Under consideration for publication

Deadlocks and Livelocks in Concurrent Objects with Futures

Elena Giachino, Cosimo Laneve, and Tudor Lascu

Dipartimento di Scienze dell'Informazione, Università di Bologna

Received 6 June 2012

We study `FJf`, a concurrent object calculus with future types and operations for getting the values and releasing the control. Programs in `FJf` may manifest locks (deadlocks or livelocks) due to badly programmed release points. In order to statically detect possible misbehaviours, we develop a technique for the lock analysis based on contracts, which are abstract descriptions of method's behaviours. Contracts are derived by a type inference algorithm and are modeled by finite state automata whose states retain information on caller-callee dependencies. A potential lock is detected when a circular dependency is found in some state of the automata.

1. Introduction

Concurrent object-oriented programming is a common model of concurrency that dates back to the 80-ies (Yonezawa, 1990; America et al., 1986) and is nowadays widely used by the mainstream programming languages (`Java`, `C#`. `C++`, `Objective C`, etc.). Since the beginning, prompted by the need of combining object-orientation with distributed programming and taking inspiration from Agha's Actors (Agha, 1986), method calls have been modeled as *asynchronous message sendings*, that is the caller continues executing *in parallel with* the called method. More recently, standard method invocations have been reintroduced in a "loosely-coupled" fashion by using *future types* (Liskov and Shrira, 1988; Niehren et al., 2006) and explicit *operations for getting the values of invocations and explicitly releasing the control.* In fact, a number of extensions that include these features have been designed, often as libraries, for the above popular languages `C++` (Lavender and Schmidt, 1995), `Java` (Welc et al., 2005), `C#`, Visual Basic and .NET (Torgersen, 2010), as well as novel prototypes have been proposed (Johnsen and Owe, 2007).

While explicit scheduling mechanisms (such as getting values, releasing the control, etc.) allow flexible patterns of synchronization that are attractive for optimizing purposes or for avoiding unneeded busy waitings, debugging programs using such mechanisms may be very difficult because of inconsistencies between release points in separate, yet cooperating, methods. 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 futures and operators for releasing the control and develop a technique for the analysis of deadlocks.

Our object-oriented language, called *Featherweight Java with futures*, `FJf` in brief, is an extension of Featherweight Java (Igarashi et al., 2001). In `FJf`, objects have multiple tasks in execution and there is at most one task per object that is active at each point in time. The active task may explicitly return the control in order to let another task of the same object progress. Tasks are created by method invocations: the caller activity continues after the invocation and the called code runs on a different task. The synchronization between the caller and the called methods is performed when the result is strictly necessary. In order to decouple method invocation and returned value, `FJf` uses *future variables*, 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 objects and explicit scheduling operation, a typical (dead- or live-)lock occurs when one or more tasks are waiting for each other termination to return a value. A simple circular dependency involves only one task as in the method

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

that defines the factorial function (for the sake of the example we include primitive types `Int` and conditional into the `FJf` syntax – see Section 3.1). In the body of `fact`, the recursive invocation `this!fact(n-1)` is postfixed by a `get` operation that retrieves the value returned by the invocation. Yet, `get` does not releases the lock of the caller object; therefore the task evaluating `this!fact(n-1)` is fated to be delayed forever because its object is the same of the caller.

Our main goal is the development of a technique for the static detection of deadlocks and livelocks in `FJf` programs. The technique we propose is based on *contracts*, which are abstract descriptions of behaviours that retain the necessary information to detect locks (Kobayashi, 2006; Laneve and Padovani, 2007). For example, the contract of `fact` (assuming the method belongs to a class `Maths` without fields) is a`[ ]()`{ `Maths.fact` a`[ ]().`(a,a) }. This contract declares that the invocation of `fact` on an object a will call (recursively) `fact` on the same object a and the invocation introduces an *object name dependency* (a, a). The dependency specifies that the object of the caller, *stored in the first element of the pair*, is released as soon as the callee gets and releases its own object, which is *stored in the second element of the pair*.

We define a type inference system for associating a contract to every method of the program and to the expression to evaluate. The type system is demonstrated to be sound with respect to the operational semantics of `FJf` – namely typing is preserved by transitions.

In order to statically detect potential locks in `FJf` programs, we introduce a finite model called *finite state automata for lock analysis*, *lafsa* in brief, whose states retain information on caller-callee dependencies, and whose transitions mimic the concrete transitions of the `FJf` semantics. In particular, every state of a *lafsa* is a relation on object names and a potential misbehaviour (a deadlock or a livelock) is signaled by the presence of a circularity in some state of its. We then define a *lafsa* semantics for contracts that, given the type systems for methods and expressions, allows one to associate a *lafsa* to every `FJf` program. For example, the model of the above method `fact` is a single state *lafsa* whose state is $\{(a, a)\}$. A pair as (a, a) in a state signals a *circular object name dependency*, which allows us to conclude that `fact` may manifest a lock – in this case a deadlock. In fact, the above `fact` is a wrong implementation of the factorial in `FJf`. (Correct implementations are discussed in Section 6.)

The key technical point of the contribution is the proof of correctness of our technique: a transition of a `FJf` program from state S to state S' is such that the states of the *lafsa* of S' contain less object name dependencies than the states of the *lafsa* of S. Said otherwise, correctness means that the precision of our technique does not decrease as the computation progresses. It is worth to notice that the *lafsa*s of S and S' are those of their contracts, which must exist for well-typed programs by the soundness of the type system.

```
class C extends Object {
     Object f ;
     C m() { return new C(this.f) ;}  }
class D extends C {
     C n(D c) { return (c!m()).await.get ;}  }
class E extends C {
     C n(E c) { return (c!m()).get ;}  }
```

Table 1. *Simple classes in* `FJf`

The paper is organized as follows. Section 2 introduces the main ideas of `FJf` by discussing few sample programs. Section 3 presents the syntax and operational semantics of `FJf`. Section 4 defines contracts and the type system for deriving contracts of expressions and methods. Section 5 introduces the model of our contracts – the *lafsa* – and defines the operations that are used in the semantics of contracts. Section 6 defines the semantics of contracts in terms of *lafsa*s and demonstrates its correctness – a program whose *lafsa* does not manifest a lock – *i.e.* an object name circularity – will never deadlock nor livelock. Section 7 surveys related works, and we give conclusions and indications of further work in Section 8.

2. `FJf` in a nutshell

In `FJf` a program is a collection of class definitions plus an expression to evaluate. A simple definition in `FJf` is the class `C` in Table 1 that defines a class with a field `f` and a method `m`. When `m` is invoked, a new object of class `C` is created (with `f` containing the same value of the creator) and returned. `FJf` also supports inheritance: the class `D` in Table 1 extends `C` with a method `n`.

Method invocations in `FJf` are *asynchronous*; for this reason we use the exclamation mark rather than the usual dot notation when methods are invoked – see the body of method `n` in Table 1.

`FJf` features explicit processor release points in method definitions, thus allowing the caller to decide the transfer of control at runtime. For example, in the scope of Table 1 declarations, the invocation

`x!n(x)`

where `x` is an object of class `D`, brings to executing the body of `m` on the object `x`. Since the caller method `n` and the method `m` called in the body of `n` share a same object, the code of `m` cannot be evaluated until the caller `n` explicitly releases the control. The `await` operation in `(c!m()).await.get` exactly lets the control be released. The following `get` operation is used for retrieving the value of the invocation, once the callee terminates.

The operations `await` and `get` permit very flexible patterns of synchronization. As usual, when flexibility grows safety reduces and `FJf` does not escape from this principle. For example, if `x` is an object of class `E` in Table 1, the above expression `x!n(x)` gets stuck because the task executing the body of `n` does not release the control on the object `x`. Therefore the body of `m` cannot be evaluated.

To detect the dangerous synchronization patterns as the one above, `FJf` uses behavioural types, called *contracts*. For example, the contract of the method `m` of Table 1 is derived using the rule

$$\frac{\Gamma + \texttt{this} : (\texttt{C}, a[\texttt{f} : X]) \vdash_a \texttt{new C(this.f)} \ : \ (\texttt{C}, b[\texttt{f} : X])\,,\ \texttt{0}}{\Gamma \vdash \ \texttt{C m ()\{return new C(this.f);\}} : a[\texttt{f} : X]()\{\texttt{0}\}\ b[\texttt{f} : X]\ \ \text{IN C}}$$

(which is an instance of (T-Method) in Table 5 where trivially true hypotheses are omitted). The contract $a[\texttt{f} : X]()\{\texttt{0}\}\ b[\texttt{f} : X]$ specifies the object receiver of `m`, namely $a[\texttt{f} : X]$, where a is the object name and X is the value of the field `f`, and the returned object $b[\texttt{f} : X]$, which has a different object name from a but the same value of the field. The contract also specifies the behaviour, which is empty ($= \texttt{0}$) in this case. The premise of the above rule has a judgment of the form $\Gamma \vdash_a \texttt{e} : (\texttt{T}, \mathbbm{r}),\ \mathbbm{c}$, where Γ is the environment, a is the object name of the method containing the expression `e`, `e` is a `FJf` expression, `T` is its (standard) type, $\mathbbm{r}$ is a *future record* that we explain in a while, and the contract $\mathbbm{c}$ bears information about caller-callee dependencies among object names.

The typing of `n` in Table 1 follows by the proof tree

$$\dfrac{\dfrac{\dfrac{\Gamma + \texttt{this} : (\texttt{D}, a[\texttt{f} : X]), \texttt{c} : (\texttt{D}, b[\texttt{f} : Y]) \vdash_a \texttt{c!m()} : (\texttt{Fut(C)}, b \rightsquigarrow b'[\texttt{f} : Y]),\ \texttt{D.m}\ b[\texttt{f} : Y]\texttt{()}}{\Gamma + \texttt{this} : (\texttt{D}, a[\texttt{f} : X]), \texttt{c} : (\texttt{D}, b[\texttt{f} : Y]) \vdash_a \texttt{c!m().await} : (\texttt{Fut(C)}, b \rightsquigarrow b'[\texttt{f} : Y]),\ \texttt{D.m}\ b[\texttt{f} : Y]\texttt{()} \centerdot (a, b)^{\mathtt{a}}}}{\Gamma + \texttt{this} : (\texttt{D}, a[\texttt{f} : X]), \texttt{c} : (\texttt{D}, b[\texttt{f} : Y]) \vdash_a \texttt{c!m().await.get} : (\texttt{C}, b'[\texttt{f} : Y]),\ \texttt{D.m}\ b[\texttt{f} : Y]\texttt{()} \centerdot (a, b)^{\mathtt{a}}}}{\Gamma \vdash\ \texttt{C n (D c)}\{\texttt{return c!m().await.get ;}\} : a[\texttt{f} : X](b[\texttt{f} : Y])\{\texttt{D.m}\ b[\texttt{f} : Y]\texttt{()} \to b'[\texttt{f} : Y] \centerdot (a, b)^{\mathtt{a}}\}\ b'[\texttt{f} : Y]\ \ \text{IN D}}$$

(the rules used are instances of (T-Await), (T-Get), and (T-Method) in Table 5 where, as before, trivially true hypotheses are omitted). This proof highlights that `FJf` types also include *future types*. In particular, when the returned type of a method is declared to be `C`, the corresponding invocations return *future values* of type `Fut(C)` because the context of the invocations cannot assume the presence of the returned value. The operation retrieving values, called `get`, takes an expression of type `Fut(C)` and returns `C`. As the reader may expect, the operation releasing the control, called `await`, takes an expression of type `Fut(C)` and returns `Fut(C)`.

Back to the judgment $\Gamma \vdash_a \texttt{e} : (\texttt{T}, \mathbbm{r}),\ \mathbbm{c}$, the future record $\mathbbm{r}$ stores the object names to access to future values. These values, being the results of method invocations, are available *provided* the control of the objects of the invoked methods has been acquired. For example, the above expression `new C(this.f)`, being its type the (standard) class `C`, has $b[\texttt{f} : Y]$ as future record. The expression `c!m()`, being its type `Fut(C)`, has $b \rightsquigarrow b'[\texttt{f} : Y]$ as future record. This means that, if the context needs the value of `c!m()` – a future record $b'[\texttt{f} : Y]$ –, as it is the case in the body of `n`, then *it is necessary to get the control of the object with name* b (otherwise `m` cannot be executed). By storing object names, future records $\mathbbm{r}$ play a critical role to enforce the aforementioned constraint.

The method `n` has contract $\texttt{D.m}\ b[\texttt{f} : Y](\,) \to b'[\texttt{f} : Y] \centerdot (a, b)^{\mathtt{a}}$, where a is the object name of the caller and b is the object name of the callee This contract shows up that `n` invokes `m` and waits for `m` termination by releasing the control on its object a – the pair $(a^{\mathtt{a}}, b^{\mathtt{a}})$. Said otherwise, the method `n` may complete provided the control on the object name b is released. It is worth to notice that the `get` operation does not add further commitments to the contract of `n`: a `get` after an `await` always succeeds therefore it is never displayed in contracts.

Contracts are inputs to our deadlock and livelock analysis technique developed in the second part of the paper. The technique returns finite state automata, called *lafsa*, where states are relations on object names. Figure 1(i) illustrates the (single state) *lafsa* of the contract $\texttt{D.m}\ b[\texttt{f} : Y](\,) \to b'[\texttt{f} : Y] \centerdot (a, b)^{\mathtt{a}}$ (the one of `D.n`). The state contains only the pair $(a, b)^{\mathtt{a}}$ because the invocation $\texttt{D.m}\ b[\texttt{f} : Y](\,)$ has contract `0` like the homonymous method in `C`. Figure 1(ii) illustrates the (single state) *lafsa* of the contract $\texttt{E.m}\ b[\texttt{f} : Y](\,) \to b'[\texttt{f} : Y] \centerdot (a, b)$ (the one of `E.n`). These two automata do not manifest any problematic dependency between object names as long as they are invoked with values of `this` and of the argument that are different ($a \neq b$). However,

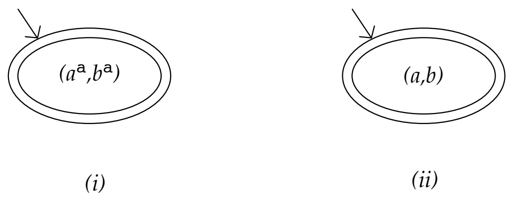

Fig. 1. Sample *lafsa* for methods of Table 1: (i) for the method `n` of `D`, (ii) for the method `n` of `E`

a critical pair appears if $a = b$ – an *object-circularity* –, as in the invocation `x!n(x)`, where `x` is an object of class `E`. In fact, in this case, the program deadlocks. On the contrary, if `x` is an object of class `D`, being Figure 1(i) the *lafsa* modelling the contract of `D.n`, the state becomes $\{(a^{\mathtt{a}}, a^{\mathtt{a}})\}$ which is not a critical pair in `FJf`. In fact, in this case, the program terminates.

3. The calculus `FJf`

The syntax and the semantics of `FJf` are illustrated in the following two subsections; the last subsection is devoted to the discussion of examples.

3.1. *Syntax*

The syntax of `FJf` uses four disjoint infinite sets of names: *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 *variables*, ranged over by `x`, `y`, $\cdots$. The special name `this` is assumed to belong to the set of variables. The notation $\bar{\mathtt{C}}$ is a shorthand for $\mathtt{C}_1;\cdots;\mathtt{C}_n$ and similarly for the other names. Sequences of pairs are abbreviated as $\mathtt{C}_1\ \mathtt{f}_1;\cdots;\mathtt{C}_n\ \mathtt{f}_n$ with $\bar{\mathtt{C}}\ \bar{\mathtt{f}}$. The concatenation of sequences is denoted by a semicolon; the empty sequence is written as $\bullet$ and is omitted when it is clear the presence of an empty sequence from the context.

The abstract syntax of *class declarations* `CL`, *method declarations* `M`, *expressions* `e`, and *types* `T` of `FJf` is the following

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

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

A program is a pair $(\textsc{ct}, \mathtt{e})$, where the *class table* $\textsc{ct}$ 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 $\textsc{ct}$.

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 and whose definition does not appear in the class table. In types, the terms `Fut(T)` are called *futures* of type `T`.

Let *fields*(`C`), *mtype*(`m`, `C`), and *mbody*(`m`, `C`) be the lookup functions that are reported in Table 2 (these are the same as in `FJ` (Igarashi et al., 2001)). We write $\mathtt{m} \in \mathtt{C}$ when *mtype*(`m`, `C`) is defined (`m` is a method of `C`). The class table satisfies the following well-formed conditions:

Field lookup:

$$\mathit{fields}(\texttt{Object}) = \bullet \qquad \frac{\textsc{ct}(\texttt{C}) = \texttt{class C extends D } \{\bar{\texttt{T}}\ \bar{\texttt{f}};\ \bar{\texttt{M}}\} \quad \mathit{fields}(\texttt{D}) = \bar{\texttt{T}}'\ \bar{\texttt{g}}}{\mathit{fields}(\texttt{C}) = \bar{\texttt{T}}\ \bar{\texttt{f}},\ \bar{\texttt{T}}'\ \bar{\texttt{g}}}$$

Method type lookup:

$$\frac{\begin{array}{c}\textsc{ct}(\texttt{C}) = \texttt{class C extends D } \{\bar{\texttt{T}}\ \bar{\texttt{f}};\ \bar{\texttt{M}}\} \\ \texttt{T}'\ \texttt{m}\ (\bar{\texttt{T}}'\ \bar{\texttt{x}})\{\texttt{return e};\} \in \bar{\texttt{M}}\end{array}}{\mathit{mtype}(\texttt{m}, \texttt{C}) = \bar{\texttt{T}}' \to \texttt{T}'} \qquad \frac{\begin{array}{c}\textsc{ct}(\texttt{C}) = \texttt{class C extends D } \{\bar{\texttt{T}}\ \bar{\texttt{f}};\ \bar{\texttt{M}}\} \\ \texttt{m} \notin \bar{\texttt{M}}\end{array}}{\mathit{mtype}(\texttt{m}, \texttt{C}) = \mathit{mtype}(\texttt{m}, \texttt{D})}$$

Method body lookup:

$$\frac{\begin{array}{c}\textsc{ct}(\texttt{C}) = \texttt{class C extends D } \{\bar{\texttt{T}}\ \bar{\texttt{f}};\ \bar{\texttt{M}}\} \\ \texttt{T}'\ \texttt{m}\ (\bar{\texttt{T}}'\ \bar{\texttt{x}})\{\texttt{return e};\} \in \bar{\texttt{M}}\end{array}}{\mathit{mbody}(\texttt{m}, \texttt{C}) = \bar{\texttt{x}}.\texttt{e}} \qquad \frac{\begin{array}{c}\textsc{ct}(\texttt{C}) = \texttt{class C extends D } \{\bar{\texttt{T}}\ \bar{\texttt{f}};\ \bar{\texttt{M}}\} \\ \texttt{m} \notin \bar{\texttt{M}}\end{array}}{\mathit{mbody}(\texttt{m}, \texttt{C}) = \mathit{mbody}(\texttt{m}, \texttt{D})}$$

Table 2. *Lookup auxiliary functions*

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

$$\frac{\texttt{T}_1 \texttt{ <: } \texttt{T}_2}{\texttt{Fut}(\texttt{T}_1) \texttt{ <: } \texttt{Fut}(\texttt{T}_2)} \qquad \frac{\textsc{ct}(\texttt{C}_1) = \texttt{class C}_1 \texttt{ extends C}_2\ \{\cdots\}}{\texttt{C}_1 \texttt{ <: } \texttt{C}_2}$$

is antisymmetric.

3.2. *Semantics*

The operational semantics of `FJf` uses two additional infinite sets of names: *object names*, ranged over by a, b, $\cdots$ and *task names*, ranged over by $\texttt{t}$, $\texttt{t}'$, $\cdots$. Object names are partitioned according to the class they belongs. We assume there are infinitely many object names per class and the function $\mathit{fresh}(\texttt{C})$ returns a new object name of class `C`. Given an object name a, the function $\mathit{class}(a)$ returns its class.

Values $\texttt{v}$, $\texttt{v}'$, $\cdots$, are terms defined by the following grammar:

$$\texttt{v} \ ::= \quad \texttt{t} \ \mid \ a[\bar{\texttt{f}} : \bar{\texttt{v}}]$$

For example $a[\,]$ is a value of a class without fields (like `Object`). Values as $a[\bar{\texttt{f}} : \bar{\texttt{v}}]$ are *named records*, where a is the name and $\bar{\texttt{v}}$ are the values stored in the fields $\bar{\texttt{f}}$. The operational semantics uses object names to implement mutual exclusion between tasks of the same object. (The analysis in Section 6 will use object names to catch circular dependencies between tasks.) In the following, with an abuse of notation, values, as well as expressions, will be ranged over by $\texttt{e}, \texttt{e}', \cdots$.

Let *states* S, S', $\cdots$, be sets of *tasks* $\texttt{t} :_a^\ell \texttt{e}$, where `t` is a task name, a is an object name, ℓ is either $\top$ (if the task owns the control of a) or $\bot$ (if not), and `e` is an expression. The operational semantics of `FJf` is the transition relation $\xrightarrow{a}$ between states defined in Table 3 where the following notations and shortenings are used:

(FIELD)
$$\frac{\mathtt{f} : \mathtt{v} \in \bar{\mathtt{f}} : \bar{\mathtt{v}}}{\mathtt{t} :_a^\top \mathsf{E}[b[\bar{\mathtt{f}} : \bar{\mathtt{v}}].\mathtt{f}] \xrightarrow{a} \mathtt{t} :_a^\top \mathsf{E}[\mathtt{v}]}$$

(INVK)
$$\frac{\mathit{mbody}(\mathtt{m}, \mathit{class}(b)) = \bar{\mathtt{x}}.\mathtt{e} \quad \mathtt{t}' = \mathit{freshtask}(\ )}{\mathtt{t} :_a^\top \mathsf{E}[b[\bar{\mathtt{f}} : \bar{\mathtt{v}}]!\mathtt{m}(\bar{\mathtt{v}}')] \xrightarrow{a} \mathtt{t} :_a^\top \mathsf{E}[\mathtt{t}'],\ \mathtt{t}' :_b^\bot \mathtt{e}[{}^{b[\bar{\mathtt{f}} : \bar{\mathtt{v}}]}/_{\mathtt{this}}][{}^{\bar{\mathtt{v}}'}/_{\bar{\mathtt{x}}}]}$$

(NEW)
$$\frac{\mathit{fields}(\mathtt{C}) = \bar{\mathtt{T}}\ \bar{\mathtt{f}} \quad b = \mathit{fresh}(\mathtt{C})}{\mathtt{t} :_a^\top \mathsf{E}[\mathtt{new}\ \mathtt{C}(\bar{\mathtt{v}})] \xrightarrow{a} \mathtt{t} :_a^\top \mathsf{E}[b[\bar{\mathtt{f}} : \bar{\mathtt{v}}]]}$$

(SEQ)
$$\mathtt{t} :_a^\top \mathtt{v}; \mathtt{e} \xrightarrow{a} \mathtt{t} :_a^\top \mathtt{e}$$

(GET)
$$\mathtt{t} :_a^\top \mathsf{E}[\mathtt{t}'.\mathtt{get}],\ \mathtt{t}' :_b \mathtt{v} \xrightarrow{a} \mathtt{t} :_a^\top \mathsf{E}[\mathtt{v}],\ \mathtt{t}' :_b \mathtt{v}$$

(AWAITT)
$$\mathtt{t} :_a^\top \mathsf{E}[\mathtt{t}'.\mathtt{await}],\ \mathtt{t}' :_b \mathtt{v} \xrightarrow{a} \mathtt{t} :_a^\top \mathsf{E}[\mathtt{t}'],\ \mathtt{t}' :_b \mathtt{v}$$

(AWAITF)
$$\frac{\mathtt{e} \neq \mathtt{v}}{\mathtt{t} :_a^\top \mathsf{E}[\mathtt{t}'.\mathtt{await}],\ \mathtt{t}' :_b \mathtt{e} \xrightarrow{a} \mathtt{t} :_a^\bot \mathsf{E}[\mathtt{t}'.\mathtt{await}],\ \mathtt{t}' :_b \mathtt{e}}$$

(RELEASE)
$$\mathtt{t} :_a^\top \mathtt{v} \xrightarrow{a} \mathtt{t} :_a^\bot \mathtt{v}$$

(LOCK)
$$\frac{\mathtt{e} \neq \mathtt{v}}{\mathtt{t} :_a^\bot \mathtt{e} \xrightarrow{a} \mathtt{t} :_a^\top \mathtt{e}}$$

(STATE)
$$\frac{\mathsf{S} \xrightarrow{a} \mathsf{S}' \quad \mathit{unlocked}(\mathsf{S}'', a)}{\mathsf{S}, \mathsf{S}'' \xrightarrow{a} \mathsf{S}', \mathsf{S}''}$$

Table 3. *The transition relation of* `FJf`*.*

– *evaluation contexts* E whose syntax is:

$$\mathsf{E} \ ::= \ [\ ] \mid \mathsf{E}!\mathtt{m}(\bar{\mathtt{e}}) \mid \mathsf{E}.\mathtt{f} \mid a[\bar{\mathtt{f}} : \bar{\mathtt{v}}]!\mathtt{m}(\bar{\mathtt{v}}, \mathsf{E}, \bar{\mathtt{e}}) \mid \mathtt{new}\ \mathtt{C}(\bar{\mathtt{v}}, \mathsf{E}, \bar{\mathtt{e}}) \mid \mathsf{E}.\mathtt{get} \mid \mathsf{E}.\mathtt{await} \mid \mathsf{E}; \mathtt{e}$$

– the predicate *unlocked*(S, a) that returns *true* if every $\mathtt{t} :_a^\ell \mathtt{e}$ in S is such that $\ell = \bot$;
– the function *freshtask*( ) always returns a new task name;
– in $\mathtt{t} :_a^\ell \mathtt{e}$, the superscript ℓ is omitted when it is not relevant.

The rules defining field selection, object creation, and sequence, namely (FIELD), (NEW), (SEQ), are standard; we therefore discuss the other ones. Rule (INVK) defines the method invocation. According to this rule, the evaluation of $b[\bar{\mathtt{f}} : \bar{\mathtt{v}}]!\mathtt{m}(\bar{\mathtt{v}}')$ produces a future reference $\mathtt{t}'$ to the value returned by `m`. The task evaluating the called method is created and the evaluation of the caller can continue – the invocation is asynchronous; however the evaluation of the called method `m` cannot begin until its value of ℓ becomes $\top$. Rule (GET) permits the retrieval of 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 – then `await` is unblocking; otherwise the control of the object is released by `t`. Rule (RELEASE) models task termination, which amounts to store the returned value in the state and releasing the control of the object. According to the transition relation, a task $\mathtt{t} :_a^\ell \mathtt{e}$ moves provided $\ell = \top$, except for rule (LOCK). This rule allows a task with a non-value expression to get the control. The rule must be read in conjunction with rule (STATE) that lifts transitions $\xrightarrow{a}$ to complex states and enforces the property that there is always at most one task per object owning the control. This means that (LOCK) cannot be used if the state has a task $\mathtt{t}' :_a^\top \mathtt{e}'$.

The following statement guarantees that the property "there is at most one task that has the control per object" is an invariance of the transition relation.

Proposition 3.1. Let S be *sound* if, for every a, there is at most one task $\mathtt{t} :_a^{\ell} \mathtt{e}$ with $\ell = \top$. If S is sound and $\mathsf{S} \xrightarrow{a} \mathsf{S}'$ then S' is sound as well.

The initial state of a program $(\textsc{ct}, \mathtt{e})$ is $\mathtt{t} :_a^{\top} \mathtt{e}[^{a[\,]}/_{\mathtt{this}}]$ where a is a name of class `Object`. We write $\mathsf{S} \longrightarrow^* \mathsf{S}'$ if there are $a_1, \cdots, a_n$ such that $\mathsf{S} \xrightarrow{a_1} \cdots \xrightarrow{a_n} \mathsf{S}'$.

3.3. *Examples*

As a first example, we detail the evaluation of the expression `(new D(this))!n(new D(this))`, where the class `D` is defined in Table 1 (`this` in the initial state is a value of class `Object`).

$$
\begin{array}{lll}
\mathtt{t} :_a^{\top} \mathtt{(new\ D(}a\mathtt{[\ ]))!n(new\ D(}a\mathtt{[\ ]))} & & \\
\quad \xrightarrow{a} \mathtt{t} :_a^{\top} b\mathtt{[f:}a\mathtt{[\ ]]!n(new\ D(}a\mathtt{[\ ]))} & (1) & (\textsc{New}) \\
\quad \xrightarrow{a} \mathtt{t} :_a^{\top} b\mathtt{[f:}a\mathtt{[\ ]]!n(}c\mathtt{[f:}a\mathtt{[\ ]])} & (2) & (\textsc{New}) \\
\quad \xrightarrow{a} \mathtt{t} :_a^{\top} \mathtt{t1},\ \mathtt{t1} :_b^{\bot} c\mathtt{[f:}a\mathtt{[\ ]]!m().await.get} & (3) & (\textsc{Invk}) \\
\quad \xrightarrow{b} \mathtt{t} :_a^{\top} \mathtt{t1},\ \mathtt{t1} :_b^{\top} \mathtt{t2.await.get},\ \mathtt{t2} :_c^{\bot} \mathtt{new\ C(}c\mathtt{[f:}a\mathtt{[\ ]].f)} & (4) & (\textsc{Invk}) \\
\quad \xrightarrow{c} \mathtt{t} :_a^{\top} \mathtt{t1},\ \mathtt{t1} :_b^{\top} \mathtt{t2.await.get},\ \mathtt{t2} :_c^{\top} \mathtt{new\ C(}c\mathtt{[f:}a\mathtt{[\ ]].f)} & (5) & (\textsc{Lock}) \\
\quad \xrightarrow{c} \mathtt{t} :_a^{\top} \mathtt{t1},\ \mathtt{t1} :_b^{\top} \mathtt{t2.await.get},\ \mathtt{t2} :_c^{\top} \mathtt{new\ C(}a\mathtt{[\ ])} & (6) & (\textsc{Field}) \\
\quad \xrightarrow{c} \mathtt{t} :_a^{\top} \mathtt{t1},\ \mathtt{t1} :_b^{\top} \mathtt{t2.await.get},\ \mathtt{t2} :_c^{\top} d\mathtt{[f:}a\mathtt{[\ ]]} & (7) & (\textsc{New}) \\
\quad \xrightarrow{b} \mathtt{t} :_a^{\top} \mathtt{t1},\ \mathtt{t1} :_b^{\top} \mathtt{t2.get},\ \mathtt{t2} :_c^{\top} d\mathtt{[f:}a\mathtt{[\ ]]} & (8) & (\textsc{AwaitT}) \\
\quad \xrightarrow{b} \mathtt{t} :_a^{\top} \mathtt{t1},\ \mathtt{t1} :_b^{\top} d\mathtt{[f:}a\mathtt{[\ ]]},\ \mathtt{t2} :_c^{\top} d\mathtt{[f:}a\mathtt{[\ ]]} & (9) & (\textsc{Get})
\end{array}
$$

The reader may notice that, in the final state, the tasks `t`, `t1` and `t2` will terminate one after the other by releasing the controls of the corresponding objects.

Consider the code of `n` in class `E` of Table 1 and let b be an object name of class `E`. Let us evaluate the state $\mathtt{t} :_a^{\top} b[\mathtt{f} : a[\,]]!\mathtt{n}(b[\mathtt{f} : a[\,]])$ (corresponding to the expression `x!n(x)` that has been already discussed in Section 2, here we are detailing its semantics):

$$
\begin{array}{ll}
\mathtt{t} :_a^{\top} b[\mathtt{f} : a[\,]]!\mathtt{n}(b[\mathtt{f} : a[\,]]) & \\
\quad \xrightarrow{a} \mathtt{t} :_a^{\top} \mathtt{t1},\ \mathtt{t1} :_b^{\bot} b\mathtt{[f:}a\mathtt{[\ ]]!m().get} & (\textsc{Invk}) \\
\quad \xrightarrow{b} \mathtt{t} :_a^{\top} \mathtt{t1},\ \mathtt{t1} :_b^{\top} b\mathtt{[f:}a\mathtt{[\ ]]!m().get} & (\textsc{Lock}) \\
\quad \xrightarrow{b} \mathtt{t} :_a^{\top} \mathtt{t1},\ \mathtt{t1} :_b^{\top} \mathtt{t2.get}\ ,\ \mathtt{t2} :_b^{\bot} \mathtt{new\ C(}b\mathtt{[f:}a\mathtt{[\ ]].f)} & (\textsc{Invk})
\end{array}
$$

The last state is a deadlock because `t2` will never get the control on the object b, which is owned by `t1`.

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

```
class F extends E {
    Fut(C) p(E b, E c){ return b!n(c);c!n(b) ;}
```

and consider the state $\mathtt{t} :_a^{\top}$ `(new F(new Object))!p(new F(new Object),` a`[f:` b`[ ]])`, where

a is an object of class `F`. Its evaluation is as follows ($\xrightarrow{a}^{k}$ means $\underbrace{\xrightarrow{a} \cdots \xrightarrow{a}}_{k \text{ times}}$):

$$\mathtt{t} :_{a}^{\top} \mathtt{(new\ F(new\ Object))!p(new\ F(new\ Object),}\ a\mathtt{[f:}\ b\mathtt{[\ ]])}$$
$$\xrightarrow{a}^{4}\ \mathtt{t} :_{a}^{\top} (a'[\mathtt{f} : b'])\mathtt{!p}(a''[\mathtt{f} : b''], a[\mathtt{f} : b[\ ]]) \qquad (\textsc{New})$$
$$\xrightarrow{a} \xrightarrow{a'}\ \mathtt{t} :_{a}^{\top} \mathtt{t1},\ \mathtt{t1} :_{a'}^{\top} (a''[\mathtt{f} : b''])\mathtt{!n}(a[\mathtt{f} : b[\ ]]); (a[\mathtt{f} : b[\ ]])\mathtt{!n}(a''[\mathtt{f} : b'']) \qquad (\textsc{Invk})+(\textsc{Lock})$$
$$\xrightarrow{a'}\ \mathtt{t} :_{a}^{\top} \mathtt{t1},\ \mathtt{t1} :_{a'}^{\top} \mathtt{t2}; (a[\mathtt{f} : b[\ ]])\mathtt{!n}(a''[\mathtt{f} : b'']),\ \mathtt{t2} :_{a''}^{\perp} (a[\mathtt{f} : b[\ ]])\mathtt{!m().get} \qquad (\textsc{Invk})$$
$$\xrightarrow{a'}^{2}\ \mathtt{t} :_{a}^{\top} \mathtt{t1},\ \mathtt{t1} :_{a'}^{\top} \mathtt{t3},\ \mathtt{t2} :_{a''}^{\perp} (a[\mathtt{f} : b[\ ]])\mathtt{!m().get},\ \mathtt{t3} :_{a}^{\perp} (a''[\mathtt{f} : b''])\mathtt{!m().get} (\textsc{Seq})+(\textsc{Invk})$$
$$\xrightarrow{a}\ \mathtt{t} :_{a}^{\perp} \mathtt{t1},\ \mathtt{t1} :_{a'}^{\top} \mathtt{t3},\ \mathtt{t2} :_{a''}^{\perp} (a[\mathtt{f} : b[\ ]])\mathtt{!m().get},\ \mathtt{t3} :_{a}^{\perp} (a''[\mathtt{f} : b''])\mathtt{!m().get} \qquad (\textsc{Release})$$
$$\xrightarrow{a''}^{2}\ \mathtt{t} :_{a}^{\perp} \mathtt{t1},\ \mathtt{t1} :_{a'}^{\top} \mathtt{t3},\ \mathtt{t2} :_{a''}^{\top} \mathtt{t4.get},\ \mathtt{t3} :_{a}^{\perp} (a''[\mathtt{f} : b''])\mathtt{!m().get},\ \mathtt{t4} :_{a}^{\perp} \mathtt{new\ C}(a[\mathtt{f} : b[\ ]].\mathtt{f})$$
$$(\textsc{Lock})+(\textsc{Invk})$$

The last state is the critical one: there are two tasks `t3` and `t4` that are waiting to get the control on the object a. According to scheduler's choice leans towards `t3` or `t4` one gets a deadlocked state or not, respectively.

A last example discusses an expression that yields a livelock state. Let `G` be the following extension of the class `D` in Table 1:

```
class G extends D {
    C p(D b) { return b!n(this).get ;}
}
```

and let us consider the evaluation:

$$\mathtt{t} :_{a}^{\top} \mathtt{(new\ G(new\ Object))!p(new\ D(new\ Object))}$$
$$\xrightarrow{a}^{2} \xrightarrow{a}^{2}\ \mathtt{t} :_{a}^{\top} a'[\mathtt{f} : b'[\ ]]\mathtt{!p}(a''[\mathtt{f} : b''[\ ]]) \qquad (1) \qquad (\textsc{New})$$
$$\xrightarrow{a}\ \mathtt{t} :_{a}^{\top} \mathtt{t1},\ \mathtt{t1} :_{a'}^{\perp} a''[\mathtt{f} : b''[\ ]]\mathtt{!n}(a'[\mathtt{f} : b'[\ ]])\mathtt{.get} \qquad (2) \qquad (\textsc{Invk})$$
$$\xrightarrow{a'}^{2}\ \mathtt{t} :_{a}^{\top} \mathtt{t1},\ \mathtt{t1} :_{a'}^{\top} \mathtt{t2.get},\ \mathtt{t2} :_{a''}^{\perp} a'[\mathtt{f} : b'[\ ]]\mathtt{!m().await.get} \qquad (3) \qquad (\textsc{Lock})+(\textsc{Invk})$$
$$\xrightarrow{a''}^{2}\ \mathtt{t} :_{a}^{\top} \mathtt{t1},\ \mathtt{t1} :_{a'}^{\top} \mathtt{t2.get},\ \mathtt{t2} :_{a''}^{\top} \mathtt{t3.await.get},\ \mathtt{t3} :_{a'}^{\perp} \mathtt{new\ C}(b'[\ ]) \qquad (4) \qquad (\textsc{Lock})$$
$$\xrightarrow{a''}\ \mathtt{t} :_{a}^{\top} \mathtt{t1},\ \mathtt{t1} :_{a'}^{\top} \mathtt{t2.get},\ \mathtt{t2} :_{a''}^{\perp} \mathtt{t3.await.get},\ \mathtt{t3} :_{a'}^{\perp} \mathtt{new\ C}(b'[\ ]) \qquad (5) \qquad (\textsc{Await})$$
$$\vdots$$

From state (4) onwards, `t1` is blocked while `t2` continuously gets an releases the lock on a'' waiting for the termination of `t3`. In turn `t3` will never get the control of the object a' (that is got by `t1`), therefore it will not terminate.

4. Inference of contracts in `FJf`

4.1. *Preliminaries: contracts and substitutions*

The analysis technique we develop in the rest of the paper uses abstract descriptions of methods and expression behaviours, called *contract methods* and *contracts*, respectively. The syntax of these descriptions uses an infinite set of *record names*, ranged over by $X, Y, Z, \cdots$. *Future records* $\mathbb{r}$, $\mathbb{s}$, $\cdots$, and *contracts* $\mathbb{c}$, $\mathbb{c}'$, $\cdots$ are defined by the following grammar:

$$\begin{array}{lcl}
\mathbb{r} & ::= & X \mid a[\bar{\mathtt{f}} : \bar{\mathbb{r}}] \mid a \rightsquigarrow \mathbb{r} \\
\mathbb{c} & ::= & \mathtt{0} \mid \mathtt{C.m}\ \mathbb{r}(\bar{\mathbb{r}}) \to \mathbb{r}' \mid \mathtt{C.m}\ \mathbb{r}(\bar{\mathbb{r}}) \to \mathbb{r}'.(a, a') \mid \mathtt{C.m}\ \mathbb{r}(\bar{\mathbb{r}}) \to \mathbb{r}'.(a, a')^{\mathtt{a}} \mid (a, a') \mid (a, a')^{\mathtt{a}} \mid \mathbb{c} \fatsemi \mathbb{c}
\end{array}$$

A record name X represents a variable that may be possibly instantiated by substitutions. The future record $a[\bar{\mathtt{f}} : \bar{\mathbb{r}}]$ defines the object name and the future records of values stored in its fields. The future record $a \rightsquigarrow \mathbb{r}$ specifies that, in order to access to $\mathbb{r}$ one has to acquire the control of the object with name a (and to release this control once the method has been evaluated). Future records as $a \rightsquigarrow \mathbb{r}$ are associated to method invocations: the object name a represents the object of the invoked method. The name a in $a[\bar{\mathtt{f}} : \bar{\mathbb{r}}]$ and $a \rightsquigarrow \mathbb{r}$ will be called the *root of the future record* and is returned by the (partial) function *root*$(\cdot)$.

The contract $\mathbb{c}$ collects the method invocations inside expressions and the object name dependencies. A contract may be empty, noted `0`, specifying that the method behaviour is irrelevant for our analysis; or $\mathtt{C.m}\,\mathbb{r}(\bar{\mathbb{r}}) \to \mathbb{r}'$, specifying that the method `m` of class `C` is going to be invoked on an object $\mathbb{r}$, with arguments $\bar{\mathbb{r}}$, and an object $\mathbb{r}'$ will be returned; or $\mathtt{C.m}\,\mathbb{r}(\bar{\mathbb{r}}) \to \mathbb{r}'\,\mathtt{.}(a, a')$, indicating that the current method execution requires the termination of method `C.m` running on a' to release the object with name a; or $\mathtt{C.m}\,\mathbb{r}(\bar{\mathbb{r}}) \to \mathbb{r}'\,\mathtt{.}(a, a')^{\mathsf{a}}$, indicating that the current method execution requires the termination of method `C.m` running on a' to continue (the object with name a may be released meanwhile); or just (a, a') (resp. $(a, a')^{\mathsf{a}}$) when the dependency is due to a `get` (resp. an `await`) operation on a field or on a parameter, which have contract `0`, instead of being directly on a method invocation. Pairs (a, a') and $(a, a')^{\mathsf{a}}$ are called *object name dependencies.* The contract $\mathbb{c} \fatsemi \mathbb{c}'$ defines the abstract behaviour of sequential composition of expressions.

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

(a) $\mathtt{C.m}\ a\mathtt{[f:}b\mathtt{[]]()}{\to}a''\mathtt{[f:}b\mathtt{[]]} \fatsemi \mathtt{C.m}\ a'\mathtt{[f:}b'\mathtt{[]]()}{\to}\ b''\mathtt{[f:}b'\mathtt{[]]}$

(b) $\mathtt{C.m}\ a\mathtt{[f:}b\mathtt{[]]()}{\to}\ a''\mathtt{[f:}b\mathtt{[]]}\mathtt{.}(a''', a) \fatsemi \mathtt{C.m}\ a'\mathtt{[f:}b'\mathtt{[]]()}{\to}\ b''\mathtt{[f:}b'\mathtt{[]]}\mathtt{.}(a''', a')^{\mathsf{a}}$

The contract (a) defines a sequence of two invocations of method `m` in Table 1; the future record of the first one is $a[\mathtt{f} : b[\,]]$, the future record of the second one is $a'[\mathtt{f} : b'[\,]]$. This contract is not enforcing any constraint on object names because the values of invocations are not needed in the context. As we will see below, an `FJf` expression retaining this contract is `x!m() ; y!m()`, with `x` and `y` variables of class `C`. The contract (b) defines two invocations of method `m` as (a) and, additionally, expresses that the value of the first invocation is required as well as the termination of the second invocation. An `FJf` expression retaining this contract is `x!m().get ; y!m().await`, with `x` and `y` variables of class `C`.

A future record is *linear* if the object names and the record names occur linearly. The function *names*$(\cdot)$ returns the object and record names. *Method contracts*, ranged over by $\mathbb{C}$, $\mathbb{C}'$, $\cdots$, are terms of the form

$$\mathbb{r}(\bar{\mathbb{s}})\ \{\mathbb{c}\}\ \mathbb{r}'$$

where

1. future records $\mathbb{r}$ and in $\bar{\mathbb{s}}$ are linear and
2. object and record names occurring in $\mathbb{r}$ and in $\bar{\mathbb{s}}$ are pairwise different (for every $\mathbb{s} \in \bar{\mathbb{s}}$, $\mathit{names}(\mathbb{r}) \cap \mathit{names}(\mathbb{s}) = \varnothing$ and for different arguments $\mathbb{s}, \mathbb{s}' \in \bar{\mathbb{s}}$, $\mathit{names}(\mathbb{s}) \cap \mathit{names}(\mathbb{s}') = \varnothing$) and
3. record names occurring in $\mathbb{c}$ or in $\mathbb{r}'$ are a subset of those in $\mathit{names}(\mathbb{r}) \cup \mathit{names}(\bar{\mathbb{s}})$.

It is worth to remark that 3. does not apply to object names occurring in $\mathbb{c}$ or in $\mathbb{r}'$, which may be not occurring in $\mathit{names}(\mathbb{r}) \cup \mathit{names}(\bar{\mathbb{s}})$.

The subterm $\mathbb{r}(\bar{\mathbb{s}})$ of a method contract $\mathbb{r}(\bar{\mathbb{s}})\ \{\mathbb{c}\}\ \mathbb{r}'$ is called *header*; $\mathbb{r}'$ is called *returned future record.* The header and the returned future record, written $\mathbb{r}(\bar{\mathbb{s}}) \to \mathbb{r}'$, are called *interface.* We observe that, in an interface $\mathbb{r}(\bar{\mathbb{s}}) \to \mathbb{r}'$, $\mathbb{r}$ and $\bar{\mathbb{s}}$ and $\mathbb{r}'$ are subjected to the constraints 1., 2. and 3. above.

In $\mathbb{r}(\bar{\mathbb{s}})\ \{\mathbb{c}\}\ \mathbb{r}'$, (object and record) names in the header *bind* the (object and record) names occurring in $\mathbb{c}$ and in $\mathbb{r}'$. For example

— the term $a[\mathtt{f} : b[\ ]]()\ \{\mathtt{0}\}\ a'[\mathtt{f} : b[\ ]]$ is the method contract of $\mathtt{C.m}$ in Table 1. The name b in the header binds the occurrence of b in the returned future record. The name a' in the returned future record is *fresh*, namely it is unbound by the header. This means that $\mathtt{m}$ returns an object that has been created during its evaluation.

— the term $a[\mathtt{f} : X](a'[\mathtt{f} : b[\ ]])\ \{\mathtt{E.m}\ a'[\mathtt{f} : b[\ ]]() \to a''[\mathtt{f} : b[\ ]]\,{\bullet}\,(a, a')\}\ a''[\mathtt{f} : b[\ ]]$ is the method contract of method $\mathtt{E.n}$ in Table 1. Few remarks are in order: (i) the field $\mathtt{f}$ of the object of $\mathtt{n}$ is never accessed in its body; for this reason we have a place-holder record name X instead of a future record; (ii) the names a, a' and b in the header of the method contract bind the occurrences of object names in the body and of the name b in the returned future record; (iii) the name a'' in the returned future record is *fresh.* The returned future record of $\mathtt{n}$ is an heredity of the method contract of $\mathtt{m}$.

A *future record substitution* σ is a finite mapping from object names to object names and from record names to future records. For example, $[{}^{a'}/_{a}][{}^{b[\bar{\mathtt{f}}\,:\,\bar{\mathbb{r}}]}/_{X}]$ indicates a substitution mapping a to a', and X to $b[\bar{\mathtt{f}} : \bar{\mathbb{r}}]$. The application of a substitution to an object type is defined in the standard way as follows:

$$\sigma(X) \;=\; \begin{cases} \mathbb{r} & \text{if } \sigma(X) = \mathbb{r} \\ X & \text{if } X \notin \mathit{dom}(\sigma) \end{cases}$$

$$\sigma(a) \;=\; \begin{cases} b & \text{if } \sigma(a) = b \\ a & \text{if } a \notin \mathit{dom}(\sigma) \end{cases}$$

$$\sigma(a[\bar{\mathtt{f}} : \bar{\mathbb{r}}]) \;=\; \sigma(a)[\bar{\mathtt{f}} : \sigma(\bar{\mathbb{r}})]$$

$$\sigma(a \rightsquigarrow \mathbb{r}) \;=\; \sigma(a) \rightsquigarrow \sigma(\mathbb{r})$$

Record substitutions are extended homomorphically to sequences of object types ($\sigma(\mathbb{r}_1, \ldots, \mathbb{r}_n) = \sigma(\mathbb{r}_1), \ldots, \sigma(\mathbb{r}_n)$) and to contracts. Additionally, when $\mathbb{r}$ is linear, we also define $\mathbb{s}[{}^{\mathbb{r}'}/_{\mathbb{r}}]$ by induction on the structure of $\mathbb{r}$:

$$\mathbb{s}[{}^{\mathbb{r}'}/_{\mathbb{r}}] \;=\; \begin{cases} \mathbb{s}[{}^{\mathbb{r}'}/_{X}] & \mathbb{r} = X \\ \mathbb{s}[{}^{b}/_{a}][{}^{\bar{\mathbb{r}}'}/_{\bar{\mathbb{r}}}] & \mathbb{r} = a[\bar{\mathtt{f}} : \bar{\mathbb{r}}] \text{ and } \mathbb{r}' = b[\bar{\mathtt{f}} : \bar{\mathbb{r}}'] \\ \mathbb{s}[{}^{b}/_{a}][{}^{\mathbb{r}'''}/_{\mathbb{r}''}] & \mathbb{r} = a \rightsquigarrow \mathbb{r}'' \text{ and } \mathbb{r}' = b \rightsquigarrow \mathbb{r}''' \end{cases}$$

It is worth to notice that $\mathbb{s}[{}^{\mathbb{r}'}/_{\mathbb{r}}]$ is partial because it requires the match between the patterns of $\mathbb{r}$ and $\mathbb{r}'$; for instance $\mathbb{s}[{}^{b[\ ]}/_{a \rightsquigarrow X}]$ and $\mathbb{s}[{}^{a \rightsquigarrow X}/_{b[\ ]}]$ are not defined.

The composition of substitutions σ and σ', written $\sigma \circ \sigma'$ is defined in the standard way as $(\sigma \circ \sigma')(\mathbb{r}) = \sigma'(\sigma(\mathbb{r}))$.

In the following type system, we use the operation $\between$ defined below.

Definition 4.1. Let $(a, a')^{[\mathtt{a}]}$ range over pairs (a, a') and $(a, a')^{\mathtt{a}}$. Then:

$$\begin{array}{rcll}
\mathtt{0} \between (a, a')^{[\mathtt{a}]} & = & (a, a')^{[\mathtt{a}]} & (1) \\
\mathbb{c} \fatsemi \mathtt{C.m}\ \mathbb{r}(\bar{\mathbb{s}}) \to \mathbb{r}' \between (a, a')^{[\mathtt{a}]} & = & \mathbb{c} \fatsemi (\mathtt{C.m}\ \mathbb{r}(\bar{\mathbb{s}}) \to \mathbb{r}'\,{\bullet}\,(a, a')^{[\mathtt{a}]}) & (2) \\
\mathbb{c} \fatsemi (a, a') \between (a, b)^{[\mathtt{a}]} & = & \mathbb{c} \fatsemi (a, a') \fatsemi (a, b)^{[\mathtt{a}]} & (3) \\
\mathbb{c} \fatsemi (\mathtt{C.m}\ \mathbb{r}(\bar{\mathbb{s}}) \to \mathbb{r}'\,{\bullet}\,(a, a')) \between (a, b)^{[\mathtt{a}]} & = & \mathbb{c} \fatsemi (\mathtt{C.m}\ \mathbb{r}(\bar{\mathbb{s}}) \to \mathbb{r}'\,{\bullet}\,(a, a')) \fatsemi (a, b)^{[\mathtt{a}]} & (4) \\
\mathbb{c} \fatsemi (a, a')^{\mathtt{a}} \between (a, a')^{[\mathtt{a}]} & = & \mathbb{c} \fatsemi (a, a')^{\mathtt{a}} & (5) \\
\mathbb{c} \fatsemi (\mathtt{C.m}\ \mathbb{r}(\bar{\mathbb{s}}) \to \mathbb{r}'\,{\bullet}\,(a, a')^{\mathtt{a}}) \between (a, a')^{[\mathtt{a}]} & = & \mathbb{c} \fatsemi (\mathtt{C.m}\ \mathbb{r}(\bar{\mathbb{s}}) \to \mathbb{r}'\,{\bullet}\,(a, a')^{\mathtt{a}}) & (6)
\end{array}$$

The purpose of $\between$ is to manage accumulations of pairs (a, b)[a]. in particular, rule (1) applies when a `get` or an `await` operation is performed on a variable or a field, which have contract `0`. Rule (2) applies when a `get` or an `await` operation is perfomed on a method invocation. Rules (3) and (4) apply when the expression to be typed is of the form `e.get.get` or `e.get.await`. In these cases the first `get` retrieves a value, that must be a future reference for the subsequent `get` to evaluate (similarly for `await`). Thus the new object name dependency (a, b) is separated from the previous one (a, a') because it does not contribute to locks resulting from the first `get` operation. The dependency (a, b) contributes to a new chain of dependencies. We remark that the first component of the two pairs is the same because it refers to the current object. Rules (5) and rule (6) apply when the typing expressions are of the form `e.await.await` and `e.await.get`; in these cases the new object name dependency is ignored, since if the first `await` has succeeded, then no lock would be arisen because of the second operation. Additionally, if the first `await` results in a lock, then the second operation will be never performed.

Since method contracts are binders for object and record names, we identify those terms that are equated by injective renaming of (free and bound) names and let $=^\alpha$ be the least equivalence. For example

$$a[\texttt{f} : b[\,]]()\ \{\texttt{0}\}\ a'[\texttt{f} : b[\,]] =^\alpha b[\texttt{f} : b[\,]]()\ \{\texttt{0}\}\ a[\texttt{f} : b'[\,]]$$

and

$$\begin{array}{ll} & a[\texttt{f} : X](a'[\texttt{f} : b[\,]])\ \{\texttt{E.m}\ a'[\texttt{f} : b[\,]]() \to a''[\texttt{f} : b[\,]] \bullet (a, a')\}\ a''[\texttt{f} : b[\,]] \\ =^\alpha & b[\texttt{f} : Z](c[\texttt{f} : a[\,]])\ \{\texttt{E.m}\ c[\texttt{f} : a[\,]]() \to d[\texttt{f} : a[\,]] \bullet (b, c)\}\ d[\texttt{f} : a[\,]] \end{array}$$

4.2. *The type system for contracts*

The typing judgments rely on *environments* Γ that bind variables to pairs $(\texttt{T}, \mathbb{r})$ and methods `C.m` to interfaces $\mathbb{r}(\bar{\mathbb{s}}) \to \mathbb{r}'$. The judgments have the form

$$\Gamma \vdash_a \texttt{e} : (\texttt{T}, \mathbb{r}),\ \mathbb{c}$$

and must be read: the expression `e` that occurs in a method of an object with root a has type `T`, future record $\mathbb{r}$, and contract $\mathbb{c}$ in the environment Γ. The judgements for expressions are defined in Table 4. In this table we use the same notation $\bar{\mathbb{c}}$ to indicate a tuple of contracts $(\mathbb{c}_1, \cdots, \mathbb{c}_n)$, when it occurs in the judgement of a tuple of expressions $\bar{\texttt{e}}$, and to indicate a sequential composition of contracts $\mathbb{c}_1 \fatsemi \cdots \fatsemi \mathbb{c}_n$, when it appears in the judgement of a single expression.

Rule (T-Field) defines the judgment for accessing to fields of an object produced by `e`. The rule constraints `e` to have a class type (not a future) and to have a future record as $a'[\bar{\texttt{f}} : \bar{\mathbb{r}}]$.

Rule (T-Invk) defines the judgments of method invocations $\texttt{e!m}(\bar{\texttt{e}})$. Let $a'[\bar{\texttt{f}} : \bar{\mathbb{r}}](\bar{\mathbb{s}}) \to \mathbb{r}$ be the interface of `C.m` stored in Γ. Object names and record names in this interface are actually place-holders for actual values. Therefore, in order to type $\texttt{e!m}(\bar{\texttt{e}})$, there must exist a substitution σ such that $\Gamma \vdash_a \texttt{e} : (\texttt{C}, \sigma(a'[\bar{\texttt{f}} : \bar{\mathbb{r}}])),\ \mathbb{c}$ and $\Gamma \vdash_a \bar{\texttt{e}} : (\bar{\texttt{T}}, \sigma(\bar{\mathbb{s}})),\ \bar{\mathbb{c}}$. (It is possible to use a unique σ since names in $a'[\bar{\texttt{f}} : \bar{\mathbb{r}}]$ and $\bar{\mathbb{s}}$ are disjoint.) The (standard) type of $\texttt{e!m}(\bar{\texttt{e}})$ is a future type $\texttt{Fut}(\texttt{T}')$, where $\texttt{T}'$ is determined with standard arguments for object-oriented languages. The future record of $\texttt{e!m}(\bar{\texttt{e}})$ is $\sigma(a') \rightsquigarrow \sigma(\mathbb{r})$ indicating that the value may be returned as soon as the control of $\sigma(a')$ is acquired. The contractual issue of $\texttt{e!m}(\bar{\texttt{e}})$ is almost obvious: it composes in sequence the contracts of `e`, $\bar{\texttt{e}}$ and the method invocation.

Rule (T-New) types object creations that, in the type system, amounts to using a fresh object name – called a' in the rule – for the root of its future record. The remaining part of the judgment is almost standard.

(T-Var)
$$\Gamma \vdash_a \mathtt{x} \,:\, \Gamma(\mathtt{x})\,,\ \mathtt{0}$$

(T-Field)
$$\frac{\Gamma \vdash_a \mathtt{e} \,:\, (\mathtt{C}, a'[\bar{\mathtt{f}} : \bar{\mathbb{r}}])\,,\ \mathbb{c} \qquad \mathit{fields}(\mathtt{C}) = \bar{\mathtt{T}}\,\bar{\mathtt{f}} \quad \mathtt{T}\,\mathtt{f} \in \bar{\mathtt{T}}\,\bar{\mathtt{f}} \quad \mathtt{f} : \mathbb{r} \in \bar{\mathtt{f}} : \bar{\mathbb{r}}}{\Gamma \vdash_a \mathtt{e.f} \,:\, (\mathtt{T}, \mathbb{r})\,,\ \mathbb{c}}$$

(T-Invk)
$$\frac{\begin{array}{c}\Gamma(\mathtt{C.m}) = a'[\bar{\mathtt{f}} : \bar{\mathbb{r}}](\bar{\mathbb{s}}) \to \mathbb{r} \\ \Gamma \vdash_a \mathtt{e} \,:\, (\mathtt{C}, \sigma(a'[\bar{\mathtt{f}} : \bar{\mathbb{r}}]))\,,\ \mathbb{c} \qquad \Gamma \vdash_a \bar{\mathtt{e}} \,:\, (\bar{\mathtt{T}}, \sigma(\bar{\mathbb{s}}))\,,\ \bar{\mathbb{c}} \\ \mathit{mtype}(\mathtt{m},\mathtt{C}) = \bar{\mathtt{T}}' \to \mathtt{T}' \quad \bar{\mathtt{T}} \mathrel{\mathtt{<:}} \bar{\mathtt{T}}'\end{array}}{\Gamma \vdash_a \mathtt{e!m}(\bar{\mathtt{e}}) \,:\, (\mathtt{Fut}(\mathtt{T}'), \sigma(a') \rightsquigarrow \sigma(\mathbb{r}))\,,\ \mathbb{c} \fatsemi \bar{\mathbb{c}} \fatsemi \mathtt{C.m}\ \sigma(a'[\bar{\mathtt{f}} : \bar{\mathbb{r}}])(\sigma(\bar{\mathbb{s}})) \to \sigma(\mathbb{r})}$$

(T-New)
$$\frac{\Gamma \vdash_a \bar{\mathtt{e}} \,:\, (\bar{\mathtt{T}}, \bar{\mathbb{r}})\,,\ \bar{\mathbb{c}} \quad \mathit{fields}(\mathtt{C}) = \bar{\mathtt{T}}'\,\bar{\mathtt{f}} \quad \bar{\mathtt{T}} \mathrel{\mathtt{<:}} \bar{\mathtt{T}}' \quad a' \ \mathit{fresh}}{\Gamma \vdash_a \mathtt{new}\ \mathtt{C}(\bar{\mathtt{e}}) \,:\, (\mathtt{C}, a'[\bar{\mathtt{f}} : \bar{\mathbb{r}}])\,,\ \bar{\mathbb{c}}}$$

(T-Get)
$$\frac{\Gamma \vdash_a \mathtt{e} \,:\, (\mathtt{Fut}(\mathtt{T}), a' \rightsquigarrow \mathbb{s})\,,\ \mathbb{c}}{\Gamma \vdash_a \mathtt{e.get} \,:\, (\mathtt{T}, \mathbb{s})\,,\ \mathbb{c} \between (a, a')}$$

(T-Await)
$$\frac{\Gamma \vdash_a \mathtt{e} \,:\, (\mathtt{Fut}(\mathtt{T}), a' \rightsquigarrow \mathbb{s})\,,\ \mathbb{c}}{\Gamma \vdash_a \mathtt{e.await} \,:\, (\mathtt{Fut}(\mathtt{T}), a' \rightsquigarrow \mathbb{s})\,,\ \mathbb{c} \between (a, a')^{\mathsf{a}}}$$

(T-Seq)
$$\frac{\Gamma \vdash_a \mathtt{e} \,:\, (\mathtt{T}, \mathbb{r})\,,\ \mathbb{c} \qquad \Gamma \vdash_a \mathtt{e}' \,:\, (\mathtt{T}', \mathbb{r}')\,,\ \mathbb{c}'}{\Gamma \vdash_a \mathtt{e}\,;\,\mathtt{e}' \,:\, (\mathtt{T}', \mathbb{r}')\,,\ \mathbb{c} \fatsemi \mathbb{c}'}$$

Table 4. *Typing rules of* `FJf` *expressions*

(T-Method)
$$\frac{\begin{array}{c}\Gamma(\mathtt{C.m}) = a[\bar{\mathtt{f}} : \bar{\mathbb{r}}](\bar{\mathbb{s}}) \to \mathbb{r}' \qquad \Gamma + \bar{\mathtt{x}} : (\bar{\mathtt{T}}, \bar{\mathbb{s}}) + \mathtt{this} : (\mathtt{C}, a[\bar{\mathtt{f}} : \bar{\mathbb{r}}]) \vdash_a \mathtt{e} \,:\, (\mathtt{T}', \mathbb{r}')\,,\ \mathbb{c} \qquad \mathtt{T}' \mathrel{\mathtt{<:}} \mathtt{T} \\ \mathtt{C} \mathrel{\mathtt{<:}} \mathtt{D} \text{ and } \mathtt{m} \in \mathtt{D} \quad \text{imply} \quad \mathit{mtype}(\mathtt{m},\mathtt{D}) = \bar{\mathtt{T}} \to \mathtt{T}'\end{array}}{\Gamma \vdash\ \mathtt{T}\ \mathtt{m}\ (\bar{\mathtt{T}}\ \bar{\mathtt{x}})\{\mathtt{return}\ \mathtt{e}\,;\} : a[\bar{\mathtt{f}} : \bar{\mathbb{r}}](\bar{\mathbb{s}})\{\mathbb{c}\}\ \mathbb{r}' \ \textsc{in}\ \mathtt{C}}$$

(T-Class)
$$\frac{\Gamma \vdash \bar{\mathtt{M}} : \bar{\mathbb{C}} \ \textsc{in}\ \mathtt{C}}{\Gamma \vdash \mathtt{class}\ \mathtt{C}\ \mathtt{extends}\ \mathtt{D}\ \{\bar{\mathtt{C}}\ \bar{\mathtt{f}};\ \ \bar{\mathtt{M}}\} : \{\mathit{mname}(\bar{\mathtt{M}}) \mapsto \bar{\mathbb{C}}\}}$$

Table 5. *Typing rules for method declarations and class declarations*

Rules (T-Get) and (T-Await) define types for `e.get` and `e.await` expressions. In these cases, the type of `e` has to be `Fut(T)` and, correspondingly, the future record type has the pattern $a' \rightsquigarrow \mathbb{s}$. In case of (T-Get), the type of `e.get` is reduced to $(\mathtt{T}, \mathbb{s})$, while, in case of `e.await`, it is not changed. As regards contracts, (T-Get) and (T-Await) extend the contract of `e` with the pairs (a, a') and $(a, a')^{\mathsf{a}}$, respectively, where the index a of the judgment defines the first element of the object name dependency – a stores the root of the object whose method contains the expression `e.get` or `e.await`. The element a' of the object name dependency is the root of the future record of `e`.

Table 5 extends the rules of Table 4 for typing methods and classes. Let $\mathit{mname}(\bar{\mathtt{M}})$ be the sequence of method names in $\bar{\mathtt{M}}$. Rule (T-Method) defines the type and the contract of a method and is similar to the corresponding rule in Featherweight Java. Rule (T-Class) types a class definition associating to it a mapping from method names to method contracts.

Definition 4.2. The *contract class table*, noted CCT, of a `FJf` program (CT, `e`) is a map $\mathtt{C} \mapsto \{\bar{\mathtt{m}} \mapsto \bar{\mathbb{C}}\}$, with $\mathit{dom}(\text{CCT}) = \mathit{dom}(\text{CT})$, that satisfies the following constraints:

there is an environment Γ such that

(a) $\Gamma \vdash \textsc{ct}(\mathtt{C}) : \textsc{cct}(\mathtt{C})$, for every $\mathtt{C}$;

(b) $\Gamma(\mathtt{C.m})$ is the interface of $\textsc{cct}(\mathtt{C})(\mathtt{m})$, for every $\mathtt{C.m}$;

(c) $\textsc{cct}$ is *consistent*, namely

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

Example 4.3. According to the typing rules of Tables 4 and 5, the contract class table for the classes `C`, `D` and `E` of Table 1 is:

$$\begin{array}{rcl} \mathtt{C.m,\ D.m,\ E.m} & \mapsto & a\mathtt{[f:X]()\{\ 0\ \}}\ b\mathtt{[f:X]} \\ \mathtt{D.n} & \mapsto & a\mathtt{[f:X]}(b\mathtt{[f:Y]})\{\mathtt{D.m}\ b\mathtt{[f:Y]()}\rightarrow c\mathtt{[f:Y]}\mathbf{.}(a,b)^{\mathtt{a}}\}\ c\mathtt{[f:Y]} \\ \mathtt{E.n} & \mapsto & a\mathtt{[f:X]}(b\mathtt{[f:Y]})\{\mathtt{E.m}\ b\mathtt{[f:Y]()}\rightarrow c\mathtt{[f:Y]}\mathbf{.}(a,b)\}\ c\mathtt{[f:Y]} \end{array}$$

In particular, the above contract class table is consistent.

In the following, a `FJf` program is a *triple* $(\textsc{ct}, \mathtt{e}, \textsc{cct})$, where $\textsc{ct}$ is the class table, `e` is the expression to evaluate, and $\textsc{cct}$ is a contract class table. It is worth to notice that the consistency predicate constrains method redefinitions in subclasses to retain the same contract of the super-class method. This constraint may be weakened: we defer to future works the issue of studying a sub-contract relation that is correct with respect to class inheritance.

4.3. *Properties of the type system*

The proof of correctness of the type system in Tables 4 and 5 requires additional rules for run-time configurations. To this aim we extend environments Γ to also bind task names `t` to pairs $(\mathtt{Fut(T)}, a \rightsquigarrow \mathbb{r})$ and to include the rules accounting for run-time values $a[\bar{\mathtt{f}} : \bar{\mathbb{r}}]$ and tasks $\mathtt{t} :_a \mathtt{e}$:

$$\textsc{(T-Record)}\quad \frac{\Gamma \vdash_a \bar{\mathtt{v}} : (\bar{\mathtt{T}}, \bar{\mathbb{r}}) ,\ \mathtt{0} \quad \textit{fields}(\mathtt{C}) = \bar{\mathtt{T}}'\ \bar{\mathtt{f}} \quad \bar{\mathtt{T}} \mathrel{<:} \bar{\mathtt{T}}' \quad a' \text{ object name of class } \mathtt{C}}{\Gamma \vdash_a a'[\bar{\mathtt{f}} : \bar{\mathtt{v}}] : (\mathtt{C}, a'[\bar{\mathtt{f}} : \bar{\mathbb{r}}]) ,\ \mathtt{0}}$$

$$\textsc{(T-Task)}\quad \frac{\Gamma \vdash_a \mathtt{e} : (\mathtt{T}, \mathbb{r}) ,\ \mathbb{c} \qquad \Gamma(\mathtt{t}) = (\mathtt{Fut(T)}, a \rightsquigarrow \mathbb{r})}{\Gamma \vdash \mathtt{t} :_a \mathtt{e} : (\mathtt{T}, \mathbb{r}) ,\ \mathbb{c}}$$

$$\textsc{(T-Configuration)}\quad \frac{\mathtt{t} :_a \mathtt{e} \in \mathsf{S} \quad \text{implies} \quad \Gamma \vdash \mathtt{t} :_a \mathtt{e} : (\mathtt{T}, \mathbb{r}) ,\ \mathbb{c}}{\Gamma \vdash \mathsf{S}}$$

We are now ready to state our main result on contracts for the transition relation of Table 3. Additional lemmas and the proofs appear in Appendix A.

Theorem 4.4 (Subject reduction). Let $\Gamma \vdash \mathsf{S}$ and $\mathsf{S} \xrightarrow{a} \mathsf{S}'$. Then there is Γ' such that $\Gamma' \vdash \mathsf{S}'$.

Actually, in Appendix A we prove a stronger result than one stated in the above theorem. Namely, each task in S has the same contract in S' but at a later stage. In particular, we define a *later stage* relation $\trianglelefteq$ (see Definition A.3) between contracts that identifies the consumption of contracts over reduction steps and we show that homonymous tasks in S and S' bear contracts in this *later stage* relation.

4.4. *The contract inference algorithm*

As it is, the type system for contracts in Tables 4 and 5 does not allow to infer the contract class table. In particular it is not specified how to define the interfaces $\Gamma(\texttt{C.m})$ that match with the method contracts in $\textsc{cct}(\texttt{C})(\texttt{m})$. Actually, the solution to this issue is almost standard and rely on unifications (and most general unifiers). In this section we briefly overview our algorithm, discussing a couple of rules.

Definition 4.5. A substitution σ *unifies*

— *a record equation* $\mathbb{r} = \mathbb{s}$ if $\sigma(\mathbb{r})$ and $\sigma(\mathbb{s})$ are identical;
— *an interface equation* $\mathbb{r}(\bar{\mathbb{r}}) \to \mathbb{r}' = \mathbb{s}(\bar{\mathbb{s}}) \to \mathbb{s}'$ if σ unifies $\mathbb{r} = \mathbb{s}$ and $\bar{\mathbb{r}} = \bar{\mathbb{s}}$ and $\mathbb{r}' = \mathbb{s}'$;
— *a record disequation* $\mathbb{s} \succeq \mathbb{r}$ if $\mathbb{s}$ and the substitution instance $\sigma(\mathbb{r})$ are identical;
— *two environments* Γ and Γ' if (i) for every $\texttt{C.m}$, σ unifies $\Gamma(\texttt{C.m}) = \Gamma'(\texttt{C.m})$, and (ii) for every x, σ unifies $\Gamma(x) = \Gamma'(x)$.

A substitution σ is a *more general unifier than* σ' if there is σ'' such that $\sigma' = \sigma \circ \sigma''$. A substitution σ is a *most general unifier*, written mgu, if it is more general than every other unifier.

The algorithm for inferring contracts begins with an environment binding $\texttt{C.m}$ to the interface $a_{\texttt{C.m}}[\texttt{f} : \overline{X_{\texttt{C.m}}}](\overline{Y_{\texttt{C.m}}}) \to Z_{\texttt{C.m}}$ such that

— interfaces associated to different pairs $\texttt{C.m}$ have no (object and record) name in common.

These interfaces are stepwise refined by renaming object names and instantiating record names through unification. To illustrate the algorithm, consider the rule for method invocation:

$$\frac{\begin{array}{c}\Gamma \vdash_a \texttt{e} : (\texttt{C}, \mathbb{r}),\ \mathbb{c} \qquad (\Gamma_i \vdash_a \texttt{e}_i : (\texttt{T}_i, \mathbb{s}_i),\ \mathbb{c}_i)^{i \in 1..n} \\ \mathit{mtype}(\texttt{m}, \texttt{C}) = \bar{\texttt{T}}' \to \texttt{T}' \quad \bar{\texttt{T}} \texttt{ <: } \bar{\texttt{T}}' \\ \sigma \text{ is the mgu such that } (\sigma \Vdash \Gamma = \Gamma_i)^{i \in 1..n} \\ \sigma(\Gamma)(\texttt{C.m}) = \mathbb{r}'(\bar{\mathbb{r}}) \to \mathbb{r}'' \quad \text{implies} \quad \mathbb{r} \succeq \mathbb{r}' \text{ and } \bar{\mathbb{s}} \succeq \bar{\mathbb{r}}\end{array}}{\sigma(\Gamma) \vdash_a \texttt{e!m}(\bar{\texttt{e}}) : (\texttt{Fut}(\texttt{T}'), \sigma(\mathit{root}(\mathbb{r})) \rightsquigarrow \mathbb{r}''[{}^{\sigma(\mathbb{r})}/_{\mathbb{r}'}][{}^{\sigma(\bar{\mathbb{s}})}/_{\bar{\mathbb{r}}}]),\ \sigma(\mathbb{c} \fatsemi \bar{\mathbb{c}}) \fatsemi \texttt{C.m}\ \sigma(\mathbb{r})(\sigma(\bar{\mathbb{s}})) \to \mathbb{r}''[{}^{\sigma(\mathbb{r})}/_{\mathbb{r}'}][{}^{\sigma(\bar{\mathbb{s}})}/_{\bar{\mathbb{r}}}]}$$

The major difference between this rule and (T-Invk) is the presence of distinct environments in the judgments for $\texttt{e}$ and $\bar{\texttt{e}}$ in the premise. These environment have been instantiated in the corresponding proof trees in order to comply with different requirements. For instance, in $\texttt{e}$ it is required to access to a given field of $\texttt{this}$, thus Γ must define its future record, while $\texttt{e}_i$ needs to access to another field, whose future record is defined in Γ_i. To merge the differences in the environments, the rule requires their unification by means of the (most general) substitution σ.

Another interesting rule of the inference algorithm is the one for method declaration:

$$\frac{\begin{array}{c}\Gamma + \bar{\texttt{x}} : (\bar{\texttt{T}}, \bar{\mathbb{s}}) + \texttt{this} : (\texttt{C}, \mathbb{r}) \vdash_a \texttt{e} : (\texttt{T}', \mathbb{r}'),\ \mathbb{c} \qquad \mathit{root}(\mathbb{r}) = a \qquad \texttt{T}' \texttt{ <: } \texttt{T} \\ \texttt{C <: D} \text{ and } \texttt{m} \in \texttt{D} \quad \text{imply} \quad \mathit{mtype}(\texttt{m}, \texttt{D}) = \bar{\texttt{T}} \to \texttt{T}' \\ \sigma \quad \text{unifies} \quad \mathbb{r}(\bar{\mathbb{s}}) \to \mathbb{r}' \succeq \Gamma(\texttt{C.m})\end{array}}{\sigma(\Gamma) \vdash\ \texttt{T m } (\bar{\texttt{T}}\ \bar{\texttt{x}})\{\texttt{return e ;}\} : \mathbb{r}(\bar{\mathbb{s}})\{\mathbb{c}\}\ \mathbb{r}' \quad \textsc{in } \texttt{C}}$$

In this case, it is possible that the interface of $\texttt{m}$ stored in Γ is less specific than the future records for typing $\texttt{this}$, the arguments of the method and the method body $\texttt{e}$. If this is the case, the rule permits to type the method provided Γ is opportunely instantiated.

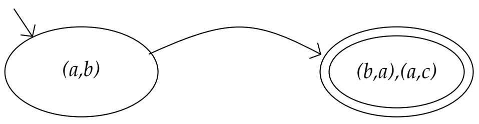

Fig. 2. A simple two-state *lafsa*.

5. The abstract method behaviours: lock analysis finite state automata

Let O be the set of object names and O^{a} be the set of $\mathtt{a}$-tagged object names. Let also $\mathcal{O}^{\mathsf{g/a}} = (\mathsf{O} \times \mathsf{O}) \cup (\mathsf{O}^{\mathsf{a}} \times \mathsf{O}^{\mathsf{a}})$.

Definition 5.1. A *finite state automaton for lock analysis*, in brief *lafsa*, is a tuple $(W, \rightarrow, \mathtt{w}, \mathtt{w}')$ where W – the set of *states* – is a finite subset of $\mathcal{P}(\mathcal{O}^{\mathsf{g/a}})$, $\rightarrow$ – the *transition relation* – is a relation on W, $\mathtt{w}$ and $\mathtt{w}'$ are the *initial state* and the *final state*, respectively – they are element of W.

*Lafsa*s are ranged over by $\mathcal{W}, \mathcal{W}', \cdots$ and are illustrated as finite graphs. For example, in Figure 2 we illustrate a simple two state *lafsa* where the initial state is represented by an entering dangling edge and the final state is represented by a double circle. The object pairs in the states are respectively $\{(a, b)\}$ and $\{(b, a), (a, c)\}$ (the brackets $\{...\}$ are omitted in the figures).

Lafsas have at least one state (that, in the extreme case is initial and final) and retain an order relation $\preceq$ that is defined as follows: $(W_1, \rightarrow_1, \mathtt{w}_1, \mathtt{w}_1') \preceq (W_2, \rightarrow_2, \mathtt{w}_2, \mathtt{w}_2')$ if there is a total map $f : W_1 \rightarrow W_2$ such that

1. for every $\mathtt{w}$: $\mathtt{w} \subseteq f(\mathtt{w})$;
2. $\mathtt{w}_2 = f(\mathtt{w}_1)$;
3. if $\mathtt{w}_1' \rightarrow_1 \mathtt{w}_1''$ then $f(\mathtt{w}_1') \rightarrow_2 f(\mathtt{w}_1'')$;

According to this definition, the *lafsa* $(\{\varnothing\}, \varnothing, \varnothing, \varnothing)$, noted $\mathtt{0}$, is the least element of the order $\preceq$. By definition, $(\{\mathcal{O}^{\mathsf{g/a}}\}, \{(\mathcal{O}^{\mathsf{g/a}}, \mathcal{O}^{\mathsf{g/a}})\}, \mathcal{O}^{\mathsf{g/a}}, \mathcal{O}^{\mathsf{g/a}})$ is the greatest *lafsa*. Given $\preceq$ on *lafsa*, it is standard to define a partial order on cartesian products of *lafsa*s by imposing the coordinatewise order defined by

$$\Big(\mathcal{W}_1, \cdots, \mathcal{W}_k\Big) \preceq \Big(\mathcal{W}_1', \cdots, \mathcal{W}_k'\Big) \quad \stackrel{def}{=} \quad \text{for every } i \,:\, \mathcal{W}_i \preceq \mathcal{W}_i'$$

A number of operations on *lafsa* are defined:

addition: $(W, \rightarrow, \mathtt{w}, \mathtt{w}') \oplus (a, b) \stackrel{def}{=} (\{\mathtt{w}'' \cup \{(a, b)\} \mid \mathtt{w}'' \in W\}, \{(\mathtt{w}'' \cup \{(a, b)\}, \mathtt{w}''' \cup \{(a, b)\}) \mid \mathtt{w}'' \rightarrow \mathtt{w}'''\}, \mathtt{w} \cup \{(a, b)\} \mathtt{w}' \cup \{(a, b)\})$ (similarly for $(W, \rightarrow, \mathtt{w}, \mathtt{w}') \oplus (a^{\mathsf{a}}, b^{\mathsf{a}})$);

sequence: $\curvearrowright$ is defined as follows:

- $\mathtt{0} \curvearrowright (W_2, \rightarrow_2, \mathtt{w}_2, \mathtt{w}_2') \stackrel{def}{=} (W_2, \rightarrow_2, \mathtt{w}_2, \mathtt{w}_2')$;
- $(W_1, \rightarrow_1, \mathtt{w}_1, \mathtt{w}_1') \curvearrowright \mathtt{0} \stackrel{def}{=} (W_1, \rightarrow_1, \mathtt{w}_1, \mathtt{w}_1')$;
- (otherwise) $(W_1, \rightarrow_1, \mathtt{w}_1, \mathtt{w}_1') \curvearrowright (W_2, \rightarrow_2, \mathtt{w}_2, \mathtt{w}_2') \stackrel{def}{=} (W_1 \cup W_2, \rightarrow_1 \cup \rightarrow_2 \cup \{(\mathtt{w}_1', \mathtt{w}_2)\}, \mathtt{w}_1, \mathtt{w}_2')$;

parallel: $(W_1, \rightarrow_1, \mathtt{w}_1, \mathtt{w}_1') \mid (W_2, \rightarrow_2, \mathtt{w}_2, \mathtt{w}_2') \stackrel{def}{=} [(W_1, \rightarrow_1, \mathtt{w}_1, \mathtt{w}_1') \times (W_2, \rightarrow_2, \mathtt{w}_2, \mathtt{w}_2')]^{\mathsf{D}}$, where

- the *cartesian product* $(W_1, \rightarrow_1, \mathtt{w}_1, \mathtt{w}_1') \times (W_2, \rightarrow_2, \mathtt{w}_2, \mathtt{w}_2') \stackrel{def}{=} (W_1 \times W_2, \rightarrow_1 \times \rightarrow_2, (\mathtt{w}_1, \mathtt{w}_2), (\mathtt{w}_1', \mathtt{w}_2'))$, where $\rightarrow_1 \times \rightarrow_2$ is the least set of $((\mathtt{w}, \mathtt{w}'), (\mathtt{w}'', \mathtt{w}'''))$ such that either (i) $\mathtt{w} \rightarrow_1 \mathtt{w}''$ and $\mathtt{w}' = \mathtt{w}'''$ or (ii) $\mathtt{w}' \rightarrow_2 \mathtt{w}'''$ and $\mathtt{w} = \mathtt{w}''$;
- the *downgrading* $[\cdot]^{\mathsf{D}}$ turns pairs $(\mathtt{w}, \mathtt{w}')$ into sets $\mathtt{w} \cup \mathtt{w}'$. Formally, $[(W, \rightarrow, (\mathtt{w}_1, \mathtt{w}_1'), (\mathtt{w}_2, \mathtt{w}_2'))]^{\mathsf{D}} \stackrel{def}{=} (\{\mathtt{w}_1 \cup \mathtt{w}_2 \mid (\mathtt{w}_1, \mathtt{w}_2) \in W\}, \{(\mathtt{w}_1 \cup \mathtt{w}_2, \mathtt{w}_1' \cup \mathtt{w}_2') \mid (\mathtt{w}_1, \mathtt{w}_2) \rightarrow (\mathtt{w}_1', \mathtt{w}_2')\}, \mathtt{w}_1 \cup \mathtt{w}_1', \mathtt{w}_2 \cup \mathtt{w}_2')$.

It is worth to notice that the *lafsa*s obtained from `0` applying the above operations are either `0` or contain at least a nonempty state and with initial and final states that are nonempty. This property follows from the definition of sequence that discharges `0` arguments.

A relevant property for the following theoretical development is the one below. The proof is detailed in the Appendix B.

Proposition 5.2. An operation `op` is monotone with respect to $\preceq$, if, whenever $\mathcal{W}_1 \preceq \mathcal{W}_2$ then $\mathtt{op}(\mathcal{W}_1) \preceq \mathtt{op}(\mathcal{W}_2)$ (similarly for binary operations).

The operations of addition, sequence, and parallel are all monotone with respect to $\preceq$.

The following abstract semantics associates *pairs of lafsas* $\langle \mathcal{W}, \mathcal{W}' \rangle$ to expressions and methods. To illustrate the need of pairs, consider the contract $\mathbb{c} = \mathtt{C.m}\ b[\ ](\ )\,\bullet\,(a, b)$. This contract adds the dependency pair (a, b) to the current state. If the method `m` of class `C` only performs a method invocation, let it be $\mathtt{D.n}\ b[\ ](\ )$ (without any `get` or `await`), then the invocation $\mathtt{C.m}\ b[\ ](\ )$ does not contribute to the current state with other pairs. However it is possible that $\mathtt{D.n}\ b[\ ](\ )$ introduces dependency pairs that affect the *future states* and that have nothing to do with (a, b). The same arguments apply in the cases when `D.n` is an automaton: future dependency pairs are added according to the schedule prescribed by the automaton. Therefore, in order to augment the precision of our (compositional) abstract semantics, we keep separate the above sets of dependencies in the construction of the abstract model by using pairs of *lafsa*s. When the construction terminates, the pair $\langle \mathcal{W}, \mathcal{W}' \rangle$ returned by the following algorithm for the input `FJf` program, must be interpreted as the (single) *lafsa* $\mathcal{W} \curvearrowright \mathcal{W}'$. That is, futures are simply the states after the final state of the first *lafsa* in the pair.

The following operations on pairs of *lafsa*s are used:

pair addition: $\langle \mathcal{W}, \mathcal{W}' \rangle \oplus (a, b) \stackrel{def}{=} \langle \mathcal{W} \oplus (a, b), \mathcal{W}' \rangle$; (similarly for $\langle \mathcal{W}, \mathcal{W}' \rangle \oplus (a^{\mathtt{a}}, b^{\mathtt{a}})$);
pair sequence:

$$\langle \mathcal{W}_1, \mathcal{W}_1' \rangle \,\mathbin{;}\, \langle \mathcal{W}_2, \mathcal{W}_2' \rangle \stackrel{def}{=} \begin{cases} \langle \mathcal{W}_1, \mathcal{W}_1' \mid \mathcal{W}_2' \rangle & \text{if } \mathcal{W}_2 = \mathtt{0} \\ \langle \mathcal{W}_1 \curvearrowright (\mathcal{W}_2 \mid \mathcal{W}_1')\,,\, \mathcal{W}_1' \mid \mathcal{W}_2' \rangle & \text{otherwise} \end{cases}$$

We notice that, by Proposition 5.2, pair addition and pair sequence are monotone (on pairs of *lafsa*s).

5.1. *The finite approximants of abstract method behaviours*

The abstract model of a `FJf` program is obtained by means of a standard fixpoint technique. A basic operation of the technique is the replacement of object names in the states of a *lafsa* with other names. In particular, let $\mathcal{W}[{}^{b}/_{a}]$ be $\mathcal{W}$ where every occurrence of a has been replaced by b. Similarly to what we have done with contracts in Section 4, it is possible to define $\mathcal{W}[{}^{\mathbb{s}}/_{\mathbb{r}}]$. The details are omitted because standard. We let $\langle \mathcal{W}, \mathcal{W}' \rangle[{}^{\mathbb{s}}/_{\mathbb{r}}] = \langle \mathcal{W}[{}^{\mathbb{s}}/_{\mathbb{r}}], \mathcal{W}'[{}^{\mathbb{s}}/_{\mathbb{r}}] \rangle$.

Definition 5.3. Let CCT be a contract class table of a `FJf` program and $(\cdots \langle \mathcal{W}_{\mathtt{C,m}}, \mathcal{W}_{\mathtt{C,m}}' \rangle, \cdots)$ be a tuple of pairs of *lafsa*s such that, for every `C`, `m` in the program there is a corresponding pair $\langle \mathcal{W}_{\mathtt{C,m}}, \mathcal{W}_{\mathtt{C,m}}' \rangle$.

The *lafsa transformation of* CCT, denoted

$$\Big(\cdots, \mathbb{r}_{\mathtt{C,m}}(\overline{\mathbb{s}_{\mathtt{C,m}}}).\mathbb{c}_{\mathtt{C,m}}, \cdots\Big)$$

such that $\mathrm{CCT}(\mathtt{C})(\mathtt{m}) = \mathbb{r}_{\mathtt{C,m}}(\overline{\mathbb{s}_{\mathtt{C,m}}})\ \{\mathbb{c}_{\mathtt{C,m}}\}\ \mathbb{r}_{\mathtt{C,m}}'$, is defined as follows

1. let $\overline{b} = \bigcup_{\texttt{C,m}} \mathit{names}(\mathbb{c}_{\texttt{C,m}}) \setminus \mathit{names}(\mathbb{r}_{\texttt{C,m}}, \overline{\mathbb{s}_{\texttt{C,m}}})$. These are the object names that are created at every *transformation step*;
2. let $\overline{b'}$ be *fresh* object names (they will replace $\overline{b}$);
3. the transformation $\mathbb{r}_{\texttt{C,m}}(\overline{\mathbb{s}_{\texttt{C,m}}}).\mathbb{c}_{\texttt{C,m}}(\cdots \langle \mathcal{W}_{\texttt{C,m}}, \mathcal{W}'_{\texttt{C,m}} \rangle, \cdots)$ gives a pair of *lafsa*s $\langle \mathcal{W}''_{\texttt{C,m}}, \mathcal{W}'''_{\texttt{C,m}} \rangle$ defined as follows:
   - $\Big(\langle \texttt{0}, \texttt{0} \rangle \oplus (a_1^{[\texttt{a}]}, a_2^{[\texttt{a}]})\Big)[\overline{b'}/\overline{b}]$ (W-GAzero)
     if $\mathbb{c}_{\texttt{C,m}} = (a_1, a_2)^{[\texttt{a}]}$;
   - $\Big(\langle \texttt{0}, \mathcal{W}_{\texttt{D,n}} \curvearrowright \mathcal{W}'_{\texttt{D,n}} \rangle\Big)[\mathbb{r}'/\mathbb{r}_{\texttt{D,n}}][\overline{\mathbb{s}'}/\overline{\mathbb{s}_{\texttt{D,n}}}][\mathbb{r}''/\mathbb{r}'_{\texttt{D,n}}][\overline{b'}/\overline{b}]$ (W-Invk)
     if $\mathbb{c}_{\texttt{C,m}} = \texttt{D.n}\ \mathbb{r}'(\overline{\mathbb{s}'}) \to \mathbb{r}''$ and $\textsc{cct}(\texttt{D})(\texttt{n}) = \mathbb{r}_{\texttt{D,n}}\ \{\overline{\mathbb{s}_{\texttt{D,n}}}\}\ \mathbb{r}'_{\texttt{D,n}}$
   - $\Big(\langle \mathcal{W}_{\texttt{D,n}}, \mathcal{W}'_{\texttt{D,n}} \rangle \oplus (a_1^{[\texttt{a}]}, a_2^{[\texttt{a}]})\Big)[\mathbb{r}'/\mathbb{r}_{\texttt{D,n}}][\overline{\mathbb{s}'}/\overline{\mathbb{s}_{\texttt{D,n}}}][\mathbb{r}''/\mathbb{r}'_{\texttt{D,n}}][\overline{b'}/\overline{b}]$ (W-GAinvk)
     if $\mathbb{c}_{\texttt{C,m}} = \texttt{D.n}\ \mathbb{r}'(\overline{\mathbb{s}'}) \to \mathbb{r}'' \bullet (a_1, a_2)^{[\texttt{a}]}$ and $\textsc{cct}(\texttt{D})(\texttt{n}) = \mathbb{r}_{\texttt{D,n}}\ \{\overline{\mathbb{s}_{\texttt{D,n}}}\}\ \mathbb{r}'_{\texttt{D,n}}$
   - $\mathbb{r}_{\texttt{C,m}}(\overline{\mathbb{s}_{\texttt{C,m}}}).\mathbb{c}'_{\texttt{C,m}}(\cdots \langle \mathcal{W}_{\texttt{C,m}}, \mathcal{W}'_{\texttt{C,m}} \rangle, \cdots) \mathbin{⨟} \mathbb{r}_{\texttt{C,m}}(\overline{\mathbb{s}_{\texttt{C,m}}}).\mathbb{c}''_{\texttt{C,m}}(\cdots \langle \mathcal{W}_{\texttt{C,m}}, \mathcal{W}'_{\texttt{C,m}} \rangle, \cdots)$ (W-Seq)
     if $\mathbb{c}_{\texttt{C,m}} = \mathbb{c}'_{\texttt{C,m}} \mathbin{⨟} \mathbb{c}''_{\texttt{C,m}}$.

We notice that, in item 1., we are not assuming that free names in `C.m` and free names in `D.n` of the CCT do not clash. However, a CCT with disjoint sets of free names in the contract methods yields more precise results of our technique. We also notice that the *lafsa* transformation of Definition 5.3, being defined as a composition of monotone operators, see Proposition 5.3, is monotone. Therefore, starting with the tuple $\Big(\cdots \langle {\mathcal{W}_{\texttt{C,m}}}^0, {\mathcal{W}'_{\texttt{C,m}}}^0 \rangle, \cdots\Big) = \Big(\cdots \langle \texttt{0}, \texttt{0} \rangle, \cdots\Big)$, we obtain a non-decreasing sequence (with respect to $\preceq$)

$$\Big(\cdots \langle {\mathcal{W}_{\texttt{C,m}}}^0, {\mathcal{W}'_{\texttt{C,m}}}^0 \rangle, \cdots\Big), \Big(\cdots \langle {\mathcal{W}_{\texttt{C,m}}}^1, {\mathcal{W}'_{\texttt{C,m}}}^1 \rangle, \cdots\Big), \Big(\cdots \langle {\mathcal{W}_{\texttt{C,m}}}^2, {\mathcal{W}'_{\texttt{C,m}}}^2 \rangle, \cdots\Big), \cdots$$

following the standard Knaster-Tarski technique. The pair $\langle \mathcal{W}^i_{\texttt{C,m}}, {\mathcal{W}'_{\texttt{C,m}}}^i \rangle$ is the *i-th finite approximant* of the model of `C.m`. In our *lafsa* domain, because of the creation of new object names, the fixpoint of the above sequence may not exist. For example, the contract $\texttt{C.m}\ b[\,](\,) \to b[\,] \bullet (a, b)$, where $\textsc{cct}(\texttt{C})(\texttt{m}) = a[\,](\,)\ \{\{\texttt{C.m}\ b[\,](\,) \to b[\,] \bullet (a, b)\}\}\ b[\,]$, yields the infinite sequence

$$\Big(\langle \texttt{0}, \texttt{0} \rangle\Big), \Big(\langle \{(b_0, b_1)\}, \texttt{0} \rangle\Big), \Big(\langle \{(b_0, b_1), (b_1, b_2)\}, \texttt{0} \rangle\Big), \Big(\langle \{(b_0, b_1), (b_1, b_2), (b_2, b_3)\}, \texttt{0} \rangle\Big), \cdots$$

where a set of pairs W represents the *one-state/no-transition lafsa* $(\{\texttt{w}\}, \varnothing, \texttt{w}, \texttt{w})$. The above sequence has no upper bound in the domain of *lafsa*s (which are all finite). Therefore, in order to arrive at a decision, we run the Knaster-Tarski technique *on a finite set of object names.* If the n-th approximant is not a fixpoint and consumes the object names, then the $(n + 1)$-th approximant will reuse the same object names used by the n-th approximant, and similarly for the $(n + 2)$-th approximant till a fixpoint is reached. This method is called a *saturation technique at n*. For example, in the case of the above sequence, if the usable object names are $\{b_0, b_1\}$ the saturation technique at 1 terminates in two steps yielding the pair of *lafsa*s $\langle \{(b_0, b_1), (b_1, b_1)\}, \texttt{0} \rangle$.

Definition 5.4. Let $(\textsc{ct}, \texttt{e}, \textsc{cct})$ be an `FJf` program and let $\Big(\cdots \langle {\mathcal{W}_{\texttt{C,m}}}^{n+h}, {\mathcal{W}'_{\texttt{C,m}}}^{n+h} \rangle, \cdots\Big)$ be the fixpoint obtained by the saturation technique at n. The *abstract class table at n*, written $\textsc{act}_{[n]}$, is a map that takes `C.m` and returns $\langle {\mathcal{W}_{\texttt{C,m}}}^{n+h}, {\mathcal{W}'_{\texttt{C,m}}}^{n+h} \rangle$.

Let $(\textsc{ct}, \texttt{e}, \textsc{cct})$ be an `FJf` program, $\mathbb{c}$ be the contract such that $\Gamma + \texttt{this} : (\texttt{Object}, a[\,]) \vdash_a \texttt{e} : (\texttt{T}, \mathbb{r})\ ,\ \mathbb{c}$ where Γ maps `C.m` to the interface of $\textsc{cct}(\texttt{C})(\texttt{m})$, and let $\textsc{act}_{[n]}$ be the corresponding abstract class table at n. The *abstract semantics saturated at n* of `e` is

$$a[\,](\,)\mathbb{c}\Big(\cdots, \textsc{act}_{[n]}(\texttt{C.m}), \cdots\Big)$$

that, by definition, is a pair of *lafsa*s. The abstract semantics of `FJf` programs is discussed through a number of examples.

Example 5.5. It is time to complete our sample code in Table 1 with its abstract class table. By the contract class table detailed in Example 4.3, we compute the abstract class table at 1 (which is the fixpoint).

$$\begin{array}{rcll}
\texttt{C.m, D.m, E.m} & \mapsto & \langle \mathtt{0}, \mathtt{0}\rangle & (\text{since } \mathbb{c}_{\texttt{C,m}} = \mathbb{c}_{\texttt{D,m}} = \mathbb{c}_{\texttt{E,m}} = \mathtt{0}) \\
\texttt{D.n} & \mapsto & \langle \mathtt{0}, \mathtt{0}\rangle \oplus (a^{\mathtt{a}}, b^{\mathtt{a}}) = \langle \{(a^{\mathtt{a}}, b^{\mathtt{a}})\}, \mathtt{0}\rangle & (\text{W-GAINVK}) \\
\texttt{E.n} & \mapsto & \langle \mathtt{0}, \mathtt{0}\rangle \oplus (a, b) = \langle \{(a, b)\}, \mathtt{0}\rangle & (\text{W-GAINVK})
\end{array}$$

By these mappings, we may write the *lafsa*s of the expression `x!n(x)` discussed in Section 2, where `x` is either of class `D` or of class `E`. In particular, they are the *lafsa*s of the above methods `D.n` and `E.n` where a and b are instantiated with a same object name, that is $\langle \{(a^{\mathtt{a}}, a^{\mathtt{a}})\}, \mathtt{0}\rangle$ and $\langle \{(a, a)\}, \mathtt{0}\rangle$, respectively.

Example 5.6. A basic technique for detecting locks uses sets instead of *lafsa*s. However such a technique would returns too much false negatives. For instance, the following extension of class `E` of Table 1

```
class H extends E {
    C p(H b){ return (b!q(this).get; new C(this)!m().get)}
    Fut(C) q(E a){ return (this!n(a));}
}
```

has contract class table

$$\begin{array}{lcl}
\texttt{H.m} & = & \texttt{E.m} \\
\texttt{H.n} & = & \texttt{E.n} \\
\texttt{H.p} & \mapsto & a\texttt{[f:X]}(b\texttt{[f:Y]})\{\ \texttt{H.q}\ \ b\texttt{[f:Y]}(a\texttt{[f:X]})\rightarrow(b \rightsquigarrow e[\texttt{f}:\texttt{X}])\boldsymbol{.}(a,b);\ \ \texttt{C.m}\ \ c\texttt{[f:X]}()\rightarrow d\texttt{[f:X]}\boldsymbol{.}(a,c)\}\ \ d\texttt{[f:X]} \\
\texttt{H.q} & \mapsto & a\texttt{[f:X]}(b\texttt{[f:Y]})\{\ \texttt{E.n}\ \ a\texttt{[f:X]}(\ b\texttt{[f:Y]})\rightarrow(a \rightsquigarrow d[\texttt{f}:\texttt{Y}])\}\ \ a \rightsquigarrow d[\texttt{f}:\texttt{Y}]
\end{array}$$

and abstract class table

$$\begin{array}{lcl}
\texttt{H.p} & \mapsto & \Big(\langle \mathtt{0}, \{(b', a')\}\rangle \oplus (a, b)\Big)[{}^{b[f\,:\,Y]}/{}_{b'[f\,:\,Y']}][{}^{a[f\,:\,X]}/{}_{a'[f\,:\,X']}]\,{}_{9}^{\circ}\Big(\langle \mathtt{0}, \mathtt{0}\}\rangle \oplus (a, c)\Big) \\
 & & = \langle \{(a, b)\}, \{(b, a)\}\rangle \,{}_{9}^{\circ}\, \langle \{(a, c)\}, \mathtt{0}\rangle \\
 & & = \langle \{(a, b)\} \curvearrowright \{(b, a), (a, c)\}, \{(b, a)\}\rangle \\
 & & \\
\texttt{H.q} & \mapsto & \Big(\langle \mathtt{0}, \{(b', a')\}\rangle\Big)[{}^{b[f\,:\,Y]}/{}_{b'[f\,:\,Y']}][{}^{a[f\,:\,X]}/{}_{a'[f\,:\,X']}] \\
 & & = \langle \mathtt{0}, \{(b, a)\}\rangle
\end{array}$$

The leftmost *lafsa* of `H.p` is the one depicted in Figure 2, which has no pair (a, a) in its states. However, if the states are flattened by taking their union, the pair (a, a) shows up immediately (thus signaling a lock, see below).

Example 5.7. This example discusses the approximation performed by our technique due to the object name creation. Consider the class

```
class C extend Object {
  C m() { return ((new C)!m.get ; this) ; }
}
```

The contract class table for this class and the abstract class table at n are respectively

$$\texttt{C.m} \quad \mapsto \quad a[\ ]()\{\ \texttt{C.m}\ b[\ ]() \to a[\ ]\texttt{.}(a,b)\ \}\ a[\ ]$$

$$\texttt{C.m} \quad \mapsto \quad \langle [\{(a_1,a_2),(a_2,a_3),\cdots,(a_{n-1},a_n),(a_n,a_n)\}]^{tc}\texttt{, 0}\rangle$$

We notice that the leftmost *lafsa* of `C.m` has a pair (a_n, a_n), for every index n of saturation (an object-circularity, see below).

6. Deadlock and livelock analysis in `FJf`

We begin with the formal definition of deadlocks and livelocks, generically called locks.

Definition 6.1. Let S be a state containing the tasks $\mathtt{t}_i :^{\ell_i}_{a_i} \mathtt{e}_i$, with $1 \leq i \leq n$ $(n \geq 1)$. Then S is *locked* (e.g., deadlocked or livelocked) if, letting $i+1=1$ when $i=n$:

1. $a_i = a_{i+1}$ implies $\mathtt{e}_i$ is not a value and $\ell_{i+1} = \top$ and $\mathtt{e}_{i+1} = \mathsf{E}_{i+1}[\mathtt{t}_{i+2}.\mathtt{get}]$, for some E_{i+1};
2. $a_i \neq a_{i+1}$ implies (i) $\mathtt{e}_i = \mathsf{E}_i[\mathtt{t}_{i+1}.\mathtt{get}]$ and $\ell_i = \top$ or (ii) $\mathtt{e}_i = \mathsf{E}_i[\mathtt{t}_{i+1}.\mathtt{await}]$, for some E_i;
3. there is i such that $\mathtt{e}_i = \mathsf{E}_i[\mathtt{t}_{i+1}.\mathtt{get}]$, for some E_i.

A state S is *lock-free* if it is not locked and, for every $\mathsf{S} \xrightarrow{a} \mathsf{S}'$, S' is *lock-free*. A program $(\text{CT}, \mathtt{e}, \text{CCT})$ is *lock-free* if its initial configuration is *lock-free*.

This definition identifies locked states by detecting chains of dependencies between tasks that cannot progress. For example

– the state $\mathtt{t}_1 :^{\perp}_{a_1} \mathtt{e}_1, \mathtt{t}_2 :^{\top}_{a_1} \mathtt{t}_2.\mathtt{get}$ is a lock, actually a deadlock, because of 1;
– the state $\mathtt{t}_1 :^{\top}_{a_1} \mathtt{t}_2.\mathtt{get}, \mathtt{t}_2 :^{\top}_{a_2} \mathtt{t}_3.\mathtt{await}$ is a lock, actually a livelock, because of 2;
– the state $\mathtt{t}_1 :^{\top}_{a_1} \mathtt{t}_2.\mathtt{get}, \mathtt{t}_2 :^{\perp}_{a_2} \mathtt{e}_2, \mathtt{t}_3 :^{\top}_{a_2} \mathtt{t}_4.\mathtt{get}, \mathtt{t}_4 :^{\perp}_{a_4} \mathtt{e}_4, \mathtt{t}_5 :^{\top}_{a_4} \mathtt{t}_1.\mathtt{get}$ is a lock because of 1 and 2;

(the three examples all satisfy 3).

We notice that the definition of locked state also takes into account states that are never exhibited by `FJf` programs such as the deadlocks $\mathtt{t} :^{\top}_{a} \mathtt{t}.\mathtt{get}$ and $\mathtt{t}_1 :^{\top}_{a_1} \mathtt{t}_2.\mathtt{get}, \mathtt{t}_2 :^{\top}_{a_2} \mathtt{t}_1.\mathtt{get}$. To produce these states, it would be necessary to create two (or more) tasks *at the same time* and to pass the reference of one task to the other and conversely. This is not possible in `FJf` due to its functional nature. On the contrary, the extension of `FJf` with either field update or variable update admits codes such as

```
this.f1=a!m(this) ; this.f2=b!n(this)
```
(1)

where `f1` and `f2` are two fields of the object `this` and the methods `m` and `n` respectively perform `x.f2.get` and `x.f1.get`, with `x` being their argument. This means that our technique can be profitably used for detecting these misbehaviours in the imperative version of `FJf`. Yet, the imperative version of `FJf` admits states that are locks, in particular livelocks, which are not addressed by Definition 6.1. For example consider the above code (1) when the methods `m` and `n` respectively perform `x.f2.await` and `x.f1.await`, with `x` being their argument. The execution of the program yields a state $\mathtt{t}_1 :^{\perp}_{a_1} \mathtt{t}_2.\mathtt{await}, \mathtt{t}_2 :^{\top}_{a_2} \mathtt{t}_1.\mathtt{await}$, which is a livelock that does not match with Definition 6.1 because of item 3. We might have delivered a weaker definition of lock by removing the third item. However, while this extension has no effect on `FJf` programs, it would require a more complex analysis for separating wrong states as the one above from "safe" states as $\mathtt{t}_1 :^{\perp}_{a_1} \mathtt{t}_2.\mathtt{await}, \mathtt{t}_2 :^{\top}_{a_1} \mathtt{e}$ ($\mathtt{e}$ does not contain $\mathtt{t}_1$). This issue is discussed in some detail in Section 8.

Below we demonstrate the correctness of our analysis technique. Since it is not possible to discover in an exact way the lock-free programs, by "correctness" we mean that not lock-free programs are detected, whilst lock-free programs may be tagged as not lock-free. In order to (partially) measure the distance between lock-freeness and the results of our algorithm, we introduce the notion of object-circularity.

Definition 6.2. A state has

(i) an *object name dependency* (a, b) if it contains the tasks $\mathtt{t} :_a^{\top} \mathsf{E}[\mathtt{t}'.\mathtt{get}], \mathtt{t}' :_b \mathtt{e}$ and $\mathtt{e}$ is not a value;

(ii) an *object name dependency* $(a^{\mathsf{a}}, b^{\mathsf{a}})$ if it contains the tasks $\mathtt{t} :_a \mathsf{E}[\mathtt{t}'.\mathtt{await}], \mathtt{t}' :_b \mathtt{e}$ and $\mathtt{e}$ is not a value.

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

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

A state contains an *object-circularity* if the $\mathtt{get}$-closure of its object name dependencies has a pair (a, a).

Notice that, while in case (i) of definition of object name dependency, the lock of $\mathtt{t}$ is $\top$ because the corresponding expression is a $\mathtt{get}$, in case (ii) the lock of $\mathtt{t}$ may be either $\top$ or $\bot$ because the corresponding expression is an $\mathtt{await}$. It is also worth to remark that the notion of $\mathtt{get}$-closure and object-circularity apply to every subset of $\mathcal{O}^{\mathsf{g/a}}$, therefore to every state of a *lafsa*.

Proposition 6.3. If a state is locked then it has an object-circularity. The converse is false.

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

$$\mathtt{t}_1 :_{a_1} \mathtt{t}_2.\mathtt{await},\ \mathtt{t}_2 :_{a_2}^{\top} \mathtt{t}_3.\mathtt{get},\ \mathtt{t}_3 :_{a_1} a_1.\mathtt{f}$$

where $\mathtt{f}$ is a field of a_1. This state has the object name dependencies $\{(a_1^{\mathsf{a}}, a_2^{\mathsf{a}}), (a_2, a_1)\}$. The $\mathtt{get}$-closure of this set contains the object-circularity (a_2, a_2). However the state is not locked. $\square$

Let $(\textsc{ct}, \mathtt{e}, \textsc{cct})$ be a $\mathtt{FJf}$ program, $\textsc{act}_{[n]}$ be the abstract class table at n, and let S be a state of its operational semantics. Let also $\Gamma \vdash \mathsf{S}$ and, for every $\mathtt{t} :_a \mathtt{e}$ in S, let $\Gamma \vdash \mathtt{t} :_a \mathtt{e} : (\mathtt{T}_\mathtt{t}, \mathbb{r}_\mathtt{t})\ ,\ \mathbb{c}_\mathtt{t}$.

— The abstract semantics of the task $\mathtt{t} :_a \mathtt{e}$, written $[\![\mathtt{t} :_a \mathtt{e}]\!]_{[n]}$, is $a[\bar{\mathtt{f}} : \bar{X}](\,)\mathbb{c}_\mathtt{t}\Big(\cdots, \textsc{act}_{[n]}(\mathtt{C.m}), \cdots\Big)$;

— the abstract semantics of the state S, written $[\![\mathsf{S}]\!]_{[n]}$, is $\prod_{\mathtt{t}:_a\mathtt{e}\in\mathsf{S}} [\![\mathtt{t} :_a \mathtt{e}]\!]_{[n]}$, where $\prod_{i\in 1..m} \langle \mathcal{W}_i, \mathcal{W}'_i \rangle \overset{def}{=} \langle \mathcal{W}_1 \mid \cdots \mid \mathcal{W}_m \mathbf{,}\ \mathcal{W}'_1 \mid \cdots \mid \mathcal{W}'_m \rangle$.

Theorem 6.4. Let $(\textsc{ct}, \mathtt{e}, \textsc{cct})$ be an $\mathtt{FJf}$ program, $\textsc{act}_{[n]}$ be its abstract class table at n, and S be a state of its operational semantics.

1. If S has an object-circularity then at least one state of the lafsas $[\![\mathsf{S}]\!]_{[n]}$ has an object-circularity;
2. if $\mathsf{S} \xrightarrow{a} \mathsf{S}'$ and a state of $[\![\mathsf{S}']\!]_{[n]}$ contains an object-circularity then an object-circularity is already present in a state of $[\![\mathsf{S}]\!]_{[n]}$.

An immediate consequence of Theorem 6.4 is:

Corollary 6.5. Let $(\textsc{ct}, \texttt{e}, \textsc{cct})$ be an `FJf` program and let the abstract semantics saturated at n of `e` be $\langle \mathcal{W}, \mathcal{W}' \rangle$. If no state of $\mathcal{W}$ and $\mathcal{W}'$ manifest an object-circularity then the program is lock-free.

6.1. *Additional remarks on the deadlock and livelock analysis*

We discuss weaknesses of our technique in this sections. The first one is due to name creations. Back to the bugged code of the factorial in Section 1, there are two correct patches:

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

and

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

The first solution, i.e. the method `a_better_fact`, is recognized to be correct by our technique (well, we need to model the `if-then-else` operator, but it is standard: take the union of the object name dependencies in the two branches of the conditional). The second solution, i.e. the method `another_fact`, uses the expedient of performing a `get` on a new object. An invocation of `another_fact` will never produce a lock, while our technique manifests an object-circularity as discussed in Example 5.7.

A different problem follows by the presence in a state of the *lafsa* of pairs $(a, b), (b^{\mathsf{a}}, a^{\mathsf{a}})$. This is an object-circularity, according to Definition 6.2, and this circularity allows us to refuse codes like method `p` of class `G` in Section 3.3. However, in this way we returns false negatives in those case where the `await` operation is performed before the `get`. For example, in the following extension of Table 1:

```
class I extends C {
     C n(I a) { return (a!m().get) ;}
     Fut(C) p() { return ((new I(this.f))!n(this).await) ;}
}
```

The program `(new I(new Object))!p()` does not manifest any lock, however its pair of *lafsa*s – the reader is invited to compute it – has $(a, b), (b^{\mathsf{a}}, a^{\mathsf{a}})$ in its unique state (the second *lafsa* of the pair is `0`).

7. Related work

The proposals in the literature that statically analyze deadlocks are largely based on types. Several proposals concern process calculi (Kobayashi, 2006; Suenaga and Kobayashi, 2007; Suenaga, 2008; Vasconcelos et al., 2009), but there is some contribution also addressing deadlocks in object-oriented programs (Boyapati et al., 2002; Flanagan and Qadeer, 2003; Abadi et al., 2006). 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. An example is the factorial with the recursive invocation underneath a `await.get` – the method `a_better_factorial` in Section 6.1. A further difference with the above works is that we use contracts that are terms in simple (= with finite states) process algebras (Laneve and Padovani, 2007). 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 (Parastatidis and Webber, 2005), which are based on CSP (Brookes et al., 1984) and the pi-calculus (Milner et al., 1992), or of the behavioral types in (Nielson and Nielson, 1994) and (Chaki et al., 2002), which use CCS (Milner, 1982). We expect that finite state abstract descriptions of behaviors can support very powerful verification techniques, even more than the one used in this contribution, such as model checking using tools like the Concurrency Workbench (Cleaveland and Sims, 1996) or like the Mobility Workbench (Victor and Moller, 1994).

Our `FJf` calculus is inspired to the language Creol (Johnsen and Owe, 2007), which has additional scheduling primitives and operations for field update. Contracts similar to the one used in this paper have been already studied in (Giachino and Laneve, 2011) for a language in the family of Creol (Johnsen et al., 2011) with the same purpose of checking deadlocks. The model used in (Giachino and Laneve, 2011) is simpler than *lafsa*s. Apart this source of inspiration, `FJf` is mostly the extension of Featherweight Java (Igarashi et al., 2001) with futures and the `get` operation as described in (Welc et al., 2005). We refer to these papers for additional pointers to the literature.

8. Conclusions

We have developed a technique for the deadlock analysis in a concurrent object-oriented calculus that is based on abstract descriptions of methods behaviours. The abstract descriptions are then analyzed by building a finite-state model and checking the object names dependencies.

This study can be extended in several directions. One direction is the study of techniques for augmenting the accuracy of the fixpoint algorithm on *lafsa*, which is imprecise at the moment when contracts bear name creation. One possibility is to use finite state automata with name creation, such as those in (Montanari and Pistore, 2005), in the modelling step of Section 5 and study deadlocks in these automata. Alternatively, one may try to recognize recurrent patterns of object-name creations and of object name dependencies. Then these patterns should be modeled in some finite way and should be verified whether they are lock-safe or not in the algorithm of Section 6.

Other directions address extensions of the language `FJf`. One of these extensions is to (re)introduce synchronous method invocations for re-entrant recursive calls (usually used for tail recursions). This extension is burdensome because it requires revisions of semantics rules, of contract rules in Table 4, and of the modeling technique, but it is not theoretically difficult. A more complex extension concerns the introduction of field updates and variable updates. In this case there are extensive changes in the semantics of `FJf`, such as the introduction of heaps, and, as discussed in Section 6, there are relevant patches to the Definition 6.1 of locked state, by removing the third item, and Definition 6.2 of `get`-closure. We leave this direction to future research.

Acknowledgements. We thank Davide Sangiorgi and Gianluigi Zavattaro 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

Abadi, M., Flanagan, C., and Freund, S. N. (2006). Types for safe locking: Static race detection for java. *ACM Trans. Program. Lang. Syst.*, 28.

Agha, G. (1986). *Actors: a model of concurrent computation in distributed systems.* MIT Press, Cambridge, MA, USA.

America, P., de Bakker, J., Kok, J. N., and Rutten, J. J. M. M. (1986). Operational semantics of a parallel object-oriented language. In *Proceedings of the 13th Principles of programming languages (POPL '86)*, pages 194–208, New York, NY, USA. ACM.

Boyapati, C., Lee, R., and Rinard, M. (2002). Ownership types for safe program.: preventing data races and deadlocks. In *Proc. OOPSLA '02*, pages 211–230. ACM.

Brookes, S. D., Hoare, C. A. R., and Roscoe, A. W. (1984). A theory of communicating sequential processes. *J. ACM*, 31:560–599.

Chaki, S., Rajamani, S. K., and Rehof, J. (2002). Types as models: model checking message-passing programs. *SIGPLAN Not.*, 37(1):45–57.

Cleaveland, R. and Sims, S. (1996). The ncsu concurrency workbench. In *Computer-Aided Verification (CAV '96)*, volume 1102 of *LNCS*, pages 394–397. Springer-Verlag.

Flanagan, C. and Qadeer, S. (2003). A type and effect system for atomicity. In *In PLDI 03: Programming Language Design and Implementation*, pages 338–349. ACM.

Giachino, E. and Laneve, C. (2011). Analysis of deadlocks in object groups. In *Formal Techniques for Distributed Systems - FMOODS/FORTE*, volume 6722 of *Lecture Notes in Computer Science*, pages 168–182. Springer.

Igarashi, A., Pierce, B. C., and Wadler, P. (2001). Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23:396–450.

Johnsen, E. B., Hähnle, R., Schäfer, J., Schlatte, R., and Steffen, M. (2011). ABS: A core language for abstract behavioral specification. In *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of *LNCS*, pages 142–164. Springer-Verlag.

Johnsen, E. B. and Owe, O. (2007). An asynchronous communication model for distributed concurrent objects. *Software and System Modeling*, 6(1):39–58.

Kobayashi, N. (2006). A new type system for deadlock-free processes. In *Proc. CONCUR 2006*, volume 4137 of *LNCS*, pages 233–247. Springer.

Laneve, C. and Padovani, L. (2007). The *must* preorder revisited. In *Proc. CONCUR 2007*, volume 4703 of *LNCS*, pages 212–225. Springer.

Lavender, G. R. and Schmidt, D. C. (1995). Active Object: an Object Behavioral Pattern for Concurrent Programming. *Proc.Pattern Languages of Programs,*.

Liskov, B. and Shrira, L. (1988). Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of Programming Language design and Implementation (PLDI '88)*, pages 260–267, New York, NY, USA. ACM.

Milner, R. (1982). *A Calculus of Communicating Systems.* Springer.

Milner, R., Parrow, J., and Walker, D. (1992). A calculus of mobile processes, ii. *Inf. and Comput.*, 100:41–77.

Montanari, U. and Pistore, M. (2005). History-dependent automata: An introduction. In *Formal Methods for the Design of Computer, Communication, and Software Systems*, volume 3465 of *LNCS*, pages 1–28. Springer.

Niehren, J., Schwinghammer, J., and Smolka, G. (2006). A concurrent lambda calculus with futures. *Theor. Comput. Sci.*, 364.

Nielson, H. R. and Nielson, F. (1994). Higher-order concurrent programs with finite communication topology. In *Proc. POPL '94*, pages 84–97. ACM.

Parastatidis, S. and Webber, J. (2005). *MEP SSDL Protocol Framework*. `http://ssdl.org`.

Suenaga, K. (2008). Type-based deadlock-freedom verification for non-block-structured lock primitives and mutable references. In *Programming Languages and Systems*, volume 5356 of *LNCS*, pages 155–170. Springer.

Suenaga, K. and Kobayashi, N. (2007). Type-based analysis of deadlock for a concurrent calculus with interrupts. In *Programming Languages and Systems*, volume 4421 of *LNCS*, pages 490–504. Springer.

Torgersen, M. (2010). Asyncrony in .NET.

Vasconcelos, V. T., Martins, F., and Cogumbreiro, T. (2009). Type inference for deadlock detection in a multithreaded polymorphic typed assembly language. In *Proc. PLACES'09*, volume 17 of *EPTCS*, pages 95–109.

Victor, B. and Moller, F. (1994). The Mobility Workbench — a tool for the π-calculus. In Dill, D., editor, *CAV'94: Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 428–440. Springer-Verlag.

Welc, A., Jagannathan, S., and Hosking, A. (2005). Safe futures for java. In *Proceedings of the 20th Object-oriented programming, systems, languages, and applications (OOPSLA '05)*, pages 439–453, New York, NY, USA. ACM.

Yonezawa, A., editor (1990). *ABCL: an object-oriented concurrent system*. MIT Press, Cambridge, MA, USA.

Appendix A. Proof of the Subject Reduction Theorem.

In the following we consider *redexes* as the active (to be reduced) part in an expression:

$$r ::= a[\bar{\mathtt{f}} : \bar{\mathtt{v}}].\mathtt{f} \mid a[\bar{\mathtt{f}} : \bar{\mathtt{v}}]!\mathtt{m}(\bar{\mathtt{v}}') \mid \mathtt{new}\ \mathtt{C}(\bar{\mathtt{v}}) \mid \mathtt{t.get} \mid \mathtt{t.await}.$$

Lemma A.1. Given a well-typed runtime expression $\mathtt{e}$ that is not a value and different for $\mathtt{x}$, there exist E and r such that $\mathtt{e} = \mathsf{E}[r]$.

Proof.

<u>**Case** $\mathtt{e.f}$</u>**.**Either $\mathtt{e} = a[\bar{\mathtt{f}} : \bar{\mathtt{v}}]$, for some a, $\bar{\mathtt{f}}$ and $\bar{\mathtt{v}}$, then the evaluation context we seek is the empty one, or we can apply the induction hypothesis to $\mathtt{e}$ deriving that there is an evaluation context E such that $\mathtt{e} = \mathsf{E}[r]$ for some redex r. Therefore $\mathsf{E}.\mathtt{f}$ is the evaluation context for $\mathtt{e.f}$.

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

<u>**Case** $\mathtt{e};\mathtt{e}'$</u> **.**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 $\mathtt{e} = \mathsf{E}[r]$ for some 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.

$\square$

Lemma A.2. If $\Gamma \vdash_a \mathsf{E}[r] : (\mathtt{T}, \mathbb{r})\,,\ \mathbb{c}$, then $\Gamma \vdash_a r : (\mathtt{T}', \mathbb{r}')\,,\ \mathbb{c}'$, for some $\mathtt{T}', \mathbb{r}'$, and $\mathbb{c}'$ s.t. $\mathbb{c} = \mathbb{c}' \fatsemi \mathbb{c}''$ or $\mathbb{c} = \bar{0} \fatsemi \mathbb{c}' \fatsemi \mathbb{c}''$ or $\mathbb{c} = \mathbb{c}' \between (a, a'); \mathbb{c}''$ or $\mathbb{c} = \mathbb{c}' \between (a, a')^{\mathtt{a}}; \mathbb{c}''$, for some $a', \mathbb{c}''$.

Proof. The proofs is by straightforward induction on the derivation of $\Gamma \vdash_a \mathsf{E}[r] : (\mathtt{T}, \mathbb{r})\,,\ \mathbb{c}$.
$\square$

Below we demonstrate that if $\mathsf{S} \xrightarrow{a} \mathsf{S}'$ then every task in S has the same contract in S' but *at a later stage*. The notion of "later stage" is expressed by the operation $\trianglelefteq$ on contracts.

Definition A.3 ($\trianglelefteq$). Let $\mathbb{c} \trianglelefteq \mathbb{c}'$, say $\mathbb{c}$ *is at a later stage than* $\mathbb{c}'$, be the least relation such that

$$\overset{(1)}{\mathbb{c} \trianglelefteq \mathbb{c}} \qquad \overset{(2)}{0 \trianglelefteq \mathbb{c}} \qquad \overset{(3)}{(a, a')^{[\mathtt{a}]} \trianglelefteq \mathtt{C.m}\ \mathbb{r}(\bar{\mathbb{s}}) \to \mathbb{s} \bullet (a, a')^{[\mathtt{a}]}}$$

$$\overset{(4)}{\frac{\mathtt{C} <: \mathtt{D}}{\mathtt{C.m} : \mathbb{r}(\bar{\mathbb{r}}) \to \mathbb{s} \trianglelefteq \mathtt{D.m} : \mathbb{r}(\bar{\mathbb{r}}) \to \mathbb{s}}} \qquad \overset{(5)}{\frac{\mathbb{c}_1 \trianglelefteq \mathbb{c}_2}{\mathbb{c}_1 \fatsemi \mathbb{c} \trianglelefteq \mathbb{c}_2 \fatsemi \mathbb{c}}}$$

Lemma A.4. Let $\mathsf{E}[r]$ be such that $\Gamma \vdash_a \mathsf{E}[r] : (\mathtt{T}, \mathbb{r})\,,\ \mathbb{c}$ and $\Gamma \vdash_a r : (\mathtt{T}', \mathbb{r}')\,,\ \mathbb{c}'$ and $\Gamma \vdash_a \mathtt{v} : (\mathtt{T}'', \mathbb{r}')\,,\ 0$, with $\mathtt{T}'' <: \mathtt{T}'$. Then $\Gamma \vdash_a \mathsf{E}[\mathtt{v}] : (\mathtt{T}''', \mathbb{r})\,,\ \mathbb{c}''$ such that $\mathtt{T}''' <: \mathtt{T}$, $\mathbb{c}'' \trianglelefteq \mathbb{c}$.

Proof. By induction on evaluation contexts E.

Case $[\ ]$. Immediate.

Case $\mathsf{E}!\mathtt{m}(\bar{\mathtt{e}})$. Let $\Gamma \vdash_a \mathsf{E}[r]!\mathtt{m}(\bar{\mathtt{e}}) : (\mathtt{T}, \mathbb{r})\,,\ \mathbb{c}$ where $\Gamma \vdash_a r : (\mathtt{T}', \mathbb{r}')\,,\ \mathbb{c}'$, and let $\mathtt{v}$ be such that $\Gamma \vdash_a \mathtt{v} : (\mathtt{T}'', \mathbb{r}')\,,\ 0$, with $\mathtt{T}'' <: \mathtt{T}'$. By typing rule (T-Invk) we have that

$$\begin{array}{l}
\Gamma(\mathtt{C.m}) = a'[\bar{\mathtt{f}} : \bar{\mathbb{r}}](\bar{\mathbb{s}}) \to \mathbb{s} \\
\Gamma \vdash_a \mathsf{E}[r] : (\mathtt{C}, \sigma(a'[\bar{\mathtt{f}} : \bar{\mathbb{r}}]))\,,\ \mathbb{c}_0 \quad \Gamma \vdash_a \bar{\mathtt{e}} : (\bar{\mathtt{T}}, \sigma(\bar{\mathbb{s}}))\,,\ \bar{\mathbb{c}} \\
\mathit{mtype}(\mathtt{m}, \mathtt{C}) = \bar{\mathtt{T}}' \to \mathtt{T}_0 \quad \bar{\mathtt{T}} <: \bar{\mathtt{T}}' \\
\mathtt{T} = \mathtt{Fut}(\mathtt{T}_0) \quad \mathbb{c} = \mathbb{c}_0 \fatsemi \bar{\mathbb{c}} \fatsemi \mathtt{C.m}\ \sigma(a'[\bar{\mathtt{f}} : \bar{\mathbb{r}}])(\sigma(\bar{\mathbb{s}})) \to \sigma(\mathbb{s}) \quad \mathbb{r} = \sigma(a') \rightsquigarrow \sigma(\mathbb{s})
\end{array}$$

By induction hypothesis on E we have that $\Gamma \vdash_a \mathsf{E}[\mathtt{v}] : (\mathtt{D}, \sigma(a'[\bar{\mathtt{f}} : \bar{\mathbb{r}}]))\,,\ \mathbb{c}_0'$ for some $\mathtt{D} <: \mathtt{C}$ and $\mathbb{c}_0' \trianglelefteq \mathbb{c}_0$.
Since we consider a well-typed expression w.r.t. a well-typed class table, then if $\mathtt{m}$ is defined in $\mathtt{D}$, then $\mathit{mtype}(\mathtt{m}, \mathtt{C}) = \mathit{mtype}(\mathtt{m}, \mathtt{D})$ and $\Gamma(\mathtt{C.m}) =^{\alpha} \Gamma(\mathtt{D.m})$. Applying typing rule (T-Invk) we have that $\Gamma \vdash_a \mathsf{E}[\mathtt{v}]!\mathtt{m}(\bar{\mathtt{e}}) : (\mathtt{Fut}(\mathtt{T}_0), \sigma(a') \rightsquigarrow \sigma(\mathbb{s}))\,,\ \mathbb{c}_1$, where $\mathbb{c}_1 = \mathbb{c}_0' \fatsemi \bar{\mathbb{c}} \fatsemi \mathtt{D.m}\ \sigma(a'[\bar{\mathtt{f}} : \bar{\mathbb{r}}])(\sigma(\bar{\mathbb{s}})) \to \sigma(\mathbb{s})$ and, by Definition A.3, $\mathbb{c}_1 \trianglelefteq \mathbb{c}$.

Case $b[\bar{\mathtt{f}} : \bar{\mathtt{v}}].\mathtt{m}(\bar{\mathtt{v}}', \mathsf{E}, \bar{\mathtt{e}})$. Let $\Gamma \vdash_a b[\bar{\mathtt{f}} : \bar{\mathtt{v}}].\mathtt{m}(\bar{\mathtt{v}}', \mathsf{E}[r], \bar{\mathtt{e}}) : (\mathtt{T}, \mathbb{r})\,,\ \mathbb{c}$ where $\Gamma \vdash_a r : (\mathtt{T}', \mathbb{r}')\,,\ \mathbb{c}'$, and let $\mathtt{v}$ be such that $\Gamma \vdash_a \mathtt{v} : (\mathtt{T}'', \mathbb{r}')\,,\ 0$, with $\mathtt{T}'' <: \mathtt{T}'$. By typing rule (T-Invk) we have that

$$\begin{array}{l}
\Gamma(\mathtt{C.m}) = a'[\bar{\mathtt{f}} : \bar{\mathbb{r}}](\bar{\mathbb{s}}, \mathbb{s}_0, \bar{\mathbb{s}}') \to \mathbb{s} \\
\Gamma \vdash_a b[\bar{\mathtt{f}} : \bar{\mathtt{v}}] : (\mathtt{C}, \sigma(a'[\bar{\mathtt{f}} : \bar{\mathbb{r}}]))\,,\ 0 \\
\Gamma \vdash_a \bar{\mathtt{v}}' : (\bar{\mathtt{T}}', \sigma(\bar{\mathbb{s}}))\,,\ 0 \quad \Gamma \vdash_a \bar{\mathtt{e}} : (\bar{\mathtt{T}}'', \sigma(\bar{\mathbb{s}}'))\,,\ \bar{\mathbb{c}} \quad \Gamma \vdash_a \mathsf{E}[r] : (\mathtt{T}''', \sigma(\mathbb{s}_0))\,,\ \mathbb{c}_0 \\
\mathit{mtype}(\mathtt{m}, \mathtt{C}) = \bar{\mathtt{T}} \to \mathtt{T}_0 \quad \bar{\mathtt{T}}'\mathtt{T}'''\bar{\mathtt{T}}'' <: \bar{\mathtt{T}} \\
\mathtt{T} = \mathtt{Fut}(\mathtt{T}_0) \quad \mathbb{c} = \bar{0} \fatsemi \mathbb{c}_0 \fatsemi \bar{\mathbb{c}} \fatsemi \mathtt{C.m}\ \sigma(a'[\bar{\mathtt{f}} : \bar{\mathbb{r}}])(\sigma(\bar{\mathbb{s}}, \mathbb{s}_0, \bar{\mathbb{s}}')) \to \sigma(\mathbb{s}) \quad \mathbb{r} = \sigma(a') \rightsquigarrow \sigma(\mathbb{s})
\end{array}$$

By induction hypothesis on E we have that $\Gamma \vdash_a \mathsf{E}[\mathtt{v}] : (\mathtt{T}_1''', \sigma(\mathbb{s}_0))\,,\ \mathbb{c}_0'$ for some $\mathtt{T}_1''' <: \mathtt{T}'''$

and $\mathbb{c}_0'$ such that $\mathbb{c}_0' \trianglelefteq \mathbb{c}_0$. Applying typing rule (T-Invk) we have that $\Gamma \vdash_a b[\bar{\mathtt{f}} : \bar{\mathtt{v}}].\mathtt{m}(\bar{\mathtt{v}}', \mathsf{E}[\mathtt{v}], \bar{\mathtt{e}}) : (\mathtt{Fut}(\mathtt{T}_0), \sigma(a') \rightsquigarrow \sigma(\mathbb{s}))\,,\ \mathbb{c}_1$, where $\mathbb{c}_1 = \bar{\mathtt{0}} ⨟ \mathbb{c}_0' ⨟ \bar{\mathbb{c}} ⨟ \mathtt{C.m}\ \sigma(a'[\bar{\mathtt{f}} : \bar{\mathbb{r}}])(\sigma(\bar{\mathbb{s}}, \mathbb{s}_0, \bar{\mathbb{s}}')) \to \sigma(\mathbb{s})$ and, by Definition A.3, $\mathbb{c}_1 \trianglelefteq \mathbb{c}$.

Remaining Cases. Similar to the previous ones.

□

Lemma A.5. Let $\Gamma \vdash_a \mathtt{e} : (\mathtt{T}, \mathbb{r})\,,\ \mathbb{c}$. If $\Gamma \subseteq \Gamma'$, then $\Gamma' \vdash_a \mathtt{e} : (\mathtt{T}, \mathbb{r})\,,\ \mathbb{c}$

Proof. Straightforward induction on the derivation of $\Gamma \vdash_a \mathtt{e} : (\mathtt{T}, \mathbb{r})\,,\ \mathbb{c}$. □

Lemma A.6 (Substitution). If $\Gamma, \bar{\mathtt{x}} : (\bar{\mathtt{C}}, \bar{\mathbb{r}}) \vdash_a \mathtt{e} : (\mathtt{T}, \mathbb{r})\,,\ \mathbb{c}$, and $\Gamma \vdash_a \bar{\mathtt{v}} : (\bar{\mathtt{D}}, \bar{\mathbb{r}})\,,\ \bar{\mathtt{0}}$ where $\bar{\mathtt{D}} \mathrel{\mathtt{<:}} \bar{\mathtt{C}}$, then $\Gamma \vdash_a \mathtt{e}[{}^{\bar{\mathtt{v}}}/_{\bar{\mathtt{x}}}] : (\mathtt{T}', \mathbb{r})\,,\ \mathbb{c}$, for some $\mathtt{T}' \mathrel{\mathtt{<:}} \mathtt{T}$.

Proof. By straightforward induction on the derivation of $\Gamma, \bar{\mathtt{x}} : (\bar{\mathtt{C}}, \bar{\mathbb{r}}) \vdash_a \mathtt{e} : (\mathtt{T}, \mathbb{r})\,,\ \mathbb{c}$. □

Lemma A.7. If $\mathit{mtype}(\mathtt{m}, \mathtt{C}) = \bar{\mathtt{T}} \to \mathtt{T}$, $\mathit{mbody}(\mathtt{m}, \mathtt{C}) = \bar{\mathtt{x}}.\mathtt{e}$, and $a[\bar{\mathtt{f}} : \bar{\mathbb{r}}](\bar{\mathbb{s}})\ \{\mathbb{c}\}\ \mathbb{s} \in \textsc{cct}(\mathtt{m})$, then, for some $\mathtt{D}$, s.t. $\mathtt{C} \mathrel{\mathtt{<:}} \mathtt{D}$, there exist $\mathtt{T}' \mathrel{\mathtt{<:}} \mathtt{T}$ and σ such that

$$\mathtt{D.m} : a[\bar{\mathtt{f}} : \bar{\mathbb{r}}](\bar{\mathbb{s}}) \to \mathbb{s},\ \bar{\mathtt{x}} : (\bar{\mathtt{T}}, \sigma(\bar{\mathbb{s}})),\ \mathtt{this} : \mathtt{D}, \sigma(a[\bar{\mathtt{f}} : \bar{\mathbb{r}}](\bar{\mathbb{s}})) \vdash_a \mathtt{e} : (\mathtt{T}', \sigma(\mathbb{s}))\,,\ \sigma(\mathbb{c}).$$

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

Proof of Theorem 4.4 Let $\Gamma \vdash \mathsf{S}$ and $\mathsf{S} \longrightarrow \mathsf{S}'$. Then there is Γ' such that $\Gamma' \vdash \mathsf{S}'$.

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

Case (Invk)**.** Then

$$\mathtt{t} :_a^{\top} \mathsf{E}[b[\bar{\mathtt{f}} : \bar{\mathtt{v}}]!\mathtt{m}(\bar{\mathtt{v}}')] \xrightarrow{a} \mathtt{t} :_a^{\top} \mathsf{E}[\mathtt{t}'],\ \ \mathtt{t}' :_b^{\bot} \mathtt{e}[{}^{b[\bar{\mathtt{f}} : \bar{\mathtt{v}}]}/_{\mathtt{this}}][{}^{\bar{\mathtt{v}}'}/_{\bar{\mathtt{x}}}]$$

where $\mathit{class}(b) = \mathtt{C}$ and $\mathit{mbody}(\mathtt{m}, \mathtt{C}) = \bar{\mathtt{x}}.\mathtt{e}$ and $\mathtt{t}' = \mathit{freshtask}(\ )$.
Since $\Gamma \vdash \mathtt{t} :_a^{\top} \mathsf{E}[b[\bar{\mathtt{f}} : \bar{\mathtt{v}}]!\mathtt{m}(\bar{\mathtt{v}}')]$, by rule (T-Configuration) we get $\Gamma \vdash_a \mathtt{t} :_a^{\top} \mathsf{E}[b[\bar{\mathtt{f}} : \bar{\mathtt{v}}]!\mathtt{m}(\bar{\mathtt{v}}')] : (\mathtt{T}, \mathbb{r})\,,\ \mathbb{c}$, for some $\mathtt{T}$, $\mathbb{r}$ and $\mathbb{c}$. By rule (T-Task) we have

$$\Gamma \vdash_a \mathsf{E}[b[\bar{\mathtt{f}} : \bar{\mathtt{v}}]!\mathtt{m}(\bar{\mathtt{v}}')] : (\mathtt{T}, \mathbb{r})\,,\ \mathbb{c} \qquad \Gamma(\mathtt{t}) = (\mathtt{Fut}(\mathtt{T}), a \rightsquigarrow \mathbb{r})$$

By Lemma A.2 and typing rule (T-Invk) we have that there exists a substitution σ such that $\Gamma \vdash_a b[\bar{\mathtt{f}} : \bar{\mathtt{v}}]!\mathtt{m}(\bar{\mathtt{v}}') : (\mathtt{Fut}(\mathtt{T}'), \sigma(a') \rightsquigarrow \sigma(\mathbb{r}'))\,,\ \mathtt{0} ⨟ \bar{\mathtt{0}} ⨟ \mathtt{C.m}\ \sigma(a'[\bar{\mathtt{f}} : \bar{\mathbb{r}}])(\sigma(\bar{\mathbb{s}})) \to \sigma(\mathbb{r}')$ and:

$$\begin{array}{ll} \Gamma \vdash_a b[\bar{\mathtt{f}} : \bar{\mathtt{v}}] : (\mathtt{C}, \sigma(a'[\bar{\mathtt{f}} : \bar{\mathbb{r}}]))\,,\ \mathtt{0} & \Gamma \vdash_a \bar{\mathtt{v}} : (\bar{\mathtt{T}}, \sigma(\bar{\mathbb{s}}))\,,\ \bar{\mathtt{0}} \\ \Gamma(\mathtt{C.m}) = a'[\bar{\mathtt{f}} : \bar{\mathbb{r}}](\bar{\mathbb{s}}) \to \mathbb{r}' & \mathit{mtype}(\mathtt{m}, \mathtt{C}) = \bar{\mathtt{T}}' \to \mathtt{T}' \quad \bar{\mathtt{T}} \mathrel{\mathtt{<:}} \bar{\mathtt{T}}' \end{array}$$

where $\sigma(a') = b$.
By Lemmas A.5, A.6, and A.7, we get $\Gamma,\ \bar{\mathtt{x}} : (\bar{\mathtt{T}}', \sigma(\bar{\mathbb{s}})),\ \mathtt{this} : (\mathtt{D}, \sigma(a'[\bar{\mathtt{f}} : \bar{\mathbb{r}}])) \vdash_a \mathtt{e}[{}^{b[\bar{\mathtt{f}} : \bar{\mathtt{v}}]}/_{\mathtt{this}}][{}^{\bar{\mathtt{v}}'}/_{\bar{\mathtt{x}}}] : (\mathtt{T}'', \sigma(\mathbb{r}'))\,,\ \sigma(\mathbb{c}')$, with $\mathtt{C} \mathrel{\mathtt{<:}} \mathtt{D}$, $\mathtt{T}'' \mathrel{\mathtt{<:}} \mathtt{T}'$, and $\mathbb{c}' \in \textsc{cct}(\mathtt{C.m})$.
Let Γ' be such that $\Gamma' = \Gamma, \mathtt{t}' : (\mathtt{Fut}(\mathtt{T}''), b \rightsquigarrow \sigma(\mathbb{r}')), \mathtt{0},\ \bar{\mathtt{x}} : (\bar{\mathtt{T}}', \sigma(\bar{\mathbb{s}})),\ \mathtt{this} : (\mathtt{D}, \sigma(a'[\bar{\mathtt{f}} : \bar{\mathbb{r}}]))$.
By Lemma A.5 we have $\Gamma' \vdash_a \mathtt{e}[{}^{b[\bar{\mathtt{f}} : \bar{\mathtt{v}}]}/_{\mathtt{this}}][{}^{\bar{\mathtt{v}}'}/_{\bar{\mathtt{x}}}] : (\mathtt{T}'', \sigma(\mathbb{r}'))\,,\ \sigma(\mathbb{c}')$ and $\Gamma' \vdash_a \mathtt{t} :_a^{\top} \mathsf{E}[b[\bar{\mathtt{f}} : \bar{\mathtt{v}}]!\mathtt{m}(\bar{\mathtt{v}}')] : (\mathtt{T}, \mathbb{r})\,,\ \mathbb{c}$ and $\Gamma' \vdash_a b[\bar{\mathtt{f}} : \bar{\mathtt{v}}]!\mathtt{m}(\bar{\mathtt{v}}') : (\mathtt{Fut}(\mathtt{T}'), \sigma(a') \rightsquigarrow \sigma(\mathbb{r}'))\,,\ \mathtt{0} ⨟ \bar{\mathtt{0}} ⨟ \mathtt{C.m}\ \sigma(a'[\bar{\mathtt{f}} : \bar{\mathbb{r}}])(\sigma(\bar{\mathbb{s}})) \to \sigma(\mathbb{r}')$. By Lemma A.4 and rule (T-Configuration) we get the result:

$$\Gamma' \vdash_a \mathtt{t} :_a^{\top} \mathsf{E}[\mathtt{t}'] : (\mathtt{T}''', \mathbb{r})\,,\ \mathbb{c}'' \qquad \Gamma' \vdash \mathtt{t} :_a^{\top} \mathsf{E}[\mathtt{t}'],\ \ \mathtt{t}' :_b^{\bot} \mathtt{e}[{}^{b[\bar{\mathtt{f}} : \bar{\mathtt{v}}]}/_{\mathtt{this}}][{}^{\bar{\mathtt{v}}'}/_{\bar{\mathtt{x}}}]$$

where $\mathtt{T}''' \mathrel{\mathtt{<:}} \mathtt{T}$ and $\mathbb{c}'' \trianglelefteq \mathbb{c}$.

Case (GET). Then

$$\mathtt{t} :_a^{\top} \mathsf{E}[\mathtt{t}'.\mathtt{get}],\ \mathtt{t}' :_b \mathtt{v} \xrightarrow{a} \mathtt{t} :_a^{\top} \mathsf{E}[\mathtt{v}],\ \mathtt{t}' :_b \mathtt{v}$$

Since $\Gamma \vdash \mathtt{t} :_a^{\top} \mathsf{E}[\mathtt{t}'.\mathtt{get}],\ \mathtt{t}' :_b \mathtt{v}$, by rule (T-CONFIGURATION) we get $\Gamma \vdash_a \mathtt{t} :_a^{\top} \mathsf{E}[\mathtt{t}'.\mathtt{get}] : (\mathtt{T}, \mathbb{r})\,,\ \mathbb{c}$, and $\Gamma \vdash_b \mathtt{t}' :_b \mathtt{v} : (\mathtt{T}', \mathbb{r}')\,,\ \mathtt{0}$ for some $\mathtt{T}$, $\mathbb{r}$, $\mathtt{T}'$, $\mathbb{r}'$ and $\mathbb{c}$. Then by rule (T-TASK) we have that $\Gamma \vdash_a \mathsf{E}[\mathtt{t}'.\mathtt{get}] : (\mathtt{T}, \mathbb{r})\,,\ \mathbb{c}$, and $\Gamma \vdash_b \mathtt{v} : (\mathtt{T}', \mathbb{r}')\,,\ \mathtt{0}$, and $\Gamma(\mathtt{t}) = (\mathtt{Fut}(\mathtt{T}), a \rightsquigarrow \mathbb{r})$, and $\Gamma(\mathtt{t}')(\mathtt{Fut}(\mathtt{T}'), b \rightsquigarrow \mathbb{r}')$. By Lemma A.2 and typing rule (T-GET), we have $\Gamma \vdash_a \mathtt{t}'.\mathtt{get} : (\mathtt{T}', \mathbb{r}')\,,\ \mathtt{0} \between (a, b)$. By Lemma A.4 and rule (T-CONFIGURATION) we get the result.

Remaining cases. Straightforward.

□

Appendix B. Monotonicity of lafsa operations

For the sake of readability, we restate the proposition.

Proposition 5.2. An operation op is monotone with respect to $\preceq$, if, whenever $\mathcal{W}_1 \preceq \mathcal{W}_2$ then $\mathtt{op}(\mathcal{W}_1) \preceq \mathtt{op}(\mathcal{W}_2)$ (similarly for binary operations).

The operations of addition, sequence, and parallel are all monotone with respect to $\preceq$.

Proof. The proof is standard for the operations of addition and sequence. Let us focus on parallel. We may reduce to the subcases

monotonocity from the left: if $\mathcal{W}_A \preceq \mathcal{W}_A'$ then $\mathcal{W}_A \mid \mathcal{W}_B \preceq \mathcal{W}_A' \mid \mathcal{W}_B$;
monotonocity from the right: if $\mathcal{W}_B \preceq \mathcal{W}_B'$ then $\mathcal{W}_A \mid \mathcal{W}_B \preceq \mathcal{W}_A \mid \mathcal{W}_B'$;

and we detail the monotonicity from the left; the other one being similar. Let $\mathcal{W}_A = (W_A, \rightarrow_A, \mathtt{w}_A, \mathtt{w}_A')$, $\mathcal{W}_B = (W_B, \rightarrow_B, \mathtt{w}_B, \mathtt{w}_B')$, and let f be the map such that $\mathcal{W}_A \preceq \mathcal{W}_B$. We must prove that $\mathcal{W}_A \mid \mathcal{W}_C \preceq \mathcal{W}_B \mid \mathcal{W}_C$, that is there is a map g from $\mathcal{W}_A \mid \mathcal{W}_C = (W_{A|C}, \rightarrow_{A|C}, \mathtt{w}_{A|C}, \mathtt{w}_{A|C}')$ to $\mathcal{W}_B \mid \mathcal{W}_C = (W_{B|C}, \rightarrow_{B|C}, \mathtt{w}_{B|C}, \mathtt{w}_{B|C}')$ such that:

1. for every $\mathtt{w}$ in $W_{A|C}$ we have $\mathtt{w} \subseteq g(\mathtt{w})$;
2. $\mathtt{w}_{B|C} = g(\mathtt{w}_{A|C})$;
3. if $\mathtt{w}' \rightarrow_{A|C} \mathtt{w}''$ then $g(\mathtt{w}') \rightarrow_{B|C} g(\mathtt{w}'')$;

Let g be the function

$$g \ : \ [(\mathtt{w}_A, \mathtt{w}_C)]^D \ \mapsto \ [(f(\mathtt{w}_A), \mathtt{w}_C)]^D \ .$$

The above items 1. and 2. are immediate. As regards 3., we must show that $\mathtt{w} \rightarrow_{A|C} \mathtt{w}'$ implies $g(\mathtt{w}) \rightarrow_{B|C} g(\mathtt{w}')$. By definition $\mathtt{w}, \mathtt{w}'$ must be $\mathtt{w}_1 \cup \mathtt{w}_2$ and $\mathtt{w}_1' \cup \mathtt{w}_2'$, respectively, with $(\mathtt{w}_1, \mathtt{w}_2) \rightarrow_A \times \rightarrow_C (\mathtt{w}_1', \mathtt{w}_2')$ (see definition of $\rightarrow_A \times \rightarrow_C$ in the paper). By this transition in $\rightarrow_A \times \rightarrow_C$ and the facts

$$g(\mathtt{w}') = g(\mathtt{w}_1 \cup \mathtt{w}_2) = g(\mathtt{w}_1) \cup \mathtt{w}_2$$
$$g(\mathtt{w}'') = g(\mathtt{w}_1' \cup \mathtt{w}_2') = g(\mathtt{w}_1') \cup \mathtt{w}_2'$$

there are two possible cases:

(i) $\mathtt{w}_1 \rightarrow_A \mathtt{w}_1'$ and $\mathtt{w}_2 = \mathtt{w}_2'$
(ii) $\mathtt{w}_2 \rightarrow_C \mathtt{w}_2'$ and $\mathtt{w}_1 = \mathtt{w}_1'$

In case, (i), $\mathtt{w}_1 \to_A \mathtt{w}_1'$ implies $f(\mathcal{W}_A) \to_B f(\mathcal{W}_A')$, hence:

$$\begin{aligned} \text{by definition of } \times\text{:} &\quad (f(\mathtt{w}_1), \mathtt{w}_2) \to_B \times \to_C (f(\mathtt{w}_1'), \mathtt{w}_2) \\ \text{by definition of } [\cdot]^D\text{:} &\quad f(\mathtt{w}_1) \cup \mathtt{w}_2 \to_{B|C} f(\mathtt{w}_1') \cup \mathtt{w}_2 \\ \text{by definition of } g\text{:} &\quad g(\mathtt{w}_1 \cup \mathtt{w}_2) \to_{B|C} g(\mathtt{w}_1' \cup \mathtt{w}_2)\ . \end{aligned}$$

Case (ii) is straightforward. □

Appendix C. Correctness of the lock-analysis technique

For readability sake we restate the theorem.

Theorem 6.4. Let (CT, $\mathtt{e}$, CCT) be an `FJf` program, $\text{ACT}_{[n]}$ be its abstract class table at n, and S be a state of its operational semantics.

1. If S has an object-circularity then at least one state of the lafsas $\llbracket \mathsf{S} \rrbracket_{[n]}$ has an object-circularity;
2. if $\mathsf{S} \xrightarrow{a} \mathsf{S}'$ and a state of $\llbracket \mathsf{S}' \rrbracket_{[n]}$ contains an object-circularity then an object-circularity is already present in a state of $\llbracket \mathsf{S} \rrbracket_{[n]}$.

Proof. To demonstrate the item 1., let $\llbracket \mathsf{S} \rrbracket_{[n]} = \langle \mathcal{W}, \mathcal{W}' \rangle$. We prove that every object name dependencies occurring in S is also contained in the initial state of $\mathcal{W}$. By Definition 6.2, if S has an object name dependency (a, a') then there is $\{\mathtt{t} :_a^\top \mathsf{E}[\mathtt{t}'.\mathtt{get}], \mathtt{t}' :_{a'} \mathtt{e}\} \in \mathsf{S}$, where $\mathtt{e}$ is not a value. By the typing rules, the contract of $\mathtt{t} :_a^\top \mathsf{E}[\mathtt{t}'.\mathtt{get}]$ is $(a, a') \fatsemi \mathbb{c}_\mathsf{E}$, where $\mathbb{c}_\mathsf{E}$ is the contract of E. As a consequence $\llbracket \mathtt{t} :_a^\top \mathsf{E}[\mathtt{t}'.\mathtt{get}] \rrbracket_{[n]} = \langle \{(a, a')\} \curvearrowright \mathcal{W}_\mathsf{E}, \mathcal{W}_\mathsf{E}' \rangle$, where $\llbracket \mathtt{t} :_a^\top \mathsf{E}[\,] \rrbracket_{[n]} = \langle \mathcal{W}_\mathsf{E}, \mathcal{W}_\mathsf{E}' \rangle$. Let $\llbracket \mathtt{t}' :_{a'} \mathtt{e} \rrbracket_{[n]} = \langle \mathcal{W}_\mathtt{e}, \mathcal{W}_\mathtt{e}' \rangle$, then $\llbracket \mathtt{t} :_a^\top \mathsf{E}[\mathtt{t}'.\mathtt{get}] \rrbracket_{[n]} | \llbracket \mathtt{t}' :_{a'} \mathtt{e} \rrbracket_{[n]} = \langle (\{(a, a')\} \curvearrowright \mathcal{W}_\mathsf{E}) | \mathcal{W}_\mathtt{e}, \mathcal{W}_\mathsf{E}' | \mathcal{W}_\mathtt{e}' \rangle$.

A similar argument concern object name dependencies of the form $(a^\mathsf{a}, a'^\mathsf{a})$ that correspond to pair of tasks $\{\mathtt{t}'' :_{a''}^\top \mathsf{E}'[\mathtt{t}'''.\mathtt{await}], \mathtt{t}''' :_{a'''} \mathtt{e}'\}$. In this case, the *lafsa*s are $\langle (\{(a^\mathsf{a}, a'^\mathsf{a})\} \curvearrowright \mathcal{W}_{\mathsf{E}'}) | \mathcal{W}_{\mathtt{e}'}, \mathcal{W}_{\mathsf{E}'}' | \mathcal{W}_{\mathtt{e}'}' \rangle$.

In general, if k object name dependencies occur in a state S, then there is $\mathsf{S}' \subseteq \mathsf{S}$ that collects all the tasks manifesting the object name dependencies such that

$$\begin{aligned} \llbracket \mathsf{S}' \rrbracket_{[\mathsf{n}]} &= \langle (\{(a_1^{[\mathsf{a}]}, b_1^{[\mathsf{a}]})\} \curvearrowright \mathcal{W}_{\mathsf{E}_1}) | \mathcal{W}_{\mathtt{e}_1}, \mathcal{W}_{\mathsf{E}_1}' | \mathcal{W}_{\mathtt{e}_1}' \rangle | \cdots | \langle (\{(a_k^{[\mathsf{a}]}, b_k^{[\mathsf{a}]})\} \curvearrowright \mathcal{W}_{\mathsf{E}_k}) | \mathcal{W}_{\mathtt{e}_k}, \mathcal{W}_{\mathsf{E}_k}' | \mathcal{W}_{\mathtt{e}_k}' \rangle \\ &= \langle (\{(a_1^{[\mathsf{a}]}, b_1^{[\mathsf{a}]})\} \curvearrowright \mathcal{W}_{\mathsf{E}_1}) | \mathcal{W}_{\mathtt{e}_1} | \cdots | (\{(a_k^{[\mathsf{a}]}, b_k^{[\mathsf{a}]})\} \curvearrowright \mathcal{W}_{\mathsf{E}_k}) | \mathcal{W}_{\mathtt{e}_k}, \mathcal{W}_{\mathsf{E}_1}' | \mathcal{W}_{\mathtt{e}_1}' | \cdots | \mathcal{W}_{\mathsf{E}_k}' | \mathcal{W}_{\mathtt{e}_k}' \rangle \end{aligned}$$

By definition of parallel composition in Section 5, the initial state of $\mathcal{W}$ will contain all the above pairs $(a_i^{[\mathsf{a}]}, b_i^{[\mathsf{a}]})$. We notice that the object name dependencies in $\llbracket \mathsf{S} \rrbracket_{[\mathsf{n}]}$ may be more than those occurring in S, since tasks $\{\mathtt{t} :_a^\top \mathsf{E}[\mathtt{t}'.\mathtt{get}], \mathtt{t}' :_{a'} \mathtt{v}\}$ do not represent an object dependency in a state, while they do produce a pair (a, a') in the corresponding automata.

Let us prove the item 2. We show that the transition $\mathsf{S} \xrightarrow{a} \mathsf{S}'$ does not produce new object name dependencies. That is, the set of object name dependencies in the states of $\llbracket \mathsf{S}' \rrbracket_{[n]}$ is equal or smaller than the set of dependencies in the states of $\llbracket \mathsf{S} \rrbracket_{[\mathsf{n}]}$.

Since $\Gamma \vdash \mathsf{S}$, for some Γ, then, for every $\mathtt{t} :_a \mathtt{e} \in \mathsf{S}$ we have $\Gamma \vdash \mathtt{t} :_a \mathtt{e} : (\mathtt{T}, \mathbb{r})$, $\mathbb{c}$ for some $\mathtt{T}$, $\mathbb{r}$, and $\mathbb{c}$. Therefore, by Theorem 4.4, if $\mathtt{t} :_a \mathtt{e}' \in \mathsf{S}'$ and $\mathtt{e} \neq \mathtt{e}'$ then $\Gamma \vdash \mathtt{t} :_a \mathtt{e}' : (\mathtt{T}', \mathbb{r})$, $\mathbb{c}'$, with $\mathtt{T}'$ `<:` $\mathtt{T}$ and $\mathbb{c}' \trianglelefteq \mathbb{c}$. The argument is by induction on the proof tree of $\mathbb{c}' \trianglelefteq \mathbb{c}$. By Definition A.3 we distinguish five cases:

1) $\mathbb{c} = \mathbb{c}'$, then $\llbracket \mathtt{t} :_a \mathtt{e} \rrbracket_{[n]} = \llbracket \mathtt{t} :_a \mathtt{e}' \rrbracket_{[n]}$ and $\llbracket \mathsf{S} \rrbracket_{[\mathsf{n}]} = \llbracket \mathsf{S}' \rrbracket_{[\mathsf{n}]}$.

2) $\mathbb{c}' = \mathtt{0}$, then $\mathtt{t} :_a \mathtt{e} \in \mathsf{S}$, where $\mathtt{e}$ is not a value, and $\mathtt{t} :_a \mathtt{v} \in \mathsf{S}'$. This happens when the reduction rule applied is (Invk) (without an immediate `get` or `await` on the invocation) or (Get) or (AwaitT). In the first subcase we refer to the treatment of new task creation below. Otherwise, if the applied rule is (Get) or (AwaitT), then $[\![\mathsf{S}]\!]_{[\mathsf{n}]}$ contain an object dependency that does not appear in $[\![\mathsf{S}']\!]_{[\mathsf{n}]}$, as stated.
3) $\mathbb{c} = \mathtt{C.m}\ \mathbb{r}(\bar{\mathbb{s}}) \to \mathbb{s}\,\boldsymbol{.}\,(a, a')^{[\mathsf{a}]}$ and $\mathbb{c}' = (a, a')^{[\mathsf{a}]}$, then $[\![\mathtt{t} :_a \mathtt{e}]\!]_{[n]} = \langle \mathcal{W}_{\mathtt{C,m}} \oplus (a, a')\mathbf{,}\ \mathcal{W}'_{\mathtt{C,m}} \rangle$ and $[\![\mathtt{t} :_a \mathtt{e}']\!]_{[n]} = \langle \{(a, a')\}\mathbf{,}\ \mathtt{0} \rangle$. This means that the result is true with respect to $\mathtt{t}$. However when $\mathbb{c}$ and $\mathbb{c}'$ are as in this case, then a new task has been spawned in the reduction step. We refer to treatment of new task creation below.
4) $\mathbb{c} = \mathtt{C.m} : \mathbb{r}(\bar{\mathbb{r}}) \to \mathbb{s}$, $\mathbb{c}' = \mathtt{D.m} : \mathbb{r}(\bar{\mathbb{r}}) \to \mathbb{s}$ and $\mathtt{D} \mathrel{\mathtt{<:}} \mathtt{C}$, then by consistency of cct (see Definition 4.2), $\mathbb{c} =^{\alpha} \mathbb{c}'$, hence $[\![\mathsf{S}]\!]_{[\mathsf{n}]} = [\![\mathsf{S}']\!]_{[\mathsf{n}]}$.
5) $\mathbb{c} = \mathbb{c}_1 \fatsemi \mathbb{c}''$ and $\mathbb{c}' = \mathbb{c}_2 \fatsemi \mathbb{c}''$, where $\mathbb{c}_1 \trianglelefteq \mathbb{c}_2$, then we can proceed by inductive hypothesis.

Cases 2) and 3) above need to further analysis because they may apply in the same rule when a new thread is spawned, therefore they have to be considered in conjunction with the creation of new tasks. The reduction step that introduces new tasks is:

$$\textsc{(Invk)}\quad \frac{\mathit{mbody}(\mathtt{m}, \mathit{class}(b)) = \bar{\mathtt{x}}.\mathtt{e} \qquad \mathtt{t}' = \mathit{freshtask}(\ )}{\mathtt{t} :_a^{\top} \mathsf{E}[b[\bar{\mathtt{f}} : \bar{\mathtt{v}}]!\mathtt{m}(\bar{\mathtt{v}}')] \xrightarrow{a} \mathtt{t} :_a^{\top} \mathsf{E}[\mathtt{t}'],\ \ \mathtt{t}' :_b^{\perp} \mathtt{e}[{}^{b[\bar{\mathtt{f}} : \bar{\mathtt{v}}]}/_{\mathtt{this}}][{}^{\bar{\mathtt{v}}'}/_{\bar{\mathtt{x}}}]}$$

We proceed by case analysis on the structure of E, assuming $\mathit{class}(b)$=$\mathtt{C}$:

Case $\mathsf{E} = \mathsf{E}'[[\,].\mathtt{get}]$ or $\mathsf{E} = \mathsf{E}'[[\,].\mathtt{await}]$. The type system associates to $\mathtt{t} :_a^{\top} \mathsf{E}[b[\bar{\mathtt{f}} : \bar{\mathtt{v}}]!\mathtt{m}(\bar{\mathtt{v}}')]$ a contract $\mathtt{C.m}\ b[\bar{\mathtt{f}} : \bar{X}](\overline{a_{\bar{\mathtt{v}}'}[\bar{\mathtt{f}}' : \bar{Y}]})\,\boldsymbol{.}\,(a_{\mathtt{this}}, b)^{[\mathsf{a}]} \fatsemi \mathbb{c}_{\mathsf{E}'}$. Let $\mathbb{c}_{\mathtt{C,m}}$ be the contract of the body of $\mathtt{C.m}$ and $\langle \mathcal{W}_{\mathtt{C,m}}\mathbf{,}\ \mathcal{W}'_{\mathtt{C,m}} \rangle$ be the corresponding pair of *lafsa*s. Then the *lafsa transformation* will produce a pair of *lafsa*s of the form $\langle \mathcal{W}_{\mathtt{C,m}} \oplus (a^{[\mathsf{a}]}_{\mathtt{this}}, b^{[\mathsf{a}]})\mathbf{,}\ \mathcal{W}'_{\mathtt{C,m}} \rangle$. The abstract semantics of task $\mathtt{t} \in \mathsf{S}$ will be then given by $\langle \mathcal{W}_{\mathtt{C,m}} \oplus (a^{[\mathsf{a}]}_{\mathtt{this}}, b^{[\mathsf{a}]})\mathbf{,}\ \mathcal{W}'_{\mathtt{C,m}} \rangle \fatsemi \langle \mathcal{W}_{\mathsf{E}'}\mathbf{,}\ \mathcal{W}'_{\mathsf{E}'} \rangle$, that is $\langle \mathcal{W}_{\mathtt{C,m}} \oplus (a^{[\mathsf{a}]}_{\mathtt{this}}, b^{[\mathsf{a}]}) \curvearrowright (\mathcal{W}'_{\mathtt{C,m}} | \mathcal{W}_{\mathsf{E}'})\mathbf{,}\ (\mathcal{W}'_{\mathtt{C,m}} | \mathcal{W}'_{\mathsf{E}'}) \rangle$.
Regarding S', the type system associates to $\mathtt{t} :_a^{\top} \mathsf{E}[\mathtt{t}']$ the contract $(a_{\mathtt{this}}, b)^{[\mathsf{a}]} \fatsemi \mathbb{c}_{\mathsf{E}'}$ corresponding to the pair of *lafsa*s $\langle \{(a^{[\mathsf{a}]}_{\mathtt{this}}, b^{[\mathsf{a}]})\} \curvearrowright \mathcal{W}_{\mathsf{E}'}\mathbf{,}\ \mathcal{W}'_{\mathsf{E}'} \rangle$ and to $\mathtt{t}' :_b^{\perp} \mathtt{e}[{}^{b[\bar{\mathtt{f}} : \bar{\mathtt{v}}]}/_{\mathtt{this}}]$ the pair of *lafsa*s $\langle \mathcal{W}_{\mathtt{C,m}}\mathbf{,}\ \mathcal{W}'_{\mathtt{C,m}} \rangle$. Therefore, $[\![\mathtt{t} :_a^{\top} \mathsf{E}[\mathtt{t}']]\!]_{[n]} \mid [\![\mathtt{t}' :_b^{\perp} \mathtt{e}[{}^{b[\bar{\mathtt{f}} : \bar{\mathtt{v}}]}/_{\mathtt{this}}]]\!]_{[n]} = \langle (\{(a^{[\mathsf{a}]}_{\mathtt{this}}, b^{[\mathsf{a}]})\} \curvearrowright \mathcal{W}_{\mathsf{E}'}) \mid \mathcal{W}_{\mathtt{C,m}}\mathbf{,}\ \mathcal{W}'_{\mathsf{E}'} \mid \mathcal{W}'_{\mathtt{C,m}} \rangle = \langle (\mathcal{W}_{\mathtt{C,m}} \mid \{(a^{[\mathsf{a}]}_{\mathtt{this}}, b^{[\mathsf{a}]})\}) \curvearrowright (\mathcal{W}_{\mathtt{C,m}} | \mathcal{W}_{\mathsf{E}'})\mathbf{,}\ \mathcal{W}'_{\mathsf{E}'} \mid \mathcal{W}'_{\mathtt{C,m}} \rangle$. It is easy to check that $\mathcal{W}_{\mathtt{C,m}} \oplus (a^{[\mathsf{a}]}_{\mathtt{this}}, b^{[\mathsf{a}]}) = \mathcal{W}_{\mathtt{C,m}} \mid \{(a^{[\mathsf{a}]}_{\mathtt{this}}, b^{[\mathsf{a}]})\}$. Hence, since no other task has been modified by the reduction, the resulting pair of *lafsa*s for S' is the same as the pair of *lafsa* for S.

Remaining cases. The runtime type system associates to $\mathtt{t} :_a^{\top} \mathsf{E}[b[\bar{\mathtt{f}} : \bar{\mathtt{v}}]!\mathtt{m}(\bar{\mathtt{v}}')]$ a contract $\mathtt{C.m}\ b[\bar{\mathtt{f}} : \bar{X}](\overline{a_{\bar{\mathtt{v}}'}[\bar{\mathtt{f}}' : \bar{Y}]}) \fatsemi \mathbb{c}_{\mathsf{E}}$.
Let $\mathbb{c}_{\mathtt{C,m}}$ be the contract of the body of $\mathtt{C.m}$ and $\langle \mathcal{W}_{\mathtt{C,m}}\mathbf{,}\ \mathcal{W}'_{\mathtt{C,m}} \rangle$ be the corresponding pair of *lafsa*s. Then the *lafsa transformation* produces a pair of *lafsa*s of the form $\langle \mathtt{0}\mathbf{,}\ \mathcal{W}_{\mathtt{C,m}} \curvearrowright \mathcal{W}'_{\mathtt{C,m}} \rangle$ and the abstract semantics of $\mathtt{t} \in \mathsf{S}$ will be then given by $\langle \mathtt{0}\mathbf{,}\ \mathcal{W}_{\mathtt{C,m}} \curvearrowright \mathcal{W}'_{\mathtt{C,m}} \rangle \fatsemi \langle \mathcal{W}_{\mathsf{E}}\mathbf{,}\ \mathcal{W}'_{\mathsf{E}} \rangle$. That is $\langle (\mathcal{W}_{\mathtt{C,m}} \curvearrowright \mathcal{W}'_{\mathtt{C,m}}) | \mathcal{W}_{\mathsf{E}}\mathbf{,}\ (\mathcal{W}_{\mathtt{C,m}} \curvearrowright \mathcal{W}'_{\mathtt{C,m}}) | \mathcal{W}'_{\mathsf{E}} \rangle$.
Regarding S', the type system associates to $\mathtt{t} :_a^{\top} \mathsf{E}[\mathtt{t}']$ a contract $\mathbb{c}_{\mathsf{E}}$ corresponding to the pair of *lafsa* $\langle \mathcal{W}_{\mathsf{E}}\mathbf{,}\ \mathcal{W}'_{\mathsf{E}} \rangle$ and to $\mathtt{t}' :_b^{\perp} \mathtt{e}[{}^{b[\bar{\mathtt{f}} : \bar{\mathtt{v}}]}/_{\mathtt{this}}]$ a contract $\langle \mathcal{W}_{\mathtt{C,m}}\mathbf{,}\ \mathcal{W}'_{\mathtt{C,m}} \rangle$. Hence, $[\![\mathtt{t} :_a^{\top} \mathsf{E}[\mathtt{t}']]\!]_{[n]} \mid [\![\mathtt{t}' :_b^{\perp} \mathtt{e}[{}^{b[\bar{\mathtt{f}} : \bar{\mathtt{v}}]}/_{\mathtt{this}}]]\!]_{[n]} = \langle \mathcal{W}_{\mathsf{E}} \mid \mathcal{W}_{\mathtt{C,m}}\mathbf{,}\ \mathcal{W}'_{\mathsf{E}} \mid \mathcal{W}'_{\mathtt{C,m}} \rangle$, which has a subset of the states of $[\![\mathsf{S}]\!]_{[\mathsf{n}]}$.