightarrow$ and (Red-RSInvk). Finally, the result for the rule (LS-DepNull) is a consequence of the definition of $\Rrightarrow$.

Item 3 is obvious because circularities are preserved by injective renamings of cog names. $\square$