Analysis of deadlocks in object groups

Elena Giachino Cosimo Laneve

Dipartimento di Scienze dell'Informazione, Università di Bologna

Abstract. Object groups are collections of objects that perform collective work. We study a calculus with object groups and develop a technique for the deadlock analysis of such systems based on abstract descriptions of method's behaviours.

1 Introduction

Object groups are collections of objects that perform collective work. The group abstraction is an encapsulation mechanism that is convenient in distributed programming in several circumstances. For example, in order to achieve continuous availability or load balancing through replication, or for retrieval services of distributed data. In these cases, in order to keep consistencies, the group abstraction must define suitable protocols to synchronize group members. As usual with synchronization protocols, it is possible that object groups may manifest deadlocks, which are particularly hard to discover in this context because of the two encapsulation levels of the systems (the object and the group levels).

Following the practice to define lightweight fragments of languages that are sufficiently small to ease proofs of basic properties, we define an object-oriented calculus with group operations and develop a technique for the analysis of deadlocks. Our object-oriented language, called `FJg`, is an imperative version of Featherweight Java [10] with method invocations that are asynchronous and group-oriented primitives that are taken from *Creol* [11] (*cf.* the JCoBoxes [20]).

In `FJg`, objects always belong to one group that is defined when they are created. Groups consist of multiple tasks, which are the running methods of the objects therein. Tasks are cooperatively scheduled, that is there is at most one task active at each time per group and the active task explicitly returns the control in order to let other tasks progress. Tasks are created by method invocation that are *asynchronous* in `FJg`: the caller activity continues after the invocation and the called code runs on a different task that may belong to a different group. The synchronization between the caller and the called methods is performed when the result is strictly necessary [11, 24, 4]. Technically, the decoupling of method invocation and the returned value is realized using *future variables* (see [7] 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.

In a model with object groups and cooperative scheduling, a typical deadlock situation occurs when two active tasks in different groups are waiting for each other to return a value. This circular dependency may involve less or more than two tasks. For example, a case of circularity of size one is

```
Int fact(Int n){    if (n=0) then return 1 ;
                    else return n*(this!fact(n-1).get)    }
```

The above `FJg` code defines the factorial function (for the sake of the example we include primitive types `Int` and conditional into `FJg` syntax. See Section 2.1). The invocation of `this!fact(n)` deadlocks on the recursive call `this!fact(n-1)` because the caller does not explicitly release

```
class C {
     C m() { return newg C() ;} }
class D extends C {
     C n(D c) { return (c!m()).get ;} }
```

Table 1. Simple classes in FJg

the group lock. The operation get is needed in order to synchronously retrieve the value returned by the invocation.

We develop a technique for the analysis of deadlocks in FJg programs based on *contracts*. Contracts are abstract descriptions of behaviours that retain the necessary informations to detect deadlocks [13, 12]. For example, the contract of fact (assuming it belongs to the class Ops) is G(){ Ops.fact^g: G() }. This contract declares that, when fact is invoked on an object of a group G, then it will call recursively fact on an object of the same group G without releasing the control – a *group dependency*. With this contract, any invocation of fact is fated to deadlock because of the circularity between G and itself (actually this.fact(0) never deadlocks, but the above contract is not expressive enough to handle such cases).

In particular, we define an inference system for associating a contract to every method of the program and to the expression to evaluate. Then we define a simple algorithm – the dla algorithm – returning informations about group dependencies. The presence of circularities in the result of dla reveals the possible presence of deadlocked computations. Overall, our results show the possibility and the benefit of applying techniques developed for process calculi to the area of object-oriented programming.

The paper is organized as follows. Section 2 defines FJg by introducing the main ideas and presenting its syntax and operational semantics. Section 3 discusses few sample programs in FJg and the deadlocks they manifest. Section 4 defines contracts and the inference algorithm for deriving contracts of expressions and methods. Section 5 considers the problem of extracting dependencies from contracts, presents the algorithm dla, and discusses its enhancements. Section 7 surveys related works, and we give conclusions and indications of further work in Section 8.

2 The calculus FJg

In FJg a program is a collection of class definitions plus an expression to evaluate. A simple definition in FJg is the class C in Table 1. This program defines a class C with a method m. When m is invoked, a new object of class C is created and returned. A distinctive feature of FJg is that an object belongs to a unique group. In the above case, the returned object belongs to a new group – created by the operator newg. If the new object had to belong to the group of the caller method, then we would have used the standard operation new.

Method invocations in FJg are asynchronous. For example, the class D in Table 1 defines an extension of C with method n. In order to emphasize that the semantics of method invocation is not as usual, we use the exclamation mark (instead of the dot notation). In FJg, when a method is invoked, the caller continues executing *in parallel with* the callee *without releasing its own group lock*; the callee gets the control by acquiring the lock of its group when it is free. This guarantees that, at each point in time, at most one task may be active per group. The get operation in the code of n constraints the method to wait for the return value of the callee before terminating (and therefore releasing the group lock). As a consequence an expression as (newg D())!n(newg D()).get is going to complete because the called method m in the body of n belongs to a different group w.r.t. the one of n; on the contrary an expression as (newg D())!n(new D()).get is going to produce a deadlock.

In FJg, it is also possible to wait for a result without keeping the group lock. This is performed by the operation await that releases the group lock and leaves other tasks the chance to perform their activities until the value of the called method is produced. That is, x!m().await.get corresponds to a method invocation in standard object-oriented languages.

The decoupling of method invocation and the returned value is realized in FJg by using *future types*. In particular, if a method is declared to return a value of type C, then its invocations return values of type Fut(C), rather than values of type C. This means that the value is not available yet; when it will be, it is going to be of type C. The operation get takes an expression of type Fut(C) and returns C (as the reader may expect, await takes an expression of type Fut(C) and returns Fut(C)).

2.1 Syntax

The syntax of FJg uses four disjoint infinite sets of *class names*, ranged over by A, B, C, $\cdots$, *field names*, ranged over by f, g, $\cdots$, *method names*, ranged over by m, n, $\cdots$, and *parameter names*, ranged over by x, y, $\cdots$. The special name this is assumed to belong to the set of parameter names. We write $\bar{C}$ as a shorthand for $C_1, \cdots, C_n$ and similarly for the other names. We abbreviate sequences of pairs as $C_1\ f_1, \cdots, C_n\ f_n$ with $\bar{C}\ \bar{f}$.

The abstract syntax of *class declarations* CL, *method declarations* M, and *expressions* e of FJg is the following

$$\begin{array}{lcl} \mathtt{CL} & ::= & \mathtt{class\ C\ extends\ C}\ \{\bar{\mathtt{C}}\ \bar{\mathtt{f}};\ \bar{\mathtt{M}}\} \\ \mathtt{M} & ::= & \mathtt{C\ m}\ (\bar{\mathtt{C}}\ \bar{\mathtt{x}})\{\mathtt{return\ e\ ;}\ \} \\ \mathtt{e} & ::= & \mathtt{x} \mid \mathtt{this.f} \mid \mathtt{this.f} = \mathtt{e} \mid \mathtt{e!m}(\bar{\mathtt{e}}) \mid \mathtt{new\ C}(\bar{\mathtt{e}}) \mid \mathtt{e;\ e} \\ & \mid & \mathtt{newg\ C}(\bar{\mathtt{e}}) \mid \mathtt{e.get} \mid \mathtt{e.await} \end{array}$$

Sequences of field declarations $\bar{C}\ \bar{f}$, method declarations $\bar{M}$, and parameter declarations $\bar{C}\ \bar{x}$ are assumed to contain no duplicate names.

A program is a pair (CT, e), where the *class table* CT is a finite mapping from class names to class declarations CL and e is an expression. In what follows we always assume a fixed class table CT. According to the syntax, every class has a superclass declared with extends. To avoid circularities, we assume a distinguished class name Object with no field and method declarations whose definition does not appear in the class table. As usual, class C $\{\cdots\}$ abbreviates class C extends Object $\{\cdots\}$.

Let *types* T be either class names C or *futures* Fut(C). Let also *fields*(C), *mtype*(m, C), and *mbody*(m, C) [10] be the standard FJ lookup functions that are reported in Table 2. The class table satisfies the following well-formed conditions:

(*i*) Object $\notin$ *dom*(CT);
(*ii*) for every C $\in$ *dom*(CT), CT(C) = class C $\cdots$;
(*iii*) every class name occurring in CT belongs to *dom*(CT);
(*iv*) the least relation <: , called *subtyping relation*, over types T, closed by reflexivity and transitivity and containing

$$\frac{C_1 <: C_2}{\mathtt{Fut}(C_1) <: \mathtt{Fut}(C_2)} \qquad \frac{\text{CT}(C_1) = \mathtt{class}\ C_1\ \mathtt{extends}\ C_2\ \{\cdots\}}{C_1 <: C_2}$$

is antisymmetric;
(*v*) if CT(C) = class C extends D $\{\cdots\}$ and *mtype*(m, C) = $\bar{C}' \to C'$ and *mtype*(m, D) = $\bar{D}' \to D'$ then $C' <: D'$ and $\bar{D}' <: \bar{C}'$.

It is worth to remark that future types never appear in FJg programs, where types of fields and of methods are always classes. This restriction excludes either to store future values in fields or to invoke methods with future values (that later on may be

Field lookup:

$$fields(\texttt{Object}) = \bullet \qquad \frac{\textsc{ct}(\texttt{C}) = \texttt{class C extends D}\ \{\bar{\texttt{C}}\ \bar{\texttt{f}};\ \bar{\texttt{M}}\} \quad fields(\texttt{D}) = \bar{\texttt{C}}'\ \bar{\texttt{g}}}{fields(\texttt{C}) = \bar{\texttt{C}}\ \bar{\texttt{f}},\ \bar{\texttt{C}}'\ \bar{\texttt{g}}}$$

Method type lookup:

$$\frac{\textsc{ct}(\texttt{C}) = \texttt{class C extends D}\ \{\bar{\texttt{C}}\ \bar{\texttt{f}};\ \bar{\texttt{M}}\} \quad \texttt{C}'\ \texttt{m}\ (\bar{\texttt{C}}'\ \bar{\texttt{x}})\{\texttt{return e;}\ \} \in \bar{\texttt{M}}}{mtype(\texttt{m}, \texttt{C}) = \bar{\texttt{C}}' \to \texttt{C}'} \qquad \frac{\textsc{ct}(\texttt{C}) = \texttt{class C extends D}\ \{\bar{\texttt{C}}\ \bar{\texttt{f}};\ \bar{\texttt{M}}\} \quad \texttt{m} \notin \bar{\texttt{M}}}{mtype(\texttt{m}, \texttt{C}) = mtype(\texttt{m}, \texttt{D})}$$

Method body lookup:

$$\frac{\textsc{ct}(\texttt{C}) = \texttt{class C extends D}\ \{\bar{\texttt{C}}\ \bar{\texttt{f}};\ \bar{\texttt{M}}\} \quad \texttt{C}'\ \texttt{m}\ (\bar{\texttt{C}}'\ \bar{\texttt{x}})\{\texttt{return e;}\ \} \in \bar{\texttt{M}}}{mbody(\texttt{m}, \texttt{C}) = \bar{\texttt{x}}.\texttt{e}} \qquad \frac{\textsc{ct}(\texttt{C}) = \texttt{class C extends D}\ \{\bar{\texttt{C}}\ \bar{\texttt{f}};\ \bar{\texttt{M}}\} \quad \texttt{m} \notin \bar{\texttt{M}}}{mbody(\texttt{m}, \texttt{C}) = mbody(\texttt{m}, \texttt{D})}$$

Heap lookup functions:

$$\frac{\mathsf{H}(\mathsf{o}) = (\mathsf{C}, \mathsf{G}, [\bar{\texttt{f}} : \bar{\mathsf{v}}])}{class(\mathsf{H}, \mathsf{o}) = \mathsf{C} \qquad group(\mathsf{H}, \mathsf{o}) = \mathsf{G} \qquad field(\mathsf{H}, \mathsf{o}, \texttt{f}_i) = \mathsf{v}_i}$$

Table 2. Lookup auxiliary functions ($\bullet$ is the empty sequence)

2.2 Semantics

Below we use an infinite set of *object names*, ranged over by $\mathsf{o}, \mathsf{o}', \cdots$, an infinite set of *group names*, ranged over by $\mathsf{G}, \mathsf{G}', \cdots$, and an infinite set of *task names*, ranged over by $\mathsf{t}, \mathsf{t}', \cdots$. We assume that the set of group names has a distinguished element $\mathsf{Ͻ}$, associated to the expressions irrelevant for the deadlock analysis, such as `this.f`.

`FJg` has an operational semantics that is defined in terms of a transition relation $\longrightarrow$ between *configurations* $\mathsf{H} \Vdash \mathsf{S}$, where H is the *heap* and S is a set of *tasks*. The heap maps (i) objects names o to tuples $(\mathsf{C}, \mathsf{G}, [\bar{\texttt{f}} : \bar{\mathsf{v}}])$ that record their class, their group, and their fields' values; (ii) group names G to either $\bot$ or $\top$ that specify whether the group is unlocked or locked, respectively. We use the standard update operations on heaps $\mathsf{H}[\mathsf{o} \mapsto (\mathsf{C}, \mathsf{G}, [\bar{\texttt{f}} : \bar{\mathsf{v}}])]$ and $\mathsf{H}[\mathsf{G} \mapsto \mathtt{l}]$ and on fields $[\bar{\texttt{f}} : \bar{\mathsf{v}}][\texttt{f} : \mathsf{v}]$ with the usual meanings. Tasks are tuples $\mathsf{t} :^{\mathtt{l}}_{\mathsf{o}} \mathsf{e}$, where t is the task name, o is the object of the task, $\mathtt{l}$ is either $\top$ (if the task owns the group lock) or $\bot$ (if not), and e is the expression to evaluate. The superscript $\mathtt{l}$ is omitted when it is not relevant. In the semantic clauses, by abuse of the notation, the syntactic category e also addresses *values*, ranged over by v, which are either object or task names. The set of object and task names in e is returned by the function *names*(e). The same function, when applied to a set of tasks S returns the object, group, and task names in S. The operational semantics also uses

- the *heap lookup functions class*(H, o), *group*(H, o), and *field*($\mathsf{H}, \mathsf{o}, \texttt{f}_i$) that respectively return the class, the group and the values of i-th field of o in H (see Table 2);
- *evaluation contexts* E whose syntax is:

$$\begin{array}{rl} \mathsf{E} ::= & [\,]\ \mid\ \mathsf{E}!\texttt{m}(\bar{\mathsf{e}})\ \mid\ \texttt{this.f} = \mathsf{E}\ \mid\ \mathsf{o}!\texttt{m}(\bar{\mathsf{v}}, \mathsf{E}, \bar{\mathsf{e}})\ \mid\ \texttt{new C}(\bar{\mathsf{v}}, \mathsf{E}, \bar{\mathsf{e}}) \\ \mid & \texttt{newg C}(\bar{\mathsf{v}}, \mathsf{E}, \bar{\mathsf{e}})\ \mid\ \mathsf{E}.\texttt{get}\ \mid\ \mathsf{E}.\texttt{await}\ \mid\ \mathsf{E};\ \mathsf{e} \end{array}$$

The preliminary notions are all in place for the definition of the transition relation that is given in Table 3. It is worth to notice that, in every rule, a task moves if it owns the lock of its group.

(UPDATE)

$$\frac{\mathsf{H}(\mathsf{o}') = (\mathsf{C}, \mathsf{G}, [\bar{\mathtt{f}} : \bar{\mathtt{v}}])}{\mathsf{H} \Vdash \mathtt{t} :^{\top}_{\mathsf{o}} \mathsf{E}[\mathsf{o}'.\mathtt{f} = \mathtt{v}] \longrightarrow \mathsf{H}[\mathsf{o}' \mapsto (\mathsf{C}, \mathsf{G}, [\bar{\mathtt{f}} : \bar{\mathtt{v}}][\mathtt{f} : \mathtt{v}])] \Vdash \mathtt{t} :^{\top}_{\mathsf{o}} \mathsf{E}[\mathtt{v}]}$$

(INVK)

$$\frac{\mathit{class}(\mathsf{H}, \mathsf{o}') = \mathsf{C} \quad \mathit{mbody}(\mathtt{m}, \mathsf{C}) = \bar{\mathtt{x}}.\mathtt{e} \quad \mathtt{t}' \neq \mathtt{t}}{\mathsf{H} \Vdash \mathtt{t} :^{\top}_{\mathsf{o}} \mathsf{E}[\mathsf{o}'!\mathtt{m}(\bar{\mathtt{v}})] \longrightarrow \mathsf{H} \Vdash \mathtt{t} :^{\top}_{\mathsf{o}} \mathsf{E}[\mathtt{t}'],\ \ \mathtt{t}' :^{\bot}_{\mathsf{o}'} \mathtt{e}[^{\mathsf{o}'}/_{\mathtt{this}}][^{\bar{\mathtt{v}}}/_{\bar{\mathtt{x}}}]}$$

(NEW)

$$\frac{\mathit{group}(\mathsf{H}, \mathsf{o}) = \mathsf{G} \quad \mathit{fields}(\mathsf{C}) = \bar{\mathsf{T}}\ \bar{\mathtt{f}} \qquad \mathsf{o}' \notin \mathrm{dom}(\mathsf{H}) \qquad \mathsf{H}' = \mathsf{H}[\mathsf{o}' \mapsto (\mathsf{C}, \mathsf{G}, [\bar{\mathtt{f}} : \bar{\mathtt{v}}])]}{\mathsf{H} \Vdash \mathtt{t} :^{\top}_{\mathsf{o}} \mathsf{E}[\mathtt{new}\ \mathsf{C}(\bar{\mathtt{v}})] \longrightarrow \mathsf{H}' \Vdash \mathtt{t} :^{\top}_{\mathsf{o}} \mathsf{E}[\mathsf{o}']}$$

(NEWG)

$$\frac{\mathit{fields}(\mathsf{C}) = \bar{\mathsf{T}}\ \bar{\mathtt{f}} \quad \mathsf{o}', \mathsf{G}' \notin \mathrm{dom}(\mathsf{H}) \qquad \mathsf{H}' = \mathsf{H}[\mathsf{o}' \mapsto (\mathsf{C}, \mathsf{G}', [\bar{\mathtt{f}} : \bar{\mathtt{v}}])][\mathsf{G}' \mapsto \bot]}{\mathsf{H} \Vdash \mathtt{t} :^{\top}_{\mathsf{o}} \mathsf{E}[\mathtt{newg}\ \mathsf{C}(\bar{\mathtt{v}})] \longrightarrow \mathsf{H}' \Vdash \mathtt{t} :^{\top}_{\mathsf{o}} \mathsf{E}[\mathsf{o}']}$$

(GET)

$$\mathsf{H} \Vdash \mathtt{t} :^{\top}_{\mathsf{o}} \mathsf{E}[\mathtt{t}'.\mathtt{get}],\ \mathtt{t}' :_{\mathsf{o}'} \mathtt{v} \longrightarrow \mathsf{H} \Vdash \mathtt{t} :^{\top}_{\mathsf{o}} \mathsf{E}[\mathtt{v}],\ \mathtt{t}' :_{\mathsf{o}'} \mathtt{v}$$

(AWAITT)

$$\mathsf{H} \Vdash \mathtt{t} :^{\top}_{\mathsf{o}} \mathsf{E}[\mathtt{t}'.\mathtt{await}],\ \mathtt{t}' :_{\mathsf{o}'} \mathtt{v} \longrightarrow \mathsf{H} \Vdash \mathtt{t} :^{\top}_{\mathsf{o}} \mathsf{E}[\mathtt{t}'],\ \mathtt{t}' :_{\mathsf{o}'} \mathtt{v}$$

(AWAITF)

$$\frac{\mathit{group}(\mathsf{H}, \mathsf{o}) = \mathsf{G}}{\mathsf{H}[\mathsf{G} \mapsto \top] \Vdash \mathtt{t} :^{\top}_{\mathsf{o}} \mathsf{E}[\mathtt{t}'.\mathtt{await}] \longrightarrow \mathsf{H}[\mathsf{G} \mapsto \bot] \Vdash \mathtt{t} :^{\bot}_{\mathsf{o}} \mathsf{E}[\mathtt{t}'.\mathtt{await}]}$$

(LOCK)

$$\frac{\mathit{group}(\mathsf{H}, \mathsf{o}) = \mathsf{G} \quad \mathtt{e} \neq \mathtt{v}}{\mathsf{H}[\mathsf{G} \mapsto \bot] \Vdash \mathtt{t} :^{\bot}_{\mathsf{o}} \mathtt{e} \longrightarrow \mathsf{H}[\mathsf{G} \mapsto \top] \Vdash \mathtt{t} :^{\top}_{\mathsf{o}} \mathtt{e}}$$

(RELEASE)

$$\frac{\mathit{group}(\mathsf{H}, \mathsf{o}) = \mathsf{G}}{\mathsf{H}[\mathsf{G} \mapsto \top] \Vdash \mathtt{t} :^{\top}_{\mathsf{o}} \mathtt{v} \longrightarrow \mathsf{H}[\mathsf{G} \mapsto \bot] \Vdash \mathtt{t} :^{\bot}_{\mathsf{o}} \mathtt{v}}$$

(SEQ)

$$\mathsf{H} \Vdash \mathtt{t} :^{\top}_{\mathsf{o}} \mathtt{v}; \mathtt{e} \longrightarrow \mathsf{H} \Vdash \mathtt{t} :^{\top}_{\mathsf{o}} \mathtt{e}$$

(CONFIG)

$$\frac{\mathsf{H} \Vdash \mathsf{S} \longrightarrow \mathsf{H}' \Vdash \mathsf{S}' \qquad (\mathit{names}(\mathsf{S}') \setminus \mathit{names}(\mathsf{S})) \cap \mathit{names}(\mathsf{S}'') = \emptyset}{\mathsf{H} \Vdash \mathsf{S}, \mathsf{S}'' \longrightarrow \mathsf{H}' \Vdash \mathsf{S}', \mathsf{S}''}$$

Table 3. The transition relation of FJg.

Apart this detail, the operations of update, object creation, and sequence are standard. We therefore focus on the operations about groups and futures. Rule (INVK) defines asynchronous method invocation, therefore the evaluation produces a future reference $\mathtt{t}'$ to the returned value, which may be retrieved by a `get` operation if needed. Rule (NEWG) defines $\mathtt{newg}\ \mathsf{C}(\bar{\mathtt{v}})$, which creates a new group G' and a new object o' of that group with fields initialized to $\bar{\mathtt{v}}$. This object is returned and the group G' is unlocked – no task of its is running. It will be locked as soon as a method of o' is invoked – see (LOCK). Rule (RELEASE) models method termination that amounts to store the returned value in the configuration and releasing the group lock. Rule (GET) allows one to retrieve the value returned by a method. Rules (AWAITT) and (AWAITF) model the `await` operation: if the task $\mathtt{t}'$ is terminated – it is paired to a value in the configuration – then `await` is unblocking; otherwise the group lock is released and the task $\mathtt{t}$ is blocked. Rule (CONFIG) has standard premises for avoiding unwanted name matches when lifting local reductions to complete configurations.

The initial configuration of a program $(\textsc{ct}, \mathtt{e})$ is $\mathsf{H} \Vdash \mathtt{t} :^{\top}_{\mathsf{o}} \mathtt{e}[^{\mathsf{o}}/_{\mathtt{this}}]$ where $\mathsf{H} = \mathsf{o} \mapsto (\mathtt{Object}, \mathsf{G}, [\ ]), \mathsf{G} \mapsto \top$ (following our previous agreement, the class table is implicit).

Example 1. As an example, we detail the evaluation of the expression `(newg D())!n(newg D()).get`, where the class `D` is defined in Table 1.

$$
\begin{array}{lll}
\mathsf{H} \Vdash \mathtt{t} :^{\top}_{\mathtt{o}} \mathtt{(newg\ D())!n(newg\ D()).get} & & \\
\longrightarrow \mathsf{H}_1 \Vdash \mathtt{t} :^{\top}_{\mathtt{o}} \mathtt{o1!n(newg\ D()).get} & & (1)\\
\longrightarrow \mathsf{H}_2 \Vdash \mathtt{t} :^{\top}_{\mathtt{o}} \mathtt{o1!n(o2).get} & & (2)\\
\longrightarrow \mathsf{H}_2 \Vdash \mathtt{t} :^{\top}_{\mathtt{o}} \mathtt{t1.get}, \mathtt{t1} :^{\bot}_{\mathtt{o1}} \mathtt{o2!m().get} & & (3)\\
\longrightarrow \mathsf{H}_2[\mathtt{G1} \mapsto \top] \Vdash \mathtt{t} :^{\top}_{\mathtt{o}} \mathtt{t1.get}, \mathtt{t1} :^{\top}_{\mathtt{o1}} \mathtt{o2!m().get} & & (4)\\
\longrightarrow \mathsf{H}_2[\mathtt{G1} \mapsto \top] \Vdash \mathtt{t} :^{\top}_{\mathtt{o}} \mathtt{t1.get}, \mathtt{t1} :^{\top}_{\mathtt{o1}} \mathtt{t2.get}, \mathtt{t2} :^{\bot}_{\mathtt{o2}} \mathtt{newg\ C()} & & (5)\\
\longrightarrow \mathsf{H}_3 \Vdash \mathtt{t} :^{\top}_{\mathtt{o}} \mathtt{t1.get}, \mathtt{t1} :^{\top}_{\mathtt{o1}} \mathtt{t2.get}\ , \mathtt{t2} :^{\top}_{\mathtt{o2}} \mathtt{newg\ C()} & & (6)\\
\longrightarrow \mathsf{H}_4 \Vdash \mathtt{t} :^{\top}_{\mathtt{o}} \mathtt{t1.get}, \mathtt{t1} :^{\top}_{\mathtt{o1}} \mathtt{t2.get}\ , \mathtt{t2} :^{\top}_{\mathtt{o2}} \mathtt{o3} & & (7)\\
\longrightarrow \mathsf{H}_4 \Vdash \mathtt{t} :^{\top}_{\mathtt{o}} \mathtt{t1.get}, \mathtt{t1} :^{\top}_{\mathtt{o1}} \mathtt{o3}\ , \mathtt{t2} :^{\top}_{\mathtt{o2}} \mathtt{o3} & & (8)
\end{array}
$$

$$
\begin{array}{lll}
\text{where } \mathsf{H} = \mathtt{o} \mapsto (\mathtt{Object}, \mathtt{G}, [\ ]), \mathtt{G} \mapsto \top & & \mathsf{H}_1 = \mathsf{H}[\mathtt{o1} \mapsto (\mathtt{D}, \mathtt{G1}, [\ ]), \mathtt{G1} \mapsto \bot] \\
\mathsf{H}_2 = \mathsf{H}_1[\mathtt{o2} \mapsto (\mathtt{D}, \mathtt{G2}, [\ ]), \mathtt{G2} \mapsto \bot] & & \mathsf{H}_3 = \mathsf{H}_2[\mathtt{G1} \mapsto \top, \mathtt{G2} \mapsto \top] \\
\mathsf{H}_4 = \mathsf{H}_3[\mathtt{o3} \mapsto (\mathtt{C}, \mathtt{G3}, [\ ]), \mathtt{G3} \mapsto \bot] & &
\end{array}
$$

The reader may notice that, in the final configuration, the tasks `t1` and `t2` will terminate one after the other by releasing all the group locks.

3 Deadlocks

We introduce our formal developments about deadlock analysis by discussing a couple of expressions that manifest deadlocks. Let `D` be the class of Table 1 and consider the expression `(newg D())!n(new D()).get`. This expression differs from the one of Example 1 for the argument of the method (now it is `new D()`, before it was `newg D()`). The computation of `(newg D())!n(new D()).get` is the same of the one in Example 1 till step (5), replacing the value of H_2 with $\mathsf{H}_1[\mathtt{o2} \mapsto (\mathtt{D}, \mathtt{G1}, [\ ])]$ (`o1` and `o2` belong to the same group `G1`). At step (5), the task `t2` will indefinitely wait for getting the lock of `G1` since `t1` will never release it.

Deadlocks may be difficult to discover when they are caused by schedulers' choices. For example, let `D′` be the following extension of the class `C` in Table 1:

```
class D′ extends C {
    D′ n(D′ b, D′ c){ return b!p(c);c!p(b);this ;}
    C p(D′ c){ return (c!m()).get ;} }
```

and consider the expression `(newg D′())!n(newg D′(),new D′()).get`. The evaluation of this expression yields the tasks

$$\mathtt{t} :^{\bot}_{\mathtt{o}} \mathtt{o1}, \mathtt{t1} :^{\bot}_{\mathtt{o1}} \mathtt{o1}, \mathtt{t2} :^{\bot}_{\mathtt{o2}} \mathtt{o3!m().get}, \mathtt{t3} :^{\bot}_{\mathtt{o3}} \mathtt{o2!m().get}$$

with $\mathtt{o} \in \mathtt{G}, \mathtt{o1}, \mathtt{o3} \in \mathtt{G1}$ and $\mathtt{o2} \in \mathtt{G2}$. If `t2` is completed before `t3` grabs the lock (or conversely) then no deadlock will be manifested. On the contrary, the above tasks may evolve into

$$\mathtt{t} :^{\bot}_{\mathtt{o}} \mathtt{o1}, \mathtt{t1} :^{\bot}_{\mathtt{o1}} \mathtt{o1}, \mathtt{t2} :^{\top}_{\mathtt{o2}} \mathtt{t4.get}, \mathtt{t3} :^{\top}_{\mathtt{o3}} \mathtt{t5.get}, \mathtt{t4} :^{\bot}_{\mathtt{o3}} \mathtt{newg\ C()}, \mathtt{t5} :^{\bot}_{\mathtt{o2}} \mathtt{newg\ C()}$$

that is a deadlock because neither `t4` nor `t5` will have any chance to progress.

4 Contracts in `FJg`

In the following we will consider *plain* `FJg` programs where methods never return fields nor it is possible to invoke methods with fields in the subject part or in the object part. For example, the expressions `(this.f)!m()` and `x!n(this.f)` are not admitted, as well as a method declaration like `C p(){ return this.f ; }`. (The contract system in Table 4 and 5 will ban not-plain programs.) This restriction simplifies the foregoing formal developments about deadlock analysis; the impact of the restriction (and other ones) on the analysis of deadlocks is discussed in Section 8.

The analysis of deadlocks in `FJg` programs uses abstract descriptions of behaviours, called *contracts*, and an inference system for associating a contract to expressions (and methods). (The algorithm taking contracts and returning details about deadlocks is postponed to the next section.) Formally, *contracts* $\gamma, \gamma', \cdots$ are terms defined by the rules below:

$$\gamma ::= \varepsilon \mid \mathtt{C.m} : \mathtt{G}(\bar{\mathtt{G}}); \gamma \mid \mathtt{C.m}^{\mathtt{g}} : \mathtt{G}(\bar{\mathtt{G}}); \gamma \mid \mathtt{C.m}^{\mathtt{a}} : \mathtt{G}(\bar{\mathtt{G}}); \gamma$$

As usual, $\gamma; \varepsilon = \gamma = \varepsilon; \gamma$. When $\bar{\gamma}$ is a tuple of contracts $(\gamma_1, \cdots, \gamma_n)$, $\mathtt{seq}(\bar{\gamma})$ is a shortening for the sequential composition $\gamma_1; \cdots; \gamma_n$. The sequence γ collects the method invocations inside expressions. In particular, the items of the sequence may be empty, noted ε; or $\mathtt{C.m} : \mathtt{G}(\bar{\mathtt{G}})$, specifying that the method `m` of class `C` is going to be invoked on an object of group `G` and with arguments of group $\bar{\mathtt{G}}$; or $\mathtt{C.m}^{\mathtt{g}} : \mathtt{G}(\bar{\mathtt{G}})$, a method invocation followed by a `get` operation; or $\mathtt{C.m}^{\mathtt{a}} : \mathtt{G}(\bar{\mathtt{G}})$, a method invocation followed by an `await` operation. For example, the contract $\mathtt{C.m{:}G();\ D.n{:}G'()}$ defines two method invocations on groups `G` and $\mathtt{G}'$, respectively (methods carry no arguments). The contract $\mathtt{C.m{:}G();\ D.n^{g}{:}G'();\ E.p^{a}{:}G''()}$ defines three method invocations on different groups; the second invocation is followed by a `get` and the third one by an `await`.

Method contracts, ranged over by $\mathcal{G}, \mathcal{G}', \cdots$, are $\mathtt{G}(\bar{\mathtt{G}})\{\gamma\}\ \mathtt{G}'$, where $\mathtt{G}, \bar{\mathtt{G}}$ are pairwise different group names – $\mathtt{G}(\bar{\mathtt{G}})$ is the *header* –, $\mathtt{G}'$ is the *returned group*, and γ is a *contract*. A contract $\mathtt{G}(\bar{\mathtt{G}})\{\gamma\}\ \mathtt{G}'$ binds the group of the object `this` and the group of the arguments of the method invocation in the sequence γ. The returned group $\mathtt{G}'$ may belong to $\mathtt{G}, \bar{\mathtt{G}}$ or not, that is it may be a new group created by the method. For example, let $\gamma = \mathtt{C.m{:}G();\ D.n^{g}{:}G'();\ E.p^{a}{:}G''()}$ in (i) $\mathtt{G}(\mathtt{G}', \mathtt{G}'')\{\gamma\}\ \mathtt{G}''$ and (ii) $\mathtt{G}(\mathtt{G}')\{\gamma\}\ \mathtt{G}''$. In case (i) every group name in γ is *bound* by names in the header. This means that method invocations are bound to either the group name of the caller or to group names of the arguments. This is not the case for (ii), where the third method invocation in its body and the returned group address a group name that is unbound by the header. This means that the method with contract (ii) is creating an object of class `E` belonging to a new group – called $\mathtt{G}''$ in the body – and is performing the third invocation to a method of this object.

Method contracts are quotiented by the least equivalence $=^{\alpha}$ identifying two contracts that are equivalent after an injective renaming of (free and bound) group names. For example $\mathtt{G}(\mathtt{G}')\{\mathtt{C.m{:}G();\ D.n^{g}{:}G'();\ E.p^{a}{:}G''()}\}\ \mathtt{G}' =^{\alpha} \mathtt{G}_1(\mathtt{G}_2)\{\mathtt{C.m{:}G_1();\ D.n^{g}{:}G_2();\ E.p^{a}{:}G''()}\}\ \mathtt{G}_2$. Additionally, since the occurrence of $\mathtt{G}''$ represents an unbound group, writing $\mathtt{G}''$ or any other free group name is the same. That is $\mathtt{G}(\mathtt{G}')\{\mathtt{C.m{:}G();\ D.n^{g}{:}G'();\ E.p^{a}{:}G''()}\}\ \mathtt{G}' =^{\alpha} \mathtt{G}_1(\mathtt{G}_2)\{\mathtt{C.m{:}G_1();\ D.n^{g}{:}G_2();\ E.p^{a}{:}G_3()}\}\ \mathtt{G}_2$.

Let Γ, called *environment*, be a map from either names to pairs $(\mathtt{T}, \mathtt{G})$ or class and method names, *i.e.* `C.m`, to terms $\mathtt{G}(\bar{\mathtt{G}}) \to \mathtt{G}'$, called *group types*, where $\mathtt{G}, \bar{\mathtt{G}}, \mathtt{G}'$ are all different from $\supseteq$. The contract judgement for expressions has the following form and meaning: $\Gamma \vdash \mathtt{e} : (\mathtt{T}, \mathtt{G}), \gamma$ means that the expression `e` has type `T` and group `G` and has contract γ in the environment Γ.

Contract rules for expressions are presented in Tables 4 where,

- in rule (T-Invk) we use the operator *fresh*$(\bar{\mathtt{G}}, \mathtt{G})$ that returns `G` if $\mathtt{G} \in \bar{\mathtt{G}}$ or a fresh group name otherwise;

$$\text{(T-Var)}\quad \Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x}), \varepsilon \qquad\qquad \text{(T-Field)}\quad \frac{\Gamma \vdash \mathtt{this} : (\mathtt{C},\mathtt{G}), \varepsilon \quad \mathtt{D}\ \mathtt{f} \in \mathit{fields}(\mathtt{C})}{\Gamma \vdash \mathtt{this.f} : (\mathtt{D}, \Game), \varepsilon}$$

$$\text{(T-Invk)}\quad \frac{\begin{array}{c}\Gamma \vdash \mathtt{e} : (\mathtt{C},\mathtt{G}), \gamma \quad \Gamma \vdash \bar{\mathtt{e}} : (\bar{\mathtt{D}}, \bar{\mathtt{G}}), \bar{\gamma} \quad \Game \notin \mathtt{G}\bar{\mathtt{G}} \\ \mathit{mtype}(\mathtt{m}, \mathtt{C}) = \bar{\mathtt{C}} \to \mathtt{C}' \quad \bar{\mathtt{D}} <: \bar{\mathtt{C}} \\ \Gamma(\mathtt{C.m}) = \mathtt{G}'(\bar{\mathtt{G}}') \to \mathtt{G}'' \quad \mathtt{G}''' = \mathit{fresh}(\mathtt{G}'\bar{\mathtt{G}}', \mathtt{G}''[^{\mathtt{G}\bar{\mathtt{G}}}/_{\mathtt{G}'\bar{\mathtt{G}}'}])\end{array}}{\Gamma \vdash \mathtt{e!m}(\bar{\mathtt{e}}) : (\mathtt{Fut}(\mathtt{C}'), \mathtt{G}'''), \gamma; (\mathtt{seq}(\bar{\gamma})); \mathtt{C.m} : \mathtt{G}(\bar{\mathtt{G}})}$$

$$\text{(T-New)}\quad \frac{\begin{array}{c}\Gamma \vdash \mathtt{this} : (\mathtt{C},\mathtt{G}), \varepsilon \quad \Gamma \vdash \bar{\mathtt{e}} : (\bar{\mathtt{C}}, \bar{\mathtt{G}}), \bar{\gamma} \\ \mathit{fields}(\mathtt{C}') = \bar{\mathtt{C}}'\ \bar{\mathtt{f}}' \quad \bar{\mathtt{C}} <: \bar{\mathtt{C}}'\end{array}}{\Gamma \vdash \mathtt{new}\ \mathtt{C}'(\bar{\mathtt{e}}) : (\mathtt{C}', \mathtt{G}), (\mathtt{seq}(\bar{\gamma}))} \qquad \text{(T-NewG)}\quad \frac{\begin{array}{c}\Gamma \vdash \bar{\mathtt{e}} : (\bar{\mathtt{C}}, \bar{\mathtt{G}}), \bar{\gamma} \quad \mathtt{G}\ \mathit{fresh} \\ \mathit{fields}(\mathtt{C}) = \bar{\mathtt{C}}'\ \bar{\mathtt{f}}' \quad \bar{\mathtt{C}} <: \bar{\mathtt{C}}'\end{array}}{\Gamma \vdash \mathtt{newg}\ \mathtt{C}(\bar{\mathtt{e}}) : (\mathtt{C}, \mathtt{G}), (\mathtt{seq}(\bar{\gamma}))}$$

$$\text{(T-Get)}\quad \frac{\Gamma \vdash \mathtt{e} : (\mathtt{Fut}(\mathtt{C}), \mathtt{G}), \gamma}{\Gamma \vdash \mathtt{e.get} : (\mathtt{C}, \mathtt{G}), \gamma \between \mathtt{get}} \qquad \text{(T-Await)}\quad \frac{\Gamma \vdash \mathtt{e} : (\mathtt{Fut}(\mathtt{C}), \mathtt{G}), \gamma}{\Gamma \vdash \mathtt{e.await} : (\mathtt{Fut}(\mathtt{C}), \mathtt{G}), \gamma \between \mathtt{await}}$$

$$\text{(T-Update)}\quad \frac{\begin{array}{c}\Gamma \vdash \mathtt{this} : (\mathtt{C},\mathtt{G}), \varepsilon \quad \mathtt{D}\ \mathtt{f} \in \mathit{fields}(\mathtt{C}) \\ \Gamma \vdash \mathtt{e} : (\mathtt{D}', \mathtt{G}'), \gamma \quad \mathtt{D}' <: \mathtt{D}\end{array}}{\Gamma \vdash \mathtt{this.f} = \mathtt{e} : (\mathtt{D}', \mathtt{G}'), \gamma} \qquad \text{(T-Seq)}\quad \frac{\Gamma \vdash \mathtt{e} : (\mathtt{T}, \mathtt{G}), \gamma \quad \Gamma \vdash \mathtt{e}' : (\mathtt{T}', \mathtt{G}'), \gamma'}{\Gamma \vdash \mathtt{e}\,;\,\mathtt{e}' : (\mathtt{T}', \mathtt{G}'), \gamma; \gamma'}$$

Table 4. Contract rules of FJg expressions

– in rules (T-Get) and (T-Await), we use the operator $\between$ defined as follows:

$$\begin{array}{rl}\varepsilon \between \mathtt{await} & = \varepsilon \\ (\gamma; \mathtt{C.m}\ \mathtt{G}(\bar{\mathtt{G}})) \between \mathtt{await} & = \gamma; \mathtt{C.m}^{\mathtt{a}}\ \mathtt{G}(\bar{\mathtt{G}}) \\ (\gamma; \mathtt{C.m}^{\mathtt{a}}\ \mathtt{G}(\bar{\mathtt{G}})) \between \mathtt{await} & = \gamma; \mathtt{C.m}^{\mathtt{a}}\ \mathtt{G}(\bar{\mathtt{G}}) \\ (\gamma; \mathtt{C.m}\ \mathtt{G}(\bar{\mathtt{G}})) \between \mathtt{get} & = \gamma; \mathtt{C.m}^{\mathtt{g}}\ \mathtt{G}(\bar{\mathtt{G}}) \\ (\gamma; \mathtt{C.m}^{\mathtt{a}}\ \mathtt{G}(\bar{\mathtt{G}})) \between \mathtt{get} & = \gamma; \mathtt{C.m}^{\mathtt{a}}\ \mathtt{G}(\bar{\mathtt{G}})\end{array}$$

(the other combinations of get and await are forbidden by the contract system).

The rule (T-Field) associates the group $\Game$ to the expression this.f, provided the field f exists. This judgment, together with the premises of (T-Invk) and the assumption that $\Game$ does not appear in $\Gamma(\mathtt{C.m})$, imply that subjects and objects of method invocations cannot be expressions as this.f. Apart these constraint, the contract of $\mathtt{e!m}(\bar{\mathtt{e}})$ is as expected, *i.e.* the sequence of the contract of e, plus the one of $\bar{\mathtt{e}}$, with a tailing item $\mathtt{C.m}\ \mathtt{G}(\bar{\mathtt{G}})$. Rules (T-New) and (T-NewG) are almost the same, except the fact that the latter one returns a fresh group name while the former one return the group of this. The other rules are standard.

Let $\mathtt{G}(\bar{\mathtt{G}}) \to \mathtt{G}' =^{\alpha} \mathtt{H}(\bar{\mathtt{H}}) \to \mathtt{H}'$ if and only if $\mathtt{G}(\bar{\mathtt{G}})\{\varepsilon\}\mathtt{G}' =^{\alpha} \mathtt{H}(\bar{\mathtt{H}})\{\varepsilon\}\mathtt{H}'$. The contract judgements for method declarations, class declarations and programs have the following forms and meanings:

– $\Gamma; \mathtt{C} \vdash \mathtt{D}'\ \mathtt{m}\ (\bar{\mathtt{D}}\ \bar{\mathtt{x}})\{\mathtt{return}\ \mathtt{e};\} : \mathtt{G}(\bar{\mathtt{G}})\{\gamma\}\ \mathtt{G}'$ means that the method $\mathtt{D}'\ \mathtt{m}\ (\bar{\mathtt{D}}\ \bar{\mathtt{x}})\{\mathtt{return}\ \mathtt{e};\}$ has method contract $\mathtt{G}(\bar{\mathtt{G}})\{\gamma\}\ \mathtt{G}'$ in the class C and in the environment Γ;

– $\Gamma \vdash \mathtt{class}\ \mathtt{C}\ \mathtt{extends}\ \mathtt{D}\ \{\bar{\mathtt{C}}\ \bar{\mathtt{f}};\ \ \bar{\mathtt{M}}\} : \{\bar{\mathtt{m}} \mapsto \bar{\mathcal{G}}\}$ means that the class declaration C has contract $\{\bar{\mathtt{m}} \mapsto \bar{\mathcal{G}}\}$ in the environment Γ;

– $\vdash (\textsc{ct}, \mathtt{e}) : \textsc{cct}, (\mathtt{T}, \mathtt{G}), \gamma$ means that the program $(\textsc{ct}, \mathtt{e})$ has *contract class table* $\textsc{cct}$ and type/-group/contract $(\mathtt{T}, \mathtt{G}), \gamma$, where a contract class table maps class names to terms $\{\bar{\mathtt{m}} : \bar{\mathcal{G}}\}$.

Table 5 reports the typing judgments for method and class declarations and for programs. We use the auxiliary function $\mathit{mname}(\bar{\mathtt{M}})$ that returns the sequence of method names in $\bar{\mathtt{M}}$. We also

Contractually correct method declaration and class declaration:

$$\text{(T-METHOD)}\quad \frac{\begin{array}{c} \mathit{mtype}(\mathtt{m},\mathtt{C}) = \bar{\mathtt{C}}' \to \mathtt{C}' \qquad \mathit{mbody}(\mathtt{m},\mathtt{C}) = \bar{\mathtt{x}}.\mathtt{e} \\ \bar{\mathtt{G}}, \mathtt{G}\ \text{fresh} \quad \Gamma + \bar{\mathtt{x}} : (\bar{\mathtt{C}}', \bar{\mathtt{G}}) + \mathtt{this} : (\mathtt{C},\mathtt{G}) \vdash \mathtt{e} : (\mathtt{T}',\mathtt{G}'),\gamma \quad \mathtt{T}' <: \mathtt{C}' \\ \Gamma(\mathtt{C.m}) =^{\alpha} \mathtt{G}(\bar{\mathtt{G}}) \to \mathtt{G}' \end{array}}{\Gamma; \mathtt{C} \vdash\ \mathtt{C}'\ \mathtt{m}\ (\bar{\mathtt{C}}\ \bar{\mathtt{x}})\{\mathtt{return}\ \mathtt{e};\} : \mathtt{G}(\bar{\mathtt{G}})\{\gamma\}\ \mathtt{G}'}$$

Contractually correct class and program:

$$\text{(T-CLASS)}\quad \frac{\Gamma; \mathtt{C} \vdash \bar{\mathtt{M}} : \bar{\mathcal{G}}}{\Gamma \vdash \mathtt{class}\ \mathtt{C}\ \mathtt{extends}\ \mathtt{D}\ \{\bar{\mathtt{C}}\ \bar{\mathtt{f}};\ \ \bar{\mathtt{M}}\} : \{\mathit{mname}(\bar{\mathtt{M}}) \mapsto \bar{\mathcal{G}}\}}$$

$$\text{(T-PROGRAM)}\quad \frac{\begin{array}{c} \mathtt{C} \in \mathit{dom}(\textsc{ct})\ \text{ implies }\ \Gamma \vdash \textsc{ct}(\mathtt{C}) : \textsc{cct}(\mathtt{C}) \\ \textsc{cct}\ \mathit{is}\ \textsc{ct}\ \mathit{consistent} \\ \mathtt{G}\ \mathit{fresh} \quad \Gamma + \mathtt{this} : (\mathtt{Object},\mathtt{G}) \vdash \mathtt{e} : (\mathtt{T},\mathtt{G}'),\gamma \end{array}}{\vdash (\textsc{ct}, \mathtt{e}) : \textsc{cct}, (\mathtt{T},\mathtt{G}'),\gamma}$$

Table 5. Contract rules for method declarations and class declarations

write $m \in \textsc{ct}(\mathtt{C})$ if $\textsc{ct}(\mathtt{C}) = \mathtt{class\ C\ extends\ D}\ \{\bar{\mathtt{C}}\ \bar{\mathtt{f}};\ \ \bar{\mathtt{M}}\}$ and $m \in \mathit{mname}(\bar{\mathtt{M}})$. Rule (T-PROGRAM) requires that if a subclass overrides a method of a superclass then the two methods must have equal contract. This constraint is expressed by the predicate CCT *is* CT *consistent* defined as follows:

$$\begin{array}{l} \textit{for every}\ \textsc{ct}(\mathtt{C}) = \mathtt{class}\ \mathtt{C}\ \mathtt{extends}\ \mathtt{D}\ \{\cdots\} : \\ \qquad\qquad m \in \textsc{ct}(\mathtt{C})\ \textit{and}\ m \in \textsc{ct}(\mathtt{D}) \quad \textit{implies} \quad \textsc{cct}(\mathtt{C})(\mathtt{m}) =^{\alpha} \textsc{cct}(\mathtt{D})(\mathtt{m}) \end{array}$$

This consistency requirement may be definitely weakened: we defer to future works the issue of studying a sub-contract relation that is correct with respect to class inheritance.

The proof of correctness of the contract system in Tables 4 and 5 requires additional rules that define the contract correctness of (runtime) configurations. These rules are:

$$\text{(T-TASK)}\quad \frac{\Gamma \vdash \mathtt{this} : (\mathtt{C},\mathtt{G}),\varepsilon \quad \Gamma \vdash \mathtt{t} : (\mathtt{Fut(C)},\mathtt{G}'),\varepsilon}{\Gamma \vdash \mathtt{t.get} : (\mathtt{C},\mathtt{G}'),(\mathtt{G},\mathtt{G}')} \qquad \text{(T-GETR)}\quad \frac{\Gamma \vdash \mathtt{e} : (\mathtt{Fut(C)},\mathtt{G}),\varepsilon \quad \mathtt{e} \neq \mathtt{t}}{\Gamma \vdash \mathtt{e.get} : (\mathtt{C},\mathtt{G}),\varepsilon}$$

$$\text{(T-CONFIGURATION)}\quad \frac{\begin{array}{c} \mathit{fields}(\mathtt{C}) = \bar{\mathtt{C}}\ \bar{\mathtt{f}} \\ \mathsf{H}(\mathtt{o}) = (\mathtt{C},\mathtt{G},[\bar{\mathtt{f}} : \bar{\mathtt{v}}]) \quad \text{implies} \quad \Gamma \vdash \mathtt{o} : (\mathtt{C},\mathtt{G}),\varepsilon \text{ and } \Gamma \vdash \bar{\mathtt{v}} : \bar{\mathtt{C}}' \text{ and } \bar{\mathtt{C}}' <: \bar{\mathtt{C}} \\ \mathtt{t} :_{\mathtt{o}} \mathtt{e} \in \mathsf{S} \quad \text{implies} \quad \Gamma \vdash \mathtt{t} : (\mathtt{Fut(D)},\mathtt{G}'),\varepsilon \text{ and } \Gamma \vdash \mathtt{e} : (\mathtt{D},\mathtt{G}'),\gamma \text{ and } \mathtt{o} \in \mathit{dom}(\mathsf{H}) \end{array}}{\Gamma \vdash (\mathsf{H} \Vdash \mathsf{S})}$$

Rule (T-TASK) define contract correctness of runtime expressions as `t.get`. The rule uses contracts extended with terms $(\mathtt{G},\mathtt{G}').\gamma$. While rule (T-GETR) deals with the expression `t.await.get`. It is worth to notice the absence of rules for the runtime expression `t.await`. In fact, the judgment of this expression follows by (T-AWAIT) and the definition of $\varepsilon \between \mathtt{await}$.

5 Deadlock analysis in FJg

The contract system in Tables 4 and 5 does not convey any information about deadlocks: it only associates contracts to expressions (and methods). The point is that contracts retain the necessary informations about deadlocks and the analysis may be safely reduced to them, overlooking all the other details. We begin with the formal definition of a deadlock.

Definition 1. *A configuration* $\mathsf{H} \Vdash \mathsf{S}$ *is* deadlocked *if there are* $\mathtt{t}_i$, o_i, E_i, *and* e_i, *with* $1 \leq i \leq n{+}k$, *such that*

- $n \geq 1$;
- *every* $1 \leq i \leq n$ *is* $\mathtt{t}_i :^{\top}_{\mathsf{o}_i} \mathsf{E}_i[\mathtt{t}_{\ell_i}.\mathtt{get}]$ *with* $\ell_i \in 1..n+k$ *and*
- *every* $n+1 \leq j \leq n+k$ *is* $\mathtt{t}_j :^{\bot}_{\mathsf{o}_j} \mathsf{e}_j$ *with* $group(\mathsf{H}, \mathsf{o}_j) \in \{group(\mathsf{H}, \mathsf{o}_1), \cdots, group(\mathsf{H}, \mathsf{o}_n)\}$.

A configuration $\mathsf{H} \Vdash \mathsf{S}$ *is* deadlock-free *if, for every* $\mathsf{H} \Vdash \mathsf{S} \longrightarrow^* \mathsf{H}' \Vdash \mathsf{S}'$, $\mathsf{H}' \Vdash \mathsf{S}'$ *is not deadlocked. A program* (CT, e) *is deadlock-free if its initial configuration is deadlock-free.*

It is easy to verify that the programs discussed in Section 3 are not deadlock-free. We observe that a configuration may have a blocked task without retaining any deadlock. This is the case of the process $\mathtt{C.m}^{\mathtt{g}}\ \mathtt{G}()$, where $\mathtt{C.m}\ :\ \mathtt{G}()\{\mathtt{C.m}^{\mathtt{g}}(\mathtt{G}')\}\ \mathtt{G}$, that produces an infinite chain of tasks $\mathtt{t}_i :^{\top}_{\mathsf{o}_i} \mathtt{t}_{i+1}.\mathtt{get}$. (The following dla algorithm will reject this contract.)

We say that a configuration $\mathsf{H} \Vdash \mathsf{S}$ has a *group-dependency* $(\mathtt{G}, \mathtt{G}')$ if S contains either the tasks $\mathtt{t} :^{\top}_{\mathsf{o}} \mathsf{E}[\mathtt{t}'.\mathtt{get}], \mathtt{t}' :_{\mathsf{o}'} \mathsf{e}$, with $\mathtt{t}'$ retaining or not its group lock, or the tasks $\mathtt{t} :^{\bot}_{\mathsf{o}} \mathsf{e}, \mathtt{t}' :^{\top}_{\mathsf{o}'} \mathsf{E}[\mathtt{t}''.\mathtt{get}]$ (in both cases e is not a value) and $\mathtt{G} = group(\mathsf{H}, \mathsf{o})$ and $\mathtt{G}' = group(\mathsf{H}, \mathsf{o}')$. A configuration contains a *group-circularity* if the transitive closure of its group-dependencies has a pair $(\mathtt{G}, \mathtt{G})$. The following statement asserts that a group-circularity signals the presence of a sequence of tasks mutually waiting for the release of the group lock of the other.

Proposition 1. *A configuration is deadlocked if and only if it has a* group-circularity.

In the following, sets of dependencies will be noted $\boldsymbol{G}, \boldsymbol{G}', \cdots$. Sequences $\boldsymbol{G}_1; \cdots; \boldsymbol{G}_n$ are also used and shortened into $\overline{\boldsymbol{G}}$. Let $\boldsymbol{G} \cup (\boldsymbol{G}_1; \cdots; \boldsymbol{G}_n)$ be $\boldsymbol{G} \cup \boldsymbol{G}_1; \cdots; \boldsymbol{G} \cup \boldsymbol{G}_n$. A set $\boldsymbol{G}$ is *not circular*, written $\boldsymbol{G}$: not-circular, if the transitive closure of $\boldsymbol{G}$ does not contain any pair $(\mathtt{G}, \mathtt{G})$. The definition of being not circular is extended to sequences $\boldsymbol{G}_1; \cdots; \boldsymbol{G}_n$, written $\boldsymbol{G}_1; \cdots; \boldsymbol{G}_n$: not-circular, by constraining every $\boldsymbol{G}_i$ to be not circular.

Dependencies between group names are extracted from contracts by the algorithm dla defined in Table 6. This algorithm takes an *abstract class contract table* Δ_{CCT}, a group name G and a contract γ and returns a sequence $\overline{\boldsymbol{G}}$. The abstract class contract table Δ_{CCT} takes a pair class name C/method name m, written C.m, and returns an abstract method contract $(\mathtt{G}.\bar{\mathtt{G}})\boldsymbol{G}$. The map Δ_{CCT} is *the least one* such that

$$\Delta_{\mathrm{CCT}}(\mathtt{C.m}) = (\mathtt{G}.\bar{\mathtt{G}}) \bigcup_{i \in 1..n} \boldsymbol{G}_i \quad \text{if and only if} \quad \begin{array}{l} \mathrm{CCT}(\mathtt{C})(\mathtt{m}) = \mathtt{G}(\bar{\mathtt{G}})\{\gamma\}\ \mathtt{G}' \\ \text{and } \mathtt{dla}(\Delta_{\mathrm{CCT}}, \mathtt{G}, \gamma) = \boldsymbol{G}_1; \cdots; \boldsymbol{G}_n \end{array}$$

We notice that Δ_{CCT} is well-defined because: (i) group names in CCT are finitely many; (ii) dla never introduces new group names; (iii) for every C.m, the element $\Delta_{\mathrm{CCT}}(\mathtt{C.m})$ is a finite lattice where elements have shape $(\mathtt{G}.\bar{\mathtt{G}})\boldsymbol{G}$ and where the greatest set $\boldsymbol{G}$ is the cartesian product of group names in CCT. Additionally, in order to augment the precision of Δ_{CCT}, we assume that CCT satisfies the constraint that, for every C.m and D.n such that $\mathtt{C.m} \neq \mathtt{D.n}$, CCT(C)(m) and CCT(D)(n) have no group name in common (both bound and free). (When this is not the case, sets in the codomain of Δ_{CCT} are smaller, thus manifesting more circularities.)

(DLA-ε)
$$\mathtt{dla}(\Delta_{\rm CCT}, \mathtt{G}, \varepsilon) = \emptyset$$

(DLA-INVK)
$$\frac{\Delta_{\rm CCT}(\mathtt{C.m}) = (\mathtt{G}''.\bar{\mathtt{G}}'')\boldsymbol{G} \quad \mathtt{dla}(\Delta_{\rm CCT}, \mathtt{G}, \gamma') = \overline{\boldsymbol{G}}}{\mathtt{dla}(\Delta_{\rm CCT}, \mathtt{G}, \mathtt{C.m}\ \mathtt{G}'(\bar{\mathtt{G}}'); \gamma') = \boldsymbol{G}[^{\mathtt{G}'; \bar{\mathtt{G}}'}/_{\mathtt{G}''; \bar{\mathtt{G}}''}] \cup \overline{\boldsymbol{G}}}$$

(DLA-GET)
$$\frac{\Delta_{\rm CCT}(\mathtt{C.m}) = (\mathtt{G}''.\bar{\mathtt{G}}'')\boldsymbol{G} \quad \mathtt{dla}(\Delta_{\rm CCT}, \mathtt{G}, \gamma') = \overline{\boldsymbol{G}}}{\mathtt{dla}(\Delta_{\rm CCT}, \mathtt{G}, \mathtt{C.m}^{\mathtt{g}}\ \mathtt{G}'(\bar{\mathtt{G}}'); \gamma') = (\{(\mathtt{G}, \mathtt{G}')\} \cup \boldsymbol{G}[^{\mathtt{G}'; \bar{\mathtt{G}}'}/_{\mathtt{G}''; \bar{\mathtt{G}}''}]); \overline{\boldsymbol{G}}}$$

(DLA-AWAIT)
$$\frac{\Delta_{\rm CCT}(\mathtt{C.m}) = (\mathtt{G}''.\bar{\mathtt{G}}'')\boldsymbol{G} \quad \mathtt{dla}(\Delta_{\rm CCT}, \mathtt{G}, \gamma') = \overline{\boldsymbol{G}}}{\mathtt{dla}(\Delta_{\rm CCT}, \mathtt{G}, \mathtt{C.m}^{\mathtt{a}}\ \mathtt{G}'(\bar{\mathtt{G}}'); \gamma') = \boldsymbol{G}[^{\mathtt{G}'; \bar{\mathtt{G}}'}/_{\mathtt{G}''; \bar{\mathtt{G}}''}]; \overline{\boldsymbol{G}}}$$

Table 6. The algorithm `dla`

Let us comment the rules of Table 6. The second rule of `dla` accounts for method invocations $\mathtt{C.m}\ \mathtt{G}'(\bar{\mathtt{G}}'); \gamma'$. Since the code of `C.m` will run asynchronously with respect to the continuation γ', *i.e.* it may be executed at any stage of γ', the rule adds the pairs of `C.m` (stored in $\Delta_{\rm CCT}(\mathtt{C.m})$) to every set of the sequence corresponding to γ'. The third rule of `dla` accounts for method invocations followed by `get` $\mathtt{C.m}^{\mathtt{g}}\ \mathtt{G}'(\bar{\mathtt{G}}'); \gamma'$. Since the code of `C.m` will run *before* the continuation γ', the rule prefixes the sequence corresponding to γ' with the pairs of `C.m` extended with $(\mathtt{G}, \mathtt{G}')$, where `G` is the group of the caller and $\mathtt{G}'$ is the group of the called method. The rule for method invocations followed by `await` is similar to the previous one, except that no pair is added because the `await` operation releases the group lock of the caller.

A program (CT, `e`) is deadlock free if $\vdash$ (CT, `e`) : CCT, $(\mathtt{T}, \mathtt{G}), \gamma$ and $\mathtt{dla}(\Delta_{\rm CCT}, \mathtt{G}, \gamma)$: `not-circular`, where `G` is a fresh group name, that is `G` does not clash with group names in CCT (group names in γ are either `G` or fresh as well – see Table 4).

Example 2. Let `C` and `D` be the classes of Table 1 and $\mathtt{D}'$ be the class in Section 3. We derive the following contract class table CCT and abstract contract class table $\Delta_{\rm CCT}$:

$\mathtt{C.m} \mapsto \mathtt{G()\{\ \varepsilon\ \}\ G'}$	$\mathtt{C.m} \mapsto \mathtt{(G)}\emptyset$
$\mathtt{D.n} \mapsto \mathtt{E(E')\{D.m^{g}:E'()\}\ E''}$	$\mathtt{D.n} \mapsto \mathtt{(E,E')\{\ (E,E')\ \}}$
$\mathtt{D.m} \mapsto \mathtt{F()\{\ \varepsilon\ \}\ F'}$	$\mathtt{D.m} \mapsto \mathtt{(F)}\emptyset$
$\mathtt{D'.n} \mapsto \mathtt{H(H',H'')\{D'.p:H'(H'');\ D'.p:H''(H')\}\ H}$	$\mathtt{D'.n} \mapsto \mathtt{(H,H',H'')\{(H',H''),(H'',H')\}}$
$\mathtt{D'.p} \mapsto \mathtt{I(I')\{\ D'.m^{g}:\ I'()\ \}\ I''}$	$\mathtt{D'.p} \mapsto \mathtt{(I,I')\{\ (I,I')\ \}}$
$\mathtt{D'.m} \mapsto \mathtt{L()\ \{\ \varepsilon\ \}\ L'}$	$\mathtt{D'.m} \mapsto \mathtt{(L)}\emptyset$

Now consider the expressions `(newg D())!n(newg D()).get` and `(newg D())!n(new D()).get` of Section 2, which have contracts $\mathtt{D.n^{g}}$:L″(L‴) and $\mathtt{D.n^{g}}$:L″(L′), respectively, being L′ the group of `this`. We obtain

$$\begin{aligned} \mathtt{dla}(\Delta_{\rm CCT}, \mathtt{L}', \mathtt{D.n^{g}} : \mathtt{L''(L''')}) &= \{(\mathtt{L''}, \mathtt{L'''}), (\mathtt{L'}, \mathtt{L''})\} \\ \mathtt{dla}(\Delta_{\rm CCT}, \mathtt{L}', \mathtt{D.n^{g}} : \mathtt{L''(L')}) &= \{(\mathtt{L''}, \mathtt{L'}), (\mathtt{L'}, \mathtt{L''})\} \end{aligned}$$

where the first set of dependencies has no group-circularity – therefore `(newg D())!n(newg D()).get` is deadlock-free – while the second has a group-circularity – `(newg D())!n(new D()).get` – may manifest a deadlock, and indeed it does.

Next consider the expression `(newg D'())!n(newg D'(),new D'()).get` of Section 3, which has contract $\mathtt{D'.n^{g}}$:L″(L‴,L′), being L′ the group of `this`. We obtain

$$\mathtt{dla}(\Delta_{\rm CCT}, \mathtt{L}', \mathtt{(newg\ D'())!n(newg\ D'(),new\ D'()).get}) = \{(\mathtt{L'''}, \mathtt{L'}), (\mathtt{L'}, \mathtt{L'''}), (\mathtt{L'}, \mathtt{L''})\}$$

where the set of dependencies manifests circularities. In fact, in Section 3, we observed that the above expression may manifest a deadlock.

The algorithm dla may be strengthened in several ways. Let us discuss this issue with a couple of examples. Let C′ be the class

```
class C′ {
    C′ m(C′ b, C′ c){ return b!n(c).get ; c!n(b).get ;}
    C′ n(D′ c){ return (c!p).get ;}
    C′ p() { return new C′() ;}  }
```

and let cct be its contract class table:

$$
\begin{array}{l}
\texttt{C'.m} \mapsto \texttt{G(G',G'')\{C'.n}^{g}\texttt{:G'(G'')} \ ; \ \texttt{C'.n}^{g}\texttt{:G''(G')\} G'} \\
\texttt{C'.n} \mapsto \texttt{F(F')\{C'.p}^{g}\texttt{:F'()\} F'} \\
\texttt{C'.p} \mapsto \texttt{E()}\{\epsilon\}\texttt{ E}
\end{array}
$$

The reader may verify that the expression (new C′())!m1(newg C′(), newg C′()) never deadlocks. However, since $\Delta_{\text{CCT}}(\texttt{C'.m}) = (\texttt{G.G'G''})\{(\texttt{G},\texttt{G'}), (\texttt{G'},\texttt{G''}), (\texttt{G},\texttt{G''}), (\texttt{G''},\texttt{G'})\}$, the algorithm dla wrongly returns a circular set of dependencies. This problem is due to the fact that Δ_{CCT} melds the group dependencies of different time points into a single set. Had we preserved the temporal separation, that is $\{(\texttt{G},\texttt{G'}), (\texttt{G'},\texttt{G''})\}; \{(\texttt{G},\texttt{G''}), (\texttt{G''},\texttt{G'})\}$, no group-circularity should have been manifested.

The second problem is due to the fact that free group names in method contracts should be renamed each time the method is invoked (with fresh group names) because two invocations of a same method create different group names. On the contrary, the algorithm dla always uses the same (free) name. This oversimplification gives a wrong result in this case. Let C″ be

```
class C″ {   C″ m(){ return (newg C″())!m().get ;}  }
```

(with $\text{cct}(\texttt{C''.m}) = \texttt{G()\{C'.m}^{g} : \texttt{G'()\}G'}$) and consider the expression (newg C″())!m().get. The evaluation of this expression never manifests a deadlock, however its contract is $\texttt{C''.m}^{g} : \texttt{F}()$ and the algorithm dla will return the set $\{(\texttt{G},\texttt{F}), (\texttt{F},\texttt{G'}), (\texttt{G'},\texttt{G'}),\}$, which has a group-circularity. In the conclusions we will discuss the techniques for reducing these errors.

6 Properties

In the following we consider *redexes* as the active (to be reduced) part in an expression. $\texttt{r} ::= \texttt{o.f} = \texttt{v} \mid \texttt{o!m}(\bar{\texttt{v}}) \mid \texttt{new C}(\bar{\texttt{v}}) \mid \texttt{newg C}(\bar{\texttt{v}}) \mid \texttt{t.get} \mid \texttt{t.await}$.

Lemma 1. *Given a well-typed runtime expression* e *that is not a value and is different from* this.f*, there exist* E *and* r *such that* e = E[r].

Proof. **Case e.m(ē).** Either e = o or we can apply the induction hypothesis to e deriving that there is an evaluation context E such that e = E[r] for some redex r. Therefore, E.m(ē) is the evaluation context for e.m(ē). If e = o, either for all $\texttt{e'} \in \bar{\texttt{e}}$ we have that e′ = v for some v in which case the expression is a redex and the evaluation context is [ ], or there is a minimum j such that $\texttt{e}_j$ is not a value and for all $k < j$ the expression $\texttt{e}_k$ is a value. In this case we apply the induction hypothesis to $\texttt{e}_j$ deriving that there is an evaluation context E such that $\texttt{e}_j = \texttt{E[r]}$ for some r. Therefore, $\texttt{o.m}(\texttt{v}_1, \ldots, \texttt{v}_{j-1}, \texttt{E}, \texttt{e}_{j+1}, \ldots)$ is the evaluation context for e.m(ē).

Case $\mathtt{e}; \mathtt{e}'$. If $\mathtt{e}$ is not a value we can apply the induction hypothesis to $\mathtt{e}$ deriving that there is an evaluation context E such that $\mathtt{e} = \mathsf{E}[\mathtt{r}]$ for some $\mathtt{r}$. Therefore, $\mathsf{E}; \mathtt{e}'$ is the evaluation context for $\mathtt{e}; \mathtt{e}'$. If $\mathtt{e}$ is a value, then the expression is a redex and $[\ ]$ is the evaluation context for the expression.

Remaining Cases. Similar to the previous ones. □

Lemma 2. *If* $\Gamma \vdash \mathsf{E}[\mathtt{r}] : (\mathtt{T}, \mathtt{G}), \gamma$, *then* $\Gamma \vdash \mathtt{r} : (\mathtt{T}', \mathtt{G}'), \gamma'$, *for some* $\mathtt{T}', \mathtt{G}'$ *and* γ' *s.t.* $\gamma = \gamma'; \gamma''$ *or* $\gamma = \gamma' \between \mathtt{get}; \gamma''$ *or* $\gamma = \gamma' \between \mathtt{await}; \gamma''$, *for some* γ''.

Proof. The proofs is by straightforward induction on the derivation of $\Gamma \vdash \mathsf{E}[\mathtt{r}] : (\mathtt{T}, \mathtt{G}), \gamma$.□

Definition 2 ($\leq_\alpha$).

$$
\begin{array}{l}
\varepsilon \leq_\alpha \varepsilon \\
\mathtt{C} \leq \mathtt{D} \textit{ and } \gamma \leq_\alpha \gamma' \textit{ imply } \mathtt{C.m} : \mathtt{G}(\bar{\mathtt{G}}); \gamma \leq_\alpha \mathtt{D.m} : \mathtt{G}(\bar{\mathtt{G}}); \gamma' \\
\mathtt{C} \leq \mathtt{D} \textit{ and } \gamma \leq_\alpha \gamma' \textit{ imply } \mathtt{C.m}^{\mathtt{g}} : \mathtt{G}(\bar{\mathtt{G}}); \gamma \leq_\alpha \mathtt{D.m}^{\mathtt{g}} : \mathtt{G}(\bar{\mathtt{G}}); \gamma' \\
\mathtt{C} \leq \mathtt{D} \textit{ and } \gamma \leq_\alpha \gamma' \textit{ imply } \mathtt{C.m}^{\mathtt{a}} : \mathtt{G}(\bar{\mathtt{G}}); \gamma \leq_\alpha \mathtt{D.m}^{\mathtt{a}} : \mathtt{G}(\bar{\mathtt{G}}); \gamma'
\end{array}
$$

Definition 3 ($\lessdot$).

$$
\gamma' \lessdot \gamma \ \textit{ iff } \ \gamma = \begin{cases} \gamma_0; \gamma_0', \textit{ where } \gamma_0 \in \{(\mathtt{G}', \mathtt{G}), \mathtt{C.m} : \mathtt{G}(\bar{\mathtt{G}}),\ \mathtt{C.m}^{\mathtt{a}} : \mathtt{G}(\bar{\mathtt{G}}),\ \varepsilon\}, \textit{ or} \\ \mathtt{C.m}^{\mathtt{g}} : \mathtt{G}(\bar{\mathtt{G}}); \gamma'' \textit{ and } \gamma_0' = (\mathtt{G}', \mathtt{G}); \gamma''. \end{cases}
$$

for some γ_0' *s.t.* $\gamma' \leq_\alpha \gamma_0'$, *and for some* γ'', $\mathtt{C}, \mathtt{m}, \mathtt{G}, \mathtt{G}', \bar{\mathtt{G}}$

Lemma 3. *Let* $\mathsf{E}[\mathtt{r}]$ *be a subexpression of a contractually correct program. If* $\Gamma \vdash \mathsf{E}[\mathtt{r}] : (\mathtt{T}, \mathtt{G}), \gamma$ *and* $\Gamma \vdash \mathtt{r} : (\mathtt{T}', \mathtt{G}'), \gamma'$ *and* $\Gamma \vdash \mathtt{v} : (\mathtt{T}'', \mathtt{G}'), \varepsilon$, *with* $\mathtt{T}'' \leq \mathtt{T}'$. *Then* $\Gamma \vdash \mathsf{E}[\mathtt{v}] : (\mathtt{T}''', \mathtt{G}), \gamma''$ *such that* $\mathtt{T}''' \leq \mathtt{T}$ *and* $\gamma'' \lessdot \gamma$.

Proof. By induction on evaluation contexts E.

Case [ ]. Immediate.

Case $\mathsf{E}!\mathtt{m}(\bar{\mathtt{e}})$. Let $\Gamma \vdash \mathsf{E}[\mathtt{r}]!\mathtt{m}(\bar{\mathtt{e}}) : (\mathtt{T}, \mathtt{G}), \gamma$ where $\Gamma \vdash \mathtt{r} : (\mathtt{T}', \mathtt{G}'), \gamma'$, and let $\mathtt{v}$ be such that $\Gamma \vdash \mathtt{v} : (\mathtt{T}'', \mathtt{G}'), \varepsilon$, with $\mathtt{T}'' \leq \mathtt{T}'$. By typing rule (T-Invk) we have that

$$
\begin{array}{l}
\Gamma \vdash \mathsf{E}[\mathtt{r}] : (\mathtt{C}, \mathtt{G}_0), \gamma_0 \quad \Gamma \vdash \bar{\mathtt{e}} : (\bar{\mathtt{D}}, \bar{\mathtt{G}}), \bar{\gamma} \quad \circlearrowleft \notin \mathtt{G}_0\bar{\mathtt{G}} \\
\textit{mtype}(\mathtt{m}, \mathtt{C}) = \bar{\mathtt{C}} \to \mathtt{C}' \quad \bar{\mathtt{D}} <: \bar{\mathtt{C}} \\
\Gamma(\mathtt{C.m}) = \mathtt{G}_0'(\bar{\mathtt{G}}') \to \mathtt{G}'' \quad \mathtt{G} = \textit{fresh}(\mathtt{G}_0'\bar{\mathtt{G}}', \mathtt{G}''[^{\mathtt{G}_0\bar{\mathtt{G}}}/_{\mathtt{G}_0'\bar{\mathtt{G}}'}]) \\
\mathtt{T} = \mathtt{Fut}(\mathtt{C}') \quad \gamma = \gamma_0; (\mathtt{seq}(\bar{\gamma})); \mathtt{C.m} : \mathtt{G}_0(\bar{\mathtt{G}})
\end{array}
$$

By induction hypothesis on E we have that $\Gamma \vdash \mathsf{E}[\mathtt{v}] : (\mathtt{D}, \mathtt{G}_0), \gamma_0'$ for some $\mathtt{D} \leq \mathtt{C}$ and γ_0' such that:

- $\gamma_0 = \alpha; \gamma_0'$, where $\alpha \in \{\mathtt{C}''.\mathtt{n} : \mathtt{F}_0(\bar{\mathtt{F}}),\ \mathtt{C}''.\mathtt{n}^{\mathtt{a}} : \mathtt{F}_0(\bar{\mathtt{F}}),\ \varepsilon\}$, or
- $\gamma_0 = \mathtt{C}''.\mathtt{n}^{\mathtt{g}} : \mathtt{F}_0(\bar{\mathtt{F}}); \gamma'''$ and $\gamma_0' = (\mathtt{F}', \mathtt{F}_0); \gamma'''$,

for some $\mathtt{C}'', \mathtt{F}_0, \mathtt{F}', \bar{\mathtt{F}}$ and γ''', by Definition 3. Because of the correctness of the program and the consequent consistency of the class table, if $\mathtt{m}$ is defined in $\mathtt{D}$, then $\textit{mtype}(\mathtt{m}, \mathtt{C}) = \textit{mtype}(\mathtt{m}, \mathtt{D})$ and $\Gamma(\mathtt{C.m}) =^{\alpha} \Gamma(\mathtt{D.m})$. Applying typing rule (T-Invk) we have that $\Gamma \vdash \mathsf{E}[\mathtt{v}]!\mathtt{m}(\bar{\mathtt{e}}) : (\mathtt{Fut}(\mathtt{C}'), \mathtt{G}), \gamma_1$, where $\gamma_1 = \gamma_0'; (\mathtt{seq}(\bar{\gamma})); \mathtt{D.m} : \mathtt{G}_0(\bar{\mathtt{G}})$ and, by Definition 3, $\gamma_1 \lessdot \gamma$.

Case $\mathtt{o.m}(\bar{\mathtt{v}}, \mathsf{E}, \bar{\mathtt{e}})$. Let $\Gamma \vdash \mathtt{o.m}(\bar{\mathtt{v}}, \mathsf{E}[\mathtt{r}], \bar{\mathtt{e}}) : (\mathtt{T}, \mathtt{G}), \gamma$ where $\Gamma \vdash \mathtt{r} : (\mathtt{T}', \mathtt{G}'), \gamma'$, and let $\mathtt{v}$ be such that $\Gamma \vdash \mathtt{v} : (\mathtt{T}'', \mathtt{G}'), \varepsilon$, with $\mathtt{T}'' \leq \mathtt{T}'$. By typing rule (T-Invk) we have that

$$
\begin{array}{l}
\Gamma \vdash \mathtt{o} : (\mathtt{C}, \mathtt{G}_0), \varepsilon \quad \Gamma \vdash \bar{\mathtt{v}} : (\bar{\mathtt{D}}, \bar{\mathtt{G}}), \varepsilon \quad \Gamma \vdash \bar{\mathtt{e}} : (\bar{\mathtt{D}}', \bar{\mathtt{G}}'), \bar{\gamma} \quad \Gamma \vdash \mathsf{E}[\mathtt{r}] : (\mathtt{D}, \mathtt{F}), \gamma_0 \quad \circlearrowleft \notin \mathtt{G}_0\bar{\mathtt{G}} \\
\textit{mtype}(\mathtt{m}, \mathtt{C}) = \bar{\mathtt{C}} \to \mathtt{C}' \quad \bar{\mathtt{D}}\mathtt{D}\bar{\mathtt{D}}' <: \bar{\mathtt{C}} \\
\Gamma(\mathtt{C.m}) = \mathtt{G}_0'(\bar{\mathtt{G}}') \to \mathtt{G}'' \quad \mathtt{G} = \textit{fresh}(\mathtt{G}_0'\bar{\mathtt{G}}', \mathtt{G}''[^{\mathtt{G}_0\bar{\mathtt{G}}}/_{\mathtt{G}_0'\bar{\mathtt{G}}'}]) \\
\mathtt{T} = \mathtt{Fut}(\mathtt{C}') \quad \gamma = \gamma_0; (\mathtt{seq}(\bar{\gamma})); \mathtt{C.m} : \mathtt{G}_0(\bar{\mathtt{G}})
\end{array}
$$

By induction hypothesis on E we have that $\Gamma \vdash \mathsf{E}[\mathsf{v}] : (\mathtt{D}', \mathtt{F}), \gamma_0'$ for some $\mathtt{D}' \leq \mathtt{D}$ and γ_0' such that:

- $\gamma_0 = \alpha; \gamma_0'$, where $\alpha \in \{\mathtt{C}''.\mathtt{n} : \mathtt{F}_0(\bar{\mathtt{F}}),\ \mathtt{C}''.\mathtt{n}^{\mathtt{a}} : \mathtt{F}_0(\bar{\mathtt{F}}),\ \varepsilon\}$, or
- $\gamma_0 = \mathtt{C}''.\mathtt{n}^{\mathtt{g}} : \mathtt{F}_0(\bar{\mathtt{F}}); \gamma'''$ and $\gamma_0' = (\mathtt{F}', \mathtt{F}_0); \gamma'''$.

for some $\mathtt{C}''$, $\mathtt{F}_0$, $\mathtt{F}'$, $\bar{\mathtt{F}}$ and γ''', by Definition 3. Applying typing rule (T-Invk) we have that $\Gamma \vdash \mathsf{E}[\mathsf{v}]!\mathtt{m}(\bar{\mathsf{e}}) : (\mathtt{Fut}(\mathtt{C}'), \mathtt{G}), \gamma_1$, where $\gamma_1 = \gamma_0'; (\mathtt{seq}(\bar{\gamma})); \mathtt{D.m} : \mathtt{G}_0(\bar{\mathtt{G}})$ and, by Definition 3, $\gamma_1 \lessdot \gamma$.

Remaining Cases. Similar to the previous ones. □

Lemma 4. *Let* $\Gamma \vdash \mathtt{e} : \mathtt{T}$*. If* $\Gamma \subseteq \Gamma'$*, then* $\Gamma' \vdash \mathtt{e} : \mathtt{T}$

Proof. Straightforward induction on the derivation of $\Gamma \vdash \mathtt{e} : \mathtt{T}$.□

Lemma 5 (Substitution). *If* $\Gamma, \bar{\mathtt{x}} : \bar{\mathtt{C}} \vdash \mathtt{e} : \mathtt{C}$*, and* $\Gamma \vdash \bar{\mathtt{e}} : \bar{\mathtt{D}}$ *where* $\bar{\mathtt{D}} \leq \bar{\mathtt{C}}$*, then* $\Gamma \vdash \mathtt{e}[^{\mathtt{e}}/_{\mathtt{x}}] : \mathtt{D}$*, for some* $\mathtt{D} \leq \mathtt{C}$*.*

Proof. By straightforward induction on the derivation of $\Gamma, \bar{\mathtt{x}} : \bar{\mathtt{C}} \vdash \mathtt{e} : \mathtt{C}$.□

Lemma 6. *If mtype*$(\mathtt{m}, \mathtt{C}_0) = \bar{\mathtt{D}} \to \mathtt{D}$*, and mbody*$(\mathtt{m}, \mathtt{C}_0) = \bar{\mathtt{x}}.\mathtt{e}$*, then, for some* $\mathtt{D}_0$*, s.t.* $\mathtt{C}_0 \leq \mathtt{D}_0$*, there exists* $\mathtt{C} \leq \mathtt{D}$ *s.t.* $\bar{\mathtt{x}} : \bar{\mathtt{D}}, \mathtt{this} : \mathtt{D}_0 \vdash \mathtt{e} : \mathtt{C}$

Proof. By straightforward induction on the derivation of *mbody*$(\mathtt{m}, \mathtt{C}_0)$.□

Theorem 1 (Subject reduction).

1. *If* $\vdash (\textsc{ct}, \mathtt{e}) : (\textsc{cct}, \gamma)$ *then the initial configuration of* $(\textsc{ct}, \mathtt{e})$ *is contractually correct. Namely, there is* Γ *such that* $\Gamma \vdash (\mathsf{H} \Vdash \mathtt{t} :_{\mathtt{o}}^{\top} \mathtt{e}[^{\mathtt{o}}/_{\mathtt{this}}])$*, where* $\Gamma = \mathtt{o} \mapsto (\mathtt{Object}, \mathtt{G}), \mathtt{t} \mapsto (\mathtt{Fut}(\mathtt{C}), \mathtt{G})$ *and* $\mathsf{H} = \mathtt{o} \mapsto (\mathtt{Object}, \mathtt{G}, [\ ]), \mathtt{G} \mapsto \top$.
2. *Let* $\Gamma \vdash (\mathsf{H} \Vdash \mathsf{S})$ *and* $\mathsf{H} \Vdash \mathsf{S} \longrightarrow \mathsf{H}' \Vdash \mathsf{S}'$*. Then there is* Γ' *such that* $\Gamma' \vdash (\mathsf{H}' \Vdash \mathsf{S}')$*. In particular,* $\mathtt{t} : \mathtt{e} \in \mathsf{S}$ *such that* $\Gamma \vdash \mathtt{e} : (\mathtt{T}, \mathtt{G}), \gamma$*, implies* $\mathtt{t} : \mathtt{e}' \in \mathsf{S}'$ *such that* $\Gamma' \vdash \mathtt{e}' : (\mathtt{T}', \mathtt{G}), \gamma'$*, where* $\mathtt{T}' \leq \mathtt{T}$ *and* $\gamma' \lessdot \gamma$*.*

Proof. 1. Immediate by rule (T-Program).

2. The proof is by case analysis on the rule of the operational semantics rule used.

Case (Invk)**.** Then

$$\mathsf{H} \Vdash \mathtt{t} :_{\mathtt{o}}^{\top} \mathsf{E}[\mathtt{o}'!\mathtt{m}(\bar{\mathtt{v}})] \longrightarrow \mathsf{H} \Vdash \mathtt{t} :_{\mathtt{o}}^{\top} \mathsf{E}[\mathtt{t}'],\quad \mathtt{t}' :_{\mathtt{o}'}^{\perp} \mathtt{e}[^{\mathtt{o}'}/_{\mathtt{this}}][^{\bar{\mathtt{v}}}/_{\bar{\mathtt{x}}}]$$

where *class*$(\mathsf{H}, \mathtt{o}') = \mathtt{C}$, *mbody*$(\mathtt{m}, \mathtt{C}) = \bar{\mathtt{x}}.\mathtt{e}$, and $\mathtt{t}' \neq \mathtt{t}$.
Since $\Gamma \vdash (\mathsf{H} \Vdash (\mathtt{t} :_{\mathtt{o}}^{\top} \mathsf{E}[\mathtt{o}'!\mathtt{m}(\bar{\mathtt{v}})]))$, by rule (T-Configuration) we get $\Gamma \vdash \mathsf{E}[\mathtt{o}'!\mathtt{m}(\bar{\mathtt{v}})] : (\mathtt{D}, \mathtt{F}), \gamma$, for some $\mathtt{D}$, $\mathtt{F}$ and γ. By lemma 2 and typing rule (T-Invk), $\Gamma \vdash \mathtt{o}'!\mathtt{m}(\bar{\mathtt{v}}) : (\mathtt{Fut}(\mathtt{C}'), \mathtt{G}'''), \mathtt{C.m} : \mathtt{G}(\bar{\mathtt{G}})$, such that:

$$\Gamma \vdash \mathtt{o}' : (\mathtt{C}, \mathtt{G}), \varepsilon \quad \Gamma \vdash \bar{\mathtt{v}} : (\bar{\mathtt{D}}, \bar{\mathtt{G}}), \bar{\varepsilon} \quad \circlearrowleft \notin \mathtt{G}\bar{\mathtt{G}}$$
$$\mathit{mtype}(\mathtt{m}, \mathtt{C}) = \bar{\mathtt{C}} \to \mathtt{C}' \quad \bar{\mathtt{D}} <: \bar{\mathtt{C}}$$
$$\Gamma(\mathtt{C.m}) = \mathtt{G}'(\bar{\mathtt{G}}') \to \mathtt{G}'' \quad \mathtt{G}''' = \mathit{fresh}(\mathtt{G}'\bar{\mathtt{G}}', \mathtt{G}''[^{\mathtt{G}\bar{\mathtt{G}}}/_{\mathtt{G}'\bar{\mathtt{G}}'}]),$$

for some $\mathtt{G}, \bar{\mathtt{D}}, \bar{\mathtt{G}}$. By Lemmas 5 and 6, we get $\Gamma \vdash \mathtt{e}[^{\mathtt{o}'}/_{\mathtt{this}}][^{\bar{\mathtt{v}}}/_{\bar{\mathtt{x}}}] : (\mathtt{T}', \mathtt{G}''), \gamma'$, with $\mathtt{T}' \leq \mathtt{C}'$, for some γ'. Let Γ' be such that $\Gamma' = \Gamma, \mathtt{t} : (\mathtt{Fut}(\mathtt{T}'), \mathtt{G}''), \varepsilon$, then we can have $\Gamma' \vdash \mathtt{t}' : (\mathtt{Fut}(\mathtt{T}'), \mathtt{G}''), \varepsilon$. By lemma 4 we have $\Gamma' \vdash \mathtt{e}[^{\mathtt{o}'}/_{\mathtt{this}}][^{\bar{\mathtt{v}}}/_{\bar{\mathtt{x}}}] : (\mathtt{T}', \mathtt{G}''), \gamma'$ and $\Gamma' \vdash \mathsf{E}[\mathtt{o}'!\mathtt{m}(\bar{\mathtt{v}})] : (\mathtt{D}, \mathtt{F}), \gamma$ and $\Gamma' \vdash \mathtt{o}'!\mathtt{m}(\bar{\mathtt{v}}) : (\mathtt{Fut}(\mathtt{C}'), \mathtt{G}'''), \mathtt{C.m} : \mathtt{G}(\bar{\mathtt{G}})$. Notice that if $\mathtt{G}'' \in \mathtt{G}'\bar{\mathtt{G}}'$ then $\mathtt{G}'' = \mathtt{G}'''$ by definition of *fresh*, otherwise if $\mathtt{G}''$ is fresh we can choose for $\mathtt{G}'''$ the same fresh name such that $\mathtt{G}'' = \mathtt{G}'''$. By lemma 3 and rule (T-Configuration) we get the result.

Case (GET). Then

$$\mathsf{H} \Vdash \mathtt{t} :^{\mathsf{T}}_{\mathsf{o}} \mathsf{E}[\mathtt{t}'.\mathtt{get}],\ \mathtt{t}' :_{\mathsf{o}'} \mathsf{v} \longrightarrow \mathsf{H} \Vdash \mathtt{t} :^{\mathsf{T}}_{\mathsf{o}} \mathsf{E}[\mathsf{v}],\ \mathtt{t}' :_{\mathsf{o}'} \mathsf{v}$$

Since $\Gamma \vdash (\mathsf{H} \Vdash \mathtt{t} :^{\mathsf{T}}_{\mathsf{o}} \mathsf{E}[\mathtt{t}'.\mathtt{get}],\ \mathtt{t}' :_{\mathsf{o}'} \mathsf{v})$, by rule (T-CONFIGURATION) we get $\Gamma \vdash \mathsf{E}[\mathtt{t}'.\mathtt{get}] : (\mathtt{D}, \mathtt{F}), \gamma$, and $\Gamma \vdash \mathsf{v} : (\mathtt{D}', \mathtt{F}'), \varepsilon$ for some $\mathtt{D}$, $\mathtt{F}$, $\mathtt{D}'$, $\mathtt{F}'$ and γ. By lemma 2 and typing rule (T-TASK), $\Gamma \vdash \mathtt{t}'.\mathtt{get} : (\mathtt{D}', \mathtt{F}'), (group(\mathsf{H}, \mathsf{o}), \mathtt{F}')$. By lemma 3 and rule (T-CONFIGURATION) we get the result.

Remaining cases. Straightforward. □

The `dla` algorithm is correct, that is, if its result contains a group-circularity, then the evaluation of the analyzed expression may manifest a deadlock (*vice versa*, if there is no group-circularity then no deadlock will be ever manifested). To demonstrate this result we need to define the set of dependencies of configurations.

Definition 4. *Let* $\Gamma \vdash (\mathsf{H} \Vdash \mathsf{S})$*, then*

$$\mathtt{dla}(\Delta_{\mathrm{CCT}}, \mathsf{H} \Vdash \mathsf{S}) \stackrel{def}{=} \{\mathtt{t} : \mathtt{dla}(\Delta_{\mathrm{CCT}}, \mathtt{G}, \gamma) \mid \mathtt{t} :_{\mathsf{o}} \mathsf{e} \in \mathsf{S} \text{ and } \mathtt{G} = group(\mathsf{H}, \mathsf{o}) \text{ and } \Gamma \vdash \mathsf{e} : (\mathtt{T}, \mathtt{G}'), \gamma\ \}$$

Moreover, we add the following rule to the ones in table 6:

(DLA-TASK)

$$\mathtt{dla}(\Delta_{\mathrm{CCT}}, \mathtt{G}, (\mathtt{G}', \mathtt{G}''); \gamma) = \{(\mathtt{G}', \mathtt{G}'')\}; \mathtt{dla}(\Delta_{\mathrm{CCT}}, \mathtt{G}, \gamma)$$

and we remind that contracts $(\mathtt{G}', \mathtt{G}'')$ are used in the judgment of the runtime expressions `t.get`.

Definition 5 ($\leq$)**.** *Let* $\boldsymbol{G}_1; \cdots; \boldsymbol{G}_m \leq \overline{\boldsymbol{G}}; \boldsymbol{G}'_1; \cdots; \boldsymbol{G}'_m$ *whenever, for every* i, $\boldsymbol{G}_i \subseteq \boldsymbol{G}'_i$

Lemma 7. *(i)* $\gamma' \leq_\alpha \gamma$ *implies* $\mathtt{dla}(\Delta_{\mathrm{CCT}}, \mathtt{G}, \gamma') = \mathtt{dla}(\Delta_{\mathrm{CCT}}, \mathtt{G}, \gamma)$.
(ii) $\gamma' \lessdot \gamma$ *implies* $\mathtt{dla}(\Delta_{\mathrm{CCT}}, \mathtt{G}, \gamma') \leq \mathtt{dla}(\Delta_{\mathrm{CCT}}, \mathtt{G}, \gamma)$.

Proof. (i) Immediate by definition of $\leq_\alpha$ and contractually correct program.
(ii) By case analysis on the definition of $\gamma' \lessdot \gamma$.
If $\gamma' \leq_\alpha \gamma$. Immediate.
If $\gamma = (\mathtt{G}, \mathtt{G}'); \gamma'_0$ and $\gamma' \leq_\alpha \gamma'_0$. Immediate from rule (DLA-TASK) and *(i)*.
If $\gamma = \mathtt{C.m} : \mathtt{G}'(\bar{\mathtt{G}}); \gamma'_0$ and $\gamma' \leq_\alpha \gamma'_0$. Immediate from rule (DLA-INVK) and *(i)*.
If $\gamma = \mathtt{C.m}^{\mathtt{a}} : \mathtt{G}'(\bar{\mathtt{G}}); \gamma'_0$ and $\gamma' \leq_\alpha \gamma'_0$. Immediate from rule (DLA-AWAIT) and *(i)*.
If $\gamma = \mathtt{C.m}^{\mathtt{g}} : \mathtt{G}'(\bar{\mathtt{G}}); \gamma''$ and $\gamma'_0 = (\mathtt{G}, \mathtt{G}'); \gamma''$ and $\gamma' \leq_\alpha \gamma'_0$. From rule (DLA-GET) we have $\mathtt{dla}(\Delta_{\mathrm{CCT}}, \mathtt{G}, \gamma) = \{(\mathtt{G}, \mathtt{G}')\} \cup \boldsymbol{G}; \overline{\boldsymbol{G}}$, for some $\boldsymbol{G}$, and from rule (DLA-TASK) we have $\mathtt{dla}(\Delta_{\mathrm{CCT}}, \mathtt{G}, \gamma'_0) = \{(\mathtt{G}, \mathtt{G}')\}; \overline{\boldsymbol{G}}$. Then the result follows immediately by Definition 5 and *(i)*.□

Definition 6. *A group substitution* σ *is a finite mapping from group names to group names, that is* $\sigma(\mathtt{G}) = \mathtt{G}'$ *if* $(\mathtt{G} \mapsto \mathtt{G}') \in \sigma$*. Application to sets of group dependencies is defined in the obvious way:*

$$\sigma((\mathtt{G}_1, \mathtt{G}_2)) = (\sigma(\mathtt{G}_1), \sigma(\mathtt{G}_2)) \qquad \sigma(\boldsymbol{G}_1; \cdots; \boldsymbol{G}_n) = \sigma(\boldsymbol{G}_1); \cdots; \sigma(\boldsymbol{G}_n)$$

Lemma 8. *Given a set of dependencies* $\boldsymbol{G}$ *and a substitution* σ*, if* $\boldsymbol{G}$ *has group-circularities then* $\sigma(\boldsymbol{G})$ *has group-circularities. That is substituions preserve group-circularities.*

Proof. By induction on $\boldsymbol{G}$.

Case $\boldsymbol{G} = (\mathtt{G}, \mathtt{G})$. Immediate.

Case $\boldsymbol{G} = \boldsymbol{G}', (\mathsf{G}, \mathsf{G}')$. If $\boldsymbol{G}'$ has circularities then the result follows from induction hypothesis. Otherwise, if the circularity is introduced by $(\mathsf{G}, \mathsf{G}')$, this means that $\boldsymbol{G}'' \subseteq \boldsymbol{G}'$ is such that the transitive closure of $\boldsymbol{G}'' \cup \{(\mathsf{G}, \mathsf{G}')\}$ is a pair of the form $(\mathsf{G}'', \mathsf{G}'')$, for some $\mathsf{G}'' \in \{\mathsf{G}, \mathsf{G}'\} \cup \mathit{names}(\boldsymbol{G}')$. Therefore since the substitution preserves identity, we get that also $\boldsymbol{G}$ has a circularity. □

Theorem 2. *1. If* $\vdash (\textsc{ct}, \mathtt{e}) : (\textsc{cct}, \gamma)$ *then* $\mathtt{dla}(\Delta_{\mathrm{CCT}}, \mathsf{G}, \gamma) = \mathtt{dla}(\Delta_{\mathrm{CCT}}, \mathsf{H} \Vdash \mathtt{t} :^{\top}_{\mathsf{o}} \mathtt{e}[^{\mathsf{o}}/_{\mathtt{this}}])$, *where* $\mathsf{H} = \mathsf{o} \mapsto (\mathtt{Object}, \mathsf{G}, [\,]), \mathsf{G} \mapsto \top$.

2. *Let* $\Gamma \vdash (\mathsf{H} \Vdash \mathsf{S})$ *and* $\mathsf{H} \Vdash \mathsf{S} \longrightarrow \mathsf{H}' \Vdash \mathsf{S}'$. *Then*
   (i) $\mathtt{t} : \overline{\boldsymbol{G}} \in \mathtt{dla}(\Delta_{\mathrm{CCT}}, \mathsf{H} \Vdash \mathsf{S})$ *implies* $\mathtt{t} : \overline{\boldsymbol{G}}' \in \mathtt{dla}(\Delta_{\mathrm{CCT}}, \mathsf{H}' \Vdash \mathsf{S}')$ *and* $\overline{\boldsymbol{G}}' \preceq \overline{\boldsymbol{G}}$*;*
   (ii) $\mathtt{t}' : \boldsymbol{G}'_1; \cdots; \boldsymbol{G}'_n \in \mathtt{dla}(\Delta_{\mathrm{CCT}}, \mathsf{H}' \Vdash \mathsf{S}')$ *and there is no task* $\mathtt{t}'$ *in* S *(*$\mathtt{t}'$ *has been created by the reduction) then there is either* $\mathtt{t} : \boldsymbol{G}; \overline{\boldsymbol{G}} \in \mathtt{dla}(\Delta_{\mathrm{CCT}}, \mathsf{H} \Vdash \mathsf{S})$ *such that* $\mathtt{t} : \boldsymbol{G}'; \overline{\boldsymbol{G}} \in \mathtt{dla}(\Delta_{\mathrm{CCT}}, \mathsf{H} \Vdash \mathsf{S}')$ *or* $\mathtt{t} : \boldsymbol{G} \cup \overline{\boldsymbol{G}} \in \mathtt{dla}(\Delta_{\mathrm{CCT}}, \mathsf{H} \Vdash \mathsf{S})$ *such that* $\mathtt{t} : \boldsymbol{G}' \cup \overline{\boldsymbol{G}} \in \mathtt{dla}(\Delta_{\mathrm{CCT}}, \mathsf{H} \Vdash \mathsf{S}')$. *In both cases* $\mathsf{G} = \sigma(\bigcup_i \boldsymbol{G}_i \cup \boldsymbol{G}')$.

Proof. 1. Immediate by Definition 4.

2. Case *(i)* The result follows from Theorem 1 and Lemma 7.

Case *(ii)* The only reduction rule that creates a new task is rule (Invk):

$$\mathsf{H} \Vdash \mathtt{t} :^{\top}_{\mathsf{o}} \mathsf{E}[\mathsf{o}'!\mathtt{m}(\bar{\mathtt{v}})] \longrightarrow \mathsf{H} \Vdash \mathtt{t} :^{\top}_{\mathsf{o}} \mathsf{E}[\mathtt{t}'],\ \ \mathtt{t}' :^{\perp}_{\mathsf{o}'} \mathtt{e}[^{\mathsf{o}'}/_{\mathtt{this}}][^{\bar{\mathtt{v}}}/_{\bar{\mathtt{x}}}]$$

where $\mathit{class}(\mathsf{H}, \mathsf{o}') = \mathsf{C}$, $\mathit{mbody}(\mathtt{m}, \mathsf{C}) = \bar{\mathtt{x}}.\mathtt{e}$, and $\mathtt{t}' \neq \mathtt{t}$.
If $\mathsf{E} = \mathsf{E}'[[].\mathtt{get}]$, from Lemmas 2, typing rules (T-Invk) and (T-Get), and Theorem 1, we have $\Gamma \vdash \mathsf{E}[\mathtt{o.m}(\bar{\mathtt{v}})] : (\mathtt{T}, \mathsf{G}), (\mathsf{C}.\mathtt{m}^{\mathtt{g}} : \mathsf{G}(\bar{\mathsf{G}})); \gamma'$ and $\Gamma \vdash \mathtt{e} : (\mathtt{T}', \mathsf{G}'), \gamma$, being $\textsc{cct}(\mathsf{C})(\mathtt{m}) = \mathtt{F}(\bar{\mathtt{F}})\{\gamma\}\mathtt{F}'$. If $\mathtt{dla}(\Delta_{\mathrm{CCT}}, \mathtt{F}, \gamma) = \boldsymbol{G}_1; \cdots; \boldsymbol{G}_n$, then $\Delta_{\mathrm{CCT}}(\mathsf{C}.\mathtt{m}) = (\mathtt{F}, \bar{\mathtt{F}}) \bigcup_{i \in 1..n} \boldsymbol{G}_i$, by definition. From rule (dla-Get) we get $\mathtt{dla}(\Delta_{\mathrm{CCT}}, \mathsf{G}', (\mathsf{C}.\mathtt{m}^{\mathtt{g}} : \mathsf{G}(\bar{\mathsf{G}})); \gamma') = (\{\mathsf{G}', \mathsf{G}\} \cup \bigcup_{i \in 1..n} \boldsymbol{G}_i[^{\mathsf{G}; \bar{\mathsf{G}}}/_{\mathtt{F}; \bar{\mathtt{F}}}]; \mathtt{dla}(\Delta_{\mathrm{CCT}}, \mathtt{F}, (\mathsf{C}.\mathtt{m} : \mathsf{G}(\bar{\mathsf{G}})); \gamma')$. From rule (dla-Task) we get the result.
If $\mathsf{E} = \mathsf{E}'[[].\mathtt{await}]$, from Lemmas 2, typing rules (T-Invk) and (T-Await), and Theorem 1, we have $\Gamma \vdash \mathsf{E}[\mathtt{o.m}(\bar{\mathtt{v}})] : (\mathtt{T}, \mathsf{G}), (\mathsf{C}.\mathtt{m}^{\mathtt{a}} : \mathsf{G}(\bar{\mathsf{G}})); \gamma'$ and $\Gamma \vdash \mathtt{e} : (\mathtt{T}', \mathsf{G}'), \gamma$, being $\textsc{cct}(\mathsf{C})(\mathtt{m}) = \mathtt{F}(\bar{\mathtt{F}})\{\gamma\}\mathtt{F}'$. If $\mathtt{dla}(\Delta_{\mathrm{CCT}}, \mathtt{F}, \gamma) = \boldsymbol{G}_1; \cdots; \boldsymbol{G}_n$, then $\Delta_{\mathrm{CCT}}(\mathsf{C}.\mathtt{m}) = (\mathtt{F}, \bar{\mathtt{F}}) \bigcup_{i \in 1..n} \boldsymbol{G}_i$, by definition. From rule (dla-Await) we get $\mathtt{dla}(\Delta_{\mathrm{CCT}}, \mathsf{G}', (\mathsf{C}.\mathtt{m}^{\mathtt{a}} : \mathsf{G}(\bar{\mathsf{G}})); \gamma') = \bigcup_{i \in 1..n} \boldsymbol{G}_i[^{\mathsf{G}; \bar{\mathsf{G}}}/_{\mathtt{F}; \bar{\mathtt{F}}}]; \mathtt{dla}(\Delta_{\mathrm{CCT}}, \mathtt{F}, (\mathsf{C}.\mathtt{m} : \mathsf{G}(\bar{\mathsf{G}})); \gamma')$. From rule (dla-Task) we get the result.
Otherwise, from Lemmas 2, typing rule (T-Invk) and Theorem 1, we have $\Gamma \vdash \mathsf{E}[\mathtt{o.m}(\bar{\mathtt{v}})] : (\mathtt{T}, \mathsf{G}), (\mathsf{C}.\mathtt{m} : \mathsf{G}(\bar{\mathsf{G}})); \gamma'$ and $\Gamma \vdash \mathtt{e} : (\mathtt{T}', \mathsf{G}'), \gamma$, being $\textsc{cct}(\mathsf{C})(\mathtt{m}) = \mathtt{F}(\bar{\mathtt{F}})\{\gamma\}\mathtt{F}'$. If $\mathtt{dla}(\Delta_{\mathrm{CCT}}, \mathtt{F}, \gamma) = \boldsymbol{G}_1; \cdots; \boldsymbol{G}_n$, then $\Delta_{\mathrm{CCT}}(\mathsf{C}.\mathtt{m}) = (\mathtt{F}, \bar{\mathtt{F}}) \bigcup_{i \in 1..n} \boldsymbol{G}_i$, by definition. From rule (dla-Invk) we get $\mathtt{dla}(\Delta_{\mathrm{CCT}}, \mathsf{G}', (\mathsf{C}.\mathtt{m} : \mathsf{G}(\bar{\mathsf{G}})); \gamma') = \bigcup_{i \in 1..n} \boldsymbol{G}_i[^{\mathsf{G}; \bar{\mathsf{G}}}/_{\mathtt{F}; \bar{\mathtt{F}}}] \cup \mathtt{dla}(\Delta_{\mathrm{CCT}}, \mathtt{F}, \gamma')$. From rule (dla-Task) we get the result.
□

We say that $\{\mathtt{t}_1 : \overline{\boldsymbol{G}}_1, \cdots, \mathtt{t}_n : \overline{\boldsymbol{G}}_n\}$ is not circular if, for every i: $\overline{\boldsymbol{G}}_i$: `not-circular`. An immediate consequence of Theorem 2, and Lemma 8, is:

Corollary 1. *If* $\vdash (\textsc{ct}, \mathtt{e}) : (\textsc{cct}, \gamma)$ *and* $\mathtt{dla}(\Delta_{\mathrm{CCT}}, \mathsf{G}, \gamma)$ *is not circular then* $(\textsc{ct}, \mathtt{e})$ *is deadlock-free.*

7 Related works

The notion of grouping objects dates back at least to the mid 80'es with the works of Yonezawa on the language ABCL/1 [9, 24]. Since then, several languages have a notion of group for structuring systems, such as Eiffel// [4], Hybrid [18], and ASP [5]. A library for object groups has also been defined for CORBA [8]. In these proposals, a single task is responsible for executing the code inside a group. Therefore it is difficult to model behaviours such as waiting for messages without blocking the group for other activities.

Our FJg calculus is inspired to the language Creol that proposes object groups, called *JCoBoxes*, with multiple cooperatively scheduled tasks [11]. In particular FJg is a subcalculus of JCoBox^c in [20], where the emphasis was the definition of the semantics and the type system of the calculus and the implementation in Java.

The proposals for statically analyzing deadlocks are largely based on types [12,22,21,23]. Some work also addresses deadlocks in object-oriented programs [1,2]. In all these papers, 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. On the contrary, our technique does not computes any ordering of locks, thus being more flexible: a computation may acquire two locks in different order at different stages, thus being correct in our case, but incorrect with the other techniques. A further difference with the above works is that we use contracts that are terms in simple (= with finite states) process algebras [13]. The use of simple process algebras to describe (communication or synchronization) protocols is not new. This is the case of the exchange patterns in SSDL [19], which are based on CSP [3] and the pi-calculus [15], or of the behavioral types in [17] and [6], which use CCS [14]. We expect that finite state abstract descriptions of behaviors can support techniques that are more powerful than the one used in this contribution.

8 Conclusions

We have developed a technique for the deadlock analysis of object groups that is based on abstract descriptions of method?s behaviours.

This study can be extended in several directions. One direction is the total coverage of the full language FJg. This is possible by using *group records* $\Theta, \Theta' = \mathtt{G}[\mathtt{f}_1 : \Theta_1, \cdots, \mathtt{f}_k : \Theta_k]$ instead of simple group names. Then contracts such as $\mathtt{C.m} : \mathtt{G}(\bar{\mathtt{G}})$ become $\mathtt{C.m} : \Theta(\Theta_1, \cdots, \Theta_n)$ and the rule (T-Field) is refined into

(T-Field-ref)

$$\frac{\Gamma \vdash \mathtt{this} : (\mathtt{C}, \mathtt{G}[\bar{\mathtt{f}} : \overline{\Theta}]), \varepsilon \quad \mathtt{D}\,\mathtt{f} \in \mathit{fields}(\mathtt{C}) \quad \mathtt{f} : \Theta' \in \bar{\mathtt{f}} : \overline{\Theta}}{\Gamma \vdash \mathtt{this.f} : (\mathtt{D}, \Theta'), \varepsilon}$$

The overall effect of this extension is to hinder the notation used in the paper, without conveying any interesting difficulty (for this reason we have restricted our analysis to a sublanguage). We claim that every statement for plain FJg in this paper also hold for full FJg.

A different direction of research is the study of techniques for augmenting the accuracy of the algorithm dla, which is imprecise at the moment. The intent is to use finite state automata with name creation, such as those in [16], and modeling method contracts in terms of finite automata and study deadlocks in sets of these automata.

Other directions address extensions of the language FJg. One of these extensions is the removal of the constraint that future types cannot be used by programmers in FJg. Future types augment the expressivity of the language. For example it is possible to synchroniza several tasks and define *livelocks*:

```
class C { f: Fut(C) ;
     C m() { return this.f = this!n() ; new C() ;}
     C n() { return this.f.get ; new C() ;}   }
```

Another extension is about re-entrant method invocations (usually used for tail recursions), which are synchronous invocations. Such extension requires revisions of semantics rules, of the contract rules in Table 4, and of the `dla` algorithm.

Acknowledgements. We thank Davide Sangiorgi for the discussions about deadlocks. The authors are member of the joint FOCUS Research Team INRIA/Università di Bologna. This research has been funded by the EU project FP7-231620 HATS.

References

1. M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for java. *ACM Trans. Program. Lang. Syst.*, 28:207–255, March 2006.
2. C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '02, pages 211–230, New York, NY, USA, 2002. ACM.
3. S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31:560–599, June 1984.
4. D. Caromel. Towards a method of object-oriented concurrent programming. *Commun. ACM*, 36(9):90–102, 1993.
5. D. Caromel, L. Henrio, and B. P. Serpette. Asynchronous and deterministic objects. In *POPL'04 - Proceedings of the 31st Symp. on Principles of Programming Languages*, pages 123–134. ACM, 2004.
6. S. Chaki, S. K. Rajamani, and J. Rehof. Types as models: model checking message-passing programs. *SIGPLAN Not.*, 37(1):45–57, 2002.
7. F. de Boer, D. Clarke, and E. Johnsen. A complete guide to the future. In R. De Nicola, editor, *Programming Languages and Systems*, volume 4421 of *Lecture Notes in Computer Science*, pages 316–330. Springer Berlin / Heidelberg, 2007.
8. P. Felber and R. Guerraoui. Programming with object groups in corba. *IEEE Concurrency*, 8:48–58, January 2000.
9. Y. Honda and A. Yonezawa. Debugging concurrent systems based on object groups. In *ECOOP'88 - European Conference on Object-Oriented Programming*, volume 322 of *Lecture Notes in Computer Science*, pages 267–282. Springer, 1988.
10. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23:396–450, May 2001.
11. E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and System Modeling*, 6(1):39–58, 2007.
12. N. Kobayashi. A new type system for deadlock-free processes. In *17th International Conference on Concurrency Theory, CONCUR 2006*, volume 4137 of *Lecture Notes in Computer Science*, pages 233–247. Springer, 2006.
13. C. Laneve and L. Padovani. The *must* preorder revisited. In *CONCUR 2007 - 18th International Conference on Concurrency Theory*, volume 4703 of *Lecture Notes in Computer Science*, pages 212–225. Springer, 2007.
14. R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.
15. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, ii. *Inf. and Comput.*, 100:41–77, 1992.
16. U. Montanari and M. Pistore. History-dependent automata: An introduction. In *5th International School on Formal Methods for the Design of Computer, Communication, and Software Systems*, volume 3465 of *Lecture Notes in Computer Science*, pages 1–28. Springer, 2005.

17. H. R. Nielson and F. Nielson. Higher-order concurrent programs with finite communication topology (extended abstract). In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 84–97, New York, NY, USA, 1994. ACM Press.
18. O. Nierstrasz. Active objects in Hybrid. In *OOPSLA*, pages 243–253, 1987.
19. S. Parastatidis and J. Webber. *MEP SSDL Protocol Framework*, Apr. 2005. `http://ssdl.org`.
20. J. Schäfer and A. Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. In *ECOOP 2010 - 24th European Conference on Object-Oriented Programming*, volume 6183 of *Lecture Notes in Computer Science*, pages 275–299. Springer, 2010.
21. K. Suenaga. Type-based deadlock-freedom verification for non-block-structured lock primitives and mutable references. In G. Ramalingam, editor, *Programming Languages and Systems*, volume 5356 of *Lecture Notes in Computer Science*, pages 155–170. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-89330-1_12.
22. K. Suenaga and N. Kobayashi. Type-based analysis of deadlock for a concurrent calculus with interrupts. In R. De Nicola, editor, *Programming Languages and Systems*, volume 4421 of *Lecture Notes in Computer Science*, pages 490–504. Springer Berlin / Heidelberg, 2007. 10.1007/978-3-540-71316-6_33.
23. V. T. Vasconcelos, F. Martins, and T. Cogumbreiro. Type inference for deadlock detection in a multithreaded polymorphic typed assembly language. In *PLACES*, pages 95–109, 2009.
24. A. Yonezawa, J.-P. Briot, and E. Shibayama. Object-oriented concurrent programming in ABCL/1. In *OOPSLA*, pages 258–268, 1986.