Actors may synchronize, safely! *

Elena Giachino, Ludovic Henrio, Cosimo Laneve, Vincenzo Mastandrea

Actors may synchronize, safely! *

Elena Giachino
University of Bologna,Italy & INRIA,France
elena.giachino@unibo.it

Ludovic Henrio
Université Côte d'Azur,CNRS,I3S,France
ludovic.henrio@cnrs.fr

Cosimo Laneve
University of Bologna, Italy & INRIA,France
cosimo.laneve@unibo.it

Vincenzo Mastandrea
Université Côte d'Azur, CNRS, I3S, France
University of Bologna, Italy & INRIA,France
mastandr@i3s.unice.fr

Abstract

We study deadlock detection in an actor model with *wait-by-necessity synchronizations*, a lightweight technique that synchronizes invocations when the corresponding values are strictly needed. This approach relies on the use of futures that are not given an explicit "Future" type. The approach we adopt allow the implicit synchronization on the availability of some value (where the producer of the value might be decided at runtime), whereas previous work allowed only explicit synchronization on the termination of a well-identified request. This way we are able to analyse the data-flow synchronization inherent to languages that feature wait-by-necessity.

We provide a type-system and a solver inferring the type of a program so that deadlocks can be identified statically. As a consequence we can automatically verify the absence of deadlocks in actor programs with wait-by-necessity synchronizations.

Categories and Subject Descriptors D.3.1 [*Programming languages*]: Formal Definitions and Theory; F.1.1 [*Computation by abstract devices*]: Models of Computation—Relations between models; F.1.2 [*Computation by abstract devices*]: Models of Computation—Parallelism and concurrency; F.3.2 [*Logics and meanings of programs*]: Semantics of Programming Languages—Operational semantics,Program analysis ; F.3.3 [*Logics and meanings of programs*]: Studies of Program Constructs—Type structure

Keywords Deadlock detection, type system, behavioral types

1. Introduction

Actors are a powerful computational model for defining distributed and concurrent systems [1, 2, 18]. This model has recently gained prominence, largely thanks to the success of the programming languages Erlang [3] and Scala [16]. The actor model relies on few key principles: (*a*) an actor encapsulates a number of data, by granting access only to the methods inside the actor itself; (*b*) method invocations are *asynchronous*, actors retain a queue for storing the invocations to their methods, which are processed sequentially by executing the corresponding instances of method bodies. The success of this programming model originates at the same time from its simplicity, from its properties, and from its abstraction level. Indeed, programming a concurrent system as a set of independent entities that only communicate through asynchronous messages eases the reasoning on the system and removes data-race conditions inherent to multi-threaded programming (in general, actors run a single applicative thread).

Problem: Actors and synchronizations. Actors do not explicitly support synchronization. Therefore, whenever a computation depends on the result of a message, the programmer must specify a callback mechanism where the invoker sends its identity and the invoked actor sends a result message to the invoker. However callbacks introduce an inversion of control that makes the reasoning on the program difficult (since results of invocations are received at a later stage, when the actor might be in a different state, it is harder to assess program correctness, for example). Providing synchronization as first-class linguistic primitive is generally preferable.

Some languages extend the actor model and provide synchronizations by allowing methods to return values: they are not anymore procedures. In general, this is realised by using *explicit futures*. A method of an actor returns a special kind of objects called *future* and some values are tagged with a *future type*. A special operation on a future allows the programmer to check whether the method has finished and at the same time retrieves the method result. The drawback of this approach is that programmers must know how to deal with them, and may be tempted to add too many synchronization points to simplify the reasoning on the program.

In this paper, we study a different extension of the actor model that uses *implicit futures* and a *wait-by-necessity* mechanism: the caller synchronizes with a method invocation only when its returned value is *strictly* necessary [7, 22]. This mechanism does not require explicit synchronization operators and ad-hoc types: the scheduler stops the flow of execution when a value to be returned by a method is needed for computing an expression. The synchronization becomes data-flow oriented: if some data is accessed and this data is not yet available, the program is automatically blocked. This way, an actor can return a result containing a future without

* This work was partially funded by the ANR project # ANR-11-LABX-0031-01 and by the EU project FP7-610582 ENVISAGE – Engineering Virtualized Services.

PPDP '16, September 05-07, 2016, Edinburgh, United Kingdom

ACM 978-1-4503-4148-6/16/09…$15.00
DOI: http://dx.doi.org/10.1145/http://dx.doi.org/10.1145/2967973.2968599

worrying about which actor will be responsible for synchronizing with the result: the synchronization will always occur as late as possible. Replacing a future by its value is no more an operation that has to be explicitly written by the programmer, as it automatically happens at some point of the computation that can be optimized by the designer of the language at runtime. A simple actor calculus with wait-by-necessity synchronizations, called gASP, is defined in Section 2.

While synchronization is useful, if it is used improperly it can cause *deadlocks* (deadlocks cannot occur in the basic actor model [10]). For example, two actors can both synchronize on an invocation to be evaluated by the other one, indefinitely waiting for the applicative thread of the other actor to be freed. Deadlock detection is a sensible issue, in particular because it is hard to verify in languages that admit systems with unbounded (mutual) recursion and dynamic actor creation. The following example illustrates the expressiveness of (implicit) futures and the difficulties of deadlock analysis:

Listing 1: Factorial method

```
1  Int fact(Int n, Int r){
2  Act x; Int y;
3  if (n == 0) return r;
4  else {  x = new Act(); r = r*n; n = n-1;
5  y = x.fact(n,r); return y;  }}
```

The access to `fact(n,1)` boils down to exactly `n` synchronizations. Indeed, since the value of `y` is never accessed within the method, the future is returned to the caller. When accessing the value of `fact(n,1)` a synchronization is performed on the result of the first nested invocation `fact(n-1,n)` which will need to access the result of the next invocation `fact(n-1,n*n-1)`, and so on. Technically, let the type of an asynchronous invocation be called *future type*. Then the type of `fact(n,r)` is a *recursive future type*[1]. Because of this type, it is not possible to determine at compile time how many explicit synchronizations happen when the value of `fact(x,1)` is needed, with `x` unknown.

A technique for deadlock analysis. To address (static-time) deadlock detection of gASP programs, we rely on a technique that has been already used for pi-calculus [14, 21] and for a concurrent object-oriented calculus called (core) ABS [13, 15]. Our technique consists of two modules:

module 1: a front-end type (inference) system that automatically extracts abstract behavioral descriptions relevant to deadlock analysis from gASP programs, called *behavioral types*. This part is developed in Section 3.

module 2: a back-end analyzer of types that computes a model of dependencies between runtime entities using a fixpoint technique. This part is discussed in Section 4.

According to this technique, a synchronization between actors α and α' is modeled by a dependency pair (α, α'), which means that the termination of a process of α depends on the termination of a process of α'. Programs are denoted by finite models that are sets of relations on names. If a circular dependency $(\alpha_1, \alpha_2) \cdots (\alpha_{n-1}, \alpha_n)(\alpha_n, \alpha_1)$ is found in one of the relations, then the corresponding program may manifest a deadlock.

As gASP and ABS are similar languages, one might be tempted to reduce deadlock detection of gASP to the corresponding problem of ABS, instead of redeveloping a similar technique. However, the compilation of gASP into ABS is not exactly possible because ABS features explicit futures. Synchronization on explicit futures boils down to checking the end of a method execution and retrieving the returned object, the retrieved object can be a future itself. On the contrary, with wait-by-necessity, if a computation requires a not-yet available value then a synchronization occurs, until a proper value (i.e. not a future) is available. Retrieving this value might require to wait for the termination of several methods. Indeed, consider the factorial example, let β be the actor needing the value of `fact(n,1)`. This synchronization requires that β simultaneously synchronizes with all the actors computing the nested factorial invocations, say $\beta_1, \ldots, \beta_{n-1}$. A translation from gASP to ABS would require to know statically the number n of synchronisation to perform. From the analysis point of view, this means that we have to collect all the dependencies of the form $(\beta, \beta_1), (\beta, \beta_2), \ldots, (\beta, \beta_{n-1})$. In [13, 15], this collection was done step-by-step by generating a dependency pair for every explicit synchronization. For synchronization on implicit futures, we need to generate a sequence of dependence pair when a value is needed, and this sequence is not bound statically.

Main contribution. Addressing adequately implicit futures amounts to define a new type system in module 1 of the above program and adapt in a non-trivial way the analyzer of module 2. The challenge we address is the ability to extend the synchronization point so that an unbounded number of events can be awaited at the same time. Our solution first extends the behavioural type with *fresh future identifiers* and to introduce specific types that identify whether a future is synchronised or not. A method signature also declares the set of actors and futures it creates to handle the potential unbounded number of future and actor creations. Then, we exploit the relation that exists between the number of dependencies of a synchronization and the number of nested method invocations. Instead of associating dependencies to synchronization points, *we delegate the production of the dependencies to method invocations*, each contributing with its own dependency. The sequence of dependencies is unfolded during the analysis. To implement this, in module 1, methods types of gASP carry an additional formal parameter, called *handle*, which is instantiated by the actor requiring the synchronization when this happens. The evaluation of behavioural types in the analyzer (module 2) also carries an environment binding future names to their values (method invocations).

The correctness of our technique is argumented in Section 5. Section 7 presents related work and Section 8 concludes the paper.

2. The calculus gASP

The syntax and the semantics of gASP are defined in the following two subsections; the last subsection defines deadlocks and discusses a few examples.

2.1 Syntax

In gASP, types T may be integers `Int` or actors `Act`. We use x, y to range over variable names The notation $\overline{T\ x\ ;}$ denotes any finite sequence of *name declarations* $T\ x$, separated by commas.

A gASP program is a sequence of field declarations $\overline{T\ x}$ and method definitions $T\ \mathtt{m}(\overline{T\ x})\ \{\, \overline{T\ y\ ;}\ \ s\,\}$, plus a main body $\{\, \overline{T\ z\ ;}\ \ s'\,\}$. The syntax of statements s, expressions with side effects z, (pure) expressions e, and values v of gASP is defined by the following grammar:

s	::=	$\mathtt{skip} \mid x = z \mid \mathtt{return}\, v \mid \mathtt{if}\, e\, \{\, s\,\}\, \mathtt{else}\, \{\, s\,\} \mid s\ ;\ s$	statements
z	::=	$e \mid v.\mathtt{m}(\overline{v}) \mid \mathtt{new}\ \mathtt{Act}(\overline{v})$	expressions with side effects
e	::=	$v \mid e \oplus e$	expressions
v	::=	$x \mid \mathtt{null} \mid$ *integer-values*	values

A statement s may be either one of the standard operations of an imperative language.

An expression z may change the state of the system. In particular, it may be an *asynchronous* method call $v.m(\overline{v})$ where v is the

[1] Precisely, the type of `fact` is `rec` X. $\mathtt{Int} + \mathtt{Fut}\langle X\rangle$, where $\mathtt{Fut}\langle\cdot\rangle$ denotes a future type.

actor executing the invocation, and $\overline{v}$ are the arguments of the invocation. This invocation does not suspend caller's execution: when the value computed by the invocation is needed *to evaluate an expression* then the caller waits for it. The intended meaning of operations taking place on different actors is that they may execute in parallel, while operations in the same actor are sequential (even if in the following operational semantics the parallelism is not explicit). Expressions z also include $\texttt{new Act}(\overline{v})$ that creates a new actor whose fields contain the values of $\overline{v}$.

A (*pure*) expression e may be a value v namely the reserved identifier `null`, or an identifier, or an integer value, or an arithmetic or relational expression. We denote arithmetic and relational expression with $e \oplus e'$. We assume that `this` is a special actor identifier that is used to refer to its own actor.

2.2 Semantics

We use two infinite sets of names: *actor names*, ranged over by α, $\beta, \ldots$, and *future names*, ranged over by f, $f', \ldots$. Let *configurations*, noted cn, be the terms defined below:

cn	$::=$	$\epsilon \mid f(w) \mid f(\bot) \mid \alpha(a,p,\overline{q}) \mid cn\ cn$	configurations
w	$::=$	$\alpha \mid f \mid v$	values and names
p, q	$::=$	$\{\ell \mid s\}$	processes
a, ℓ	$::=$	$\overline{x} \mapsto \overline{w}$	memories

Namely, configurations are sets of elements – therefore we identify configurations that are equal up-to associativity and commutativity. The elements of a configuration are denoted by the juxtaposition *cn cn'*; the empty configuration is denoted ε. Configurations consist of two types of elements. The element $\alpha(a,p,\overline{q})$ denotes an actor whose name is α, whose fields are recorded in the memory a, and with running process p and set $\overline{q}$ of processes waiting to be scheduled (the term $\overline{q}$ denotes a set in this case). The element $f(\cdot)$ represents a *future*, i.e. a pointer to a (runtime) value, which may be an actual value or a value not yet computed (an *unresolved future*)

A name, either actor or future, is *fresh* in a configuration if it does not occur in the configuration. We use the notation α fresh (by keeping implicit the configuration) to indicate that α is fresh. The following auxiliary functions are used in the operational semantics:

- $dom(\ell)$ return the domain of ℓ.
- $fields(\texttt{Act})$ returns the sequence of fields of `Act`.
- $\ell[x \mapsto v]$ is the function such that $(\ell[x \mapsto v])(x) = v$ and $(\ell[x \mapsto v])(y) = \ell(y)$, when $y \neq x$. Similarly for a.
- the memory $a + \ell$ is defined as
$$(a+\ell)(x) = \begin{cases} \ell(x) & \text{if } x \in \mathrm{dom}(\ell) \\ a(x) & \text{if } x \in \mathrm{dom}(a) \setminus \mathrm{dom}(\ell) \end{cases}$$
We also let $(a+\ell)[x \mapsto w]$ be $a + \ell[x \mapsto w]$, if $x \in \mathrm{dom}(\ell)$, or be $a[x \mapsto w] + \ell$, if $x \in \mathrm{dom}(a) \setminus \mathrm{dom}(\ell)$.
- $[\![e]\!]_{a+\ell}$ returns the value of e by computing the expression and retrieving the value of the identifiers that is stored in $a + \ell$; $[\![\overline{e}]\!]_{a+\ell}$ returns the tuple of values of $\overline{e}$.
- $\mathrm{bind}(\alpha, m, \overline{w}, f)$, where the method m is defined by $T\,\texttt{m}(\overline{T\ x})\ \{\overline{T\ y\ ;}\ s\}$, returns the following process $\{[\texttt{destiny} \mapsto f] \mid s[^{\overline{w}}/_{\overline{x}}][^{\alpha}/_{\texttt{this}}]\}$. We observe that the special field `destiny` in the local memory ℓ records the name of the future corresponding to the method invocation.

The semantics of `gASP` is defined operationally by means of a transition relation between configurations. Figure 2 collects the rules and below we discuss only three of them. Rule UPDATE performs the update of a future when the corresponding value has been computed. The point here is that the new value may be also a future: in this case, the rule is performing a dereference and it will continue in dereferencing future till an actual value is found. Rule SERVE schedules a new process to be executed. In the actor model, the processes ready to be executed are organized into a queue and SERVE just picks the first one. In this paper we are sticking to a more liberal organization for ready processes – they are a *set* – because FIFOs seem too constraining in a distributed system where the dispatch of invocations is nondeterministic. Clearly, if our technique asserts that a program is deadlock free then it will also be deadlock free when a different policy will be applied to ready processes. Rule ASSIGN stores a value or a name into a local variable or a field (*cf.* definition of $a+\ell$). The relevant point here is the evaluation function $w = [\![e]\!]_{a+\ell}$ because it may require actors' synchronizations. In fact, according to the definition of $[\![e]\!]_{a+\ell}$ in Figure 1, if e contains arithmetic operations, then the arguments must be evaluated to integers. That is, if some argument is a future then the rule can be applied *after that future has been evaluated*. We observe that, in rules INVK and INVK-SELF, the evaluation of $[\![e]\!]_{a+\ell}$ must return an actor name. If this is not the case – it returns a future – then the two rules cannot be applied and the actor must wait for the evaluation of the future.

The initial configuration of a `gASP` program with main body $\{\overline{T\ x\ ;}\ s\}$ is

$$main(\varnothing, \{[\text{destiny} \mapsto f_{main}, \overline{x \mapsto \bot}] \mid s[^{main}/_{\texttt{this}}]\}, \varnothing)$$

where $main$ is a special actor and f_{main} is a future name. As usual, let $\rightarrow^*$ be the reflexive and transitive closure of $\rightarrow$.

For example, let

```
Act x; Int y; Int z; x=new Act(); y=x.fact(1,1); z=y+0;
```

be a main body invoking `fact` in Listing 1. The corresponding initial configuration is

$$main(\varnothing, \{[\texttt{destiny} \mapsto f_{main}] \mid \textit{body-of-}\texttt{main}\}, \varnothing)$$

After several reduction steps, involving rules NEW, ASSIGN, INVK, SERVE, the configuration is extended with an actor executing the body of `fact` and a future initially $\bot$:

$$main(\varnothing, \{[\texttt{destiny} \mapsto f_{main}, x \mapsto \alpha, y \mapsto f_1] \mid \texttt{z=y+0}\}, \varnothing)$$
$$\alpha(\varnothing, \{[\texttt{destiny} \mapsto f_1, n \mapsto 1, r \mapsto 1] \mid \textit{body-of-}\texttt{fact}\}, \varnothing) \quad f_1(\bot)$$

After other reduction steps, being `1==0` false, the actor α creates a new actor β, again executing the body of method `fact` now with $n \mapsto 0$ and a new future f_0 initially $\bot$. Then α terminated the evaluation and the future $f_1(\bot)$ becomes $f_1(f_0)$, i.e., a forward to the result of the nested call on β. By rule UPDATE, the value of y in the actor $main$ becomes f_0, namely $main$ is still waiting because f_0 is not an actual value. Since the computation of β terminates because $n \mapsto 0$, the term $f_0(\bot)$ becomes $f_0(1)$. At this stage, by rule UPDATE, the value of y in the actor $main$ becomes 1 and $main$ may compute the assignment `z=y+0`.

2.3 Deadlocks

A *deadlock-free* configuration cannot be stuck forever on a synchronization. We define it formally below. Let $f \in p^{\texttt{stat}}$ whenever $p = \{\ell \mid s\}$ and s is either (*i*) $x = e \oplus e'; s'$ and f occurs in $e \oplus e'$, or (*ii*) $x = v.m(\overline{v}); s'$ and $f = [\![v]\!]_{\ell+a}$. Let also $f = p^{\texttt{mem}}$ if $p = \{\ell \mid s\}$ and $\ell(\texttt{destiny}) = f$.

Definition 2.1. *A configuration cn is* deadlock-free *if the following condition holds: whenever cn $\rightarrow^*$ cn' and $\alpha(a,p,\overline{q}) \in$ cn' such that there exists $f \in p^{\texttt{stat}}$, then there exists cn'' such that cn' $\rightarrow$ cn''.*

Lemma 2.2. *Let cn be a configuration such that*

$$\alpha_0(a_0, p_0, \overline{q_0}), \cdots, \alpha_{n-1}(a_{n-1}, p_{n-1}, \overline{q_{n-1}}) \in cn$$
$$\textit{and} \quad p'_0 \in \{p_0\} \cup \overline{q_0},\ \cdots,\ p'_{n-1} \in \{p_{n-1}\} \cup \overline{q_{n-1}}$$

such that there are $f_0, \cdots, f_{n-1}$ with $\forall i \in 0..n-1$: $f_i \in p'^{\texttt{stat}}_i$ and $f_i = p'^{\texttt{mem}}_{i+1}$, where $+$ is computed modulo n. Then cn is not deadlock-free. We call such configuration a deadlock.

According to Lemma 2.2, a configuration is deadlocked when there is a sequence of actors, each waiting for the value to be stored in a future by a process of the next actor in the sequence; the last

$$\frac{w \text{ is not a variable}}{[\![w]\!]_\ell \;=\; w} \qquad \frac{w \in \mathrm{dom}(\ell) \quad w \text{ is a variable}}{[\![w]\!]_\ell \;=\; \ell(w)} \qquad \frac{[\![e]\!]_\ell = k \qquad [\![e']\!]_\ell = k' \quad k, k' \text{ integer values}}{[\![e \oplus e']\!]_\ell \;=\; k \oplus k'}$$

Figure 1: The evaluation function

$$\textsc{Context}\ \frac{cn \to cn'}{cn\ cn'' \to cn'\ cn''} \qquad \textsc{Serve}\ \alpha(a, \varnothing, \overline{q} \cup \{p\}) \to \alpha(a, p, \overline{q})$$

$$\textsc{Return}\ \frac{w = [\![v]\!]_{a+\ell} \qquad \ell(\texttt{destiny}) = f}{\alpha(a, \{\ell \mid \texttt{return}\ v;\}, \overline{q})\ f(\bot) \to \alpha(a, \varnothing, \overline{q})\ f(w)} \qquad \textsc{Update}\ \frac{(a+\ell)(x) = f \quad (a+\ell)[x \mapsto w] = a' + \ell'}{\alpha(a, \{\ell \mid s\}, \overline{q})\ f(w) \to \alpha(a', \{\ell' \mid s\}, \overline{q})\ f(w)}$$

$$\textsc{Assign}\ \frac{x \in \mathrm{dom}(a+\ell) \qquad w = [\![e]\!]_{a+\ell} \quad (a+\ell)[x \mapsto w] = a' + \ell'}{\alpha(a, \{\ell \mid x = e; s\}, \overline{q}) \to \alpha(a', \{\ell' \mid s\}, \overline{q})}$$

$$\textsc{New}\ \frac{\overline{w} = [\![\overline{v}]\!]_{a+\ell} \qquad \beta \text{ fresh} \qquad \overline{y} = \mathit{fields}(\texttt{Act})}{\alpha(a, \{\ell \mid x = \texttt{new Act}(\overline{v}); s\}, \overline{q}) \to \alpha(a, \{\ell \mid x = \beta; s\}, \overline{q})\ \beta([\overline{y} \mapsto \overline{w}], \varnothing, \varnothing)}$$

$$\textsc{Invk}\ \frac{[\![v]\!]_{a+\ell} = \beta \qquad [\![\overline{v}]\!]_{a+\ell} = \overline{w} \qquad \beta \neq \alpha \quad f \text{ fresh} \qquad \mathrm{bind}(\beta, m, \overline{w}, f) = p''}{\alpha(a, \{\ell \mid x = v.\mathtt{m}(\overline{v}); s\}, \overline{q})\ \beta(a', p', \overline{q'}) \to \alpha(a, \{\ell \mid x = f; s\}, \overline{q})\ \beta(a', p', \overline{q'} \cup \{p''\})\ f(\bot)}$$

$$\textsc{Invk-Self}\ \frac{[\![v]\!]_{a+\ell} = \alpha \qquad [\![\overline{v}]\!]_{a+\ell} = \overline{w} \quad f \text{ fresh} \qquad \mathrm{bind}(\alpha, m, \overline{w}, f) = p'}{\alpha(a, \{\ell \mid x = v.\mathtt{m}(\overline{v}); s\}, \overline{q}) \to \alpha(a, \{\ell \mid x = f; s\}, \overline{q} \cup \{p'\})\ f(\bot)}$$

$$\textsc{If-True}\ \frac{[\![e]\!]_{a+\ell} \neq 0}{\alpha(a, \{\ell \mid \texttt{if}\ e\ \{\ s_1\ \}\ \texttt{else}\ \{\ s_2\ \}\ ;\ s\}, \overline{q}) \to \alpha(a, \{\ell \mid s_1\ ;\ s\}, \overline{q})} \qquad \textsc{If-False}\ \frac{[\![e]\!]_{a+\ell} = 0}{\alpha(a, \{\ell \mid \texttt{if}\ e\ \{\ s_1\ \}\ \texttt{else}\ \{\ s_2\ \}\ ;\ s\}\}, \overline{q}) \to \alpha(a, \{\ell \mid s_2\ ;\ s\}\}, \overline{q})}$$

Figure 2: Semantics of gASP.

one waits for a process of the first one. The following examples should make the statement clearer.

1. (self deadlock)

   $(a, \{\ell[\mathtt{y} \mapsto f] \mid \mathtt{x= y+1;}s\}, \overline{q})$ and $\{\ell'[\mathtt{destiny} \mapsto f] \mid s'\} \in \overline{q}$.

   In this case the n of Definition 2.2 is 1, namely the actor is waiting for a value that has to be computed by a task in its own queue. A method that manifests such a deadlock is

   Listing 2: Factorial self-deadlock

```
1   Int fact_d(Int n){
2       Int y;
3       if (n == 0) return 1;
4       else { n = n-1; y = this.fact_d(n);
5              y = y*(n+1) ; return y; }}
```

2. (a two-actors deadlock)

   $\alpha(a, \{\ell[\mathtt{destiny} \mapsto f, \mathtt{y} \mapsto f'] \mid \mathtt{x = y +1\ ;}s\}, \overline{q})$
   $\beta(a', \{\ell'[\mathtt{destiny} \mapsto f', \mathtt{y} \mapsto f] \mid \mathtt{x = y +1\ ;}s'\}, \overline{q'})$

   A method that manifests such a deadlock is

   Listing 3: Factorial two-actors deadlock

```
1   Int fact_d2(Int n, Act x){
2       Int y;
3       if (n==0) return 1;
4       else { n = n-1; y = x.fact_d2(n, this);
5              y = y*(n+1) ; return y; }}
```

2.4 Restrictions

We focus here on a sublanguage of gASP differing from the full language in the following aspects: fields only contain synchronized integers, i.e., (i) neither futures (ii) nor actors; and (iii) all futures created in a method must be either returned or synchronized.

Regarding (i) and (ii), they allow us to avoid any analysis of the content of fields and keeping the types for actors simpler.

Regarding item (iii) , this enforces that, *once the future for a method has been synchronized, all the futures directly or indirectly created by that method are synchronized too*. Notice that if a future is returned by the current method, then it will be synchronized by whomever will synchronize on the current method result. This prevents from having computation running in parallel without any mean to synchronize on it.

These restriction simplifies the presentation of the analysis and allow us to focus on the original aspects related to transparent futures and to the type system itself. They are enforced by the type system of Section 3. We discuss how to relax these restrictions in Section 8.

3. Behavioral Type System

The deadlock detection technique we present uses abstract descriptions, called *behavioral types*, that are associated to programs by a type system. The purpose of the type system is to collect dependencies between actors and between futures and actors. At each point of the program, the behavioral type gather informations on local synchronizations and on actors potentially running in parallel. We perform such an analysis for each method body, gathering the behavioral information at each point of the program. An environment Γ gathers types of useful variables and is updated sequentially with the body, while the dependencies themselves are summed in a control-flow independent way.

3.1 Notations

A *behavioral type program* is a pair $\left(\mathcal{L}, \Theta \cdot \mathsf{L}\right)$, where $\mathcal{L}$ is a *finite set* of *method behaviors* $\mathtt{m}(\alpha, \overline{\mathtt{x}}, X) = (\nu\, \overline{\varphi})(\Theta_{\mathtt{m}} \cdot \mathsf{L}_{\mathtt{m}})$, with $\alpha, \overline{\mathtt{x}}, X$ being the *formal parameters* of $\mathtt{m}$, $\Theta_{\mathtt{m}}$ the *future environment* of $\mathtt{m}$, $\mathsf{L}_{\mathtt{m}}$ the *body* of $\mathtt{m}$, and Θ and L are the *main future environment* and the *main behavioral type*, respectively. A future environment Θ maps future names to future behaviors (without synchronization information) $\lambda X.\mathtt{m}(\alpha, \overline{\mathtt{x}}, X)$. In the method behavior, the formal parameter α corresponds to the identity of the object on which the method is called (the `this`), while X, called *handle*, is a place-holder for the actor that will synchronize with the method. In practice several actors can synchronise with the same future, but only one at a time; X will thus be instantiated by a single actor at each point of the analysis. $\overline{\mathtt{x}}$ are the types of the method parame-

ters. The binder $(\nu\overline{\varphi})$ binds the occurrences of $\overline{\varphi}$ in $\Theta_{\mathtt{m}}$ and $\mathsf{L}_{\mathtt{m}}$, with φ ranging over future or actor names.

The syntax of behavioral types L is defined in Figure 3. The basic types $\mathbb{r}$ are used for values: they may be either $\square$, to model integers, or any actor name α. The extended type $\mathbb{x}$ is the type of variables, and it may be a value type $\mathbb{r}$ or a *not-yet-synchronized type* $\mathbb{r}_f$ (in order to retrieve the value $\mathbb{r}$ it is necessary to synchronize the future f). The behavioral type $\mathsf{0}$ enforces no dependency, (κ, α) enforces the dependency between κ and α meaning that, if κ is instantiated by an actor β, then β will need α to be available in order to proceed its execution. The term f_κ may represent different behaviors depending on the value of κ: $f_\star$ represents an unsynchronized future f, which is a pointer in the future environment to the corresponding method invocation; f_α represents the synchronization of the actor α with the future f; f_X represents the return of a future f by the method associated to the handler X. The type $\mathsf{L} \,\&\, \mathsf{L}'$ is the *parallel composition* of L and of L': it is the behavior of two methods running in parallel and not necessarily synchronized. The sum $\mathsf{L} + \mathsf{L}'$ composes the dependencies of L and L' independently: it is the composition of two behaviors that cannot occur at the same time, either because one occurs before the other or because they are exclusive. The behavior of a future (stored in the environment) is tagged $\checkmark$ if the future is synchronized and $\rightarrow$ if the future is returned by the return statement; else it has no tag. Whenever parentheses are omitted, the operation "$\&$" has precedence over "$+$". We will shorten $\mathsf{L}_1 \,\&\, \cdots \,\&\, \mathsf{L}_n$ into $\&_{i\in\{1..n\}}\, \mathsf{L}_i$. In the syntax of L, the operations "$\&$" and "$+$" are associative, commutative with $\mathsf{0}$ being the identity on $\&$, and behavioral types are equal up-to alpha renaming of bound names.

The judgments of the type system have a typing context Γ mapping variables to extended types $\mathbb{x}$, future names to future behavior $\lambda X.\mathtt{m}(\alpha, \overline{\mathbb{x}}, X)^{[\checkmark]}$, and method names to their signatures of the form $(\alpha, \overline{\mathbb{x}}, X) \rightarrow \mathbb{r}$, where $\alpha, \overline{\mathbb{x}}, X$ are the formal parameters of the method behavior and $\mathbb{r}$ is the type of the returned value, respectively. Judgments have the following form:

- $\Gamma \vdash m : (\alpha, \overline{\mathbb{x}}) \rightarrow \mathbb{r}$ for instantiating the method signature of $\mathtt{m}$.
- $\Gamma \vdash v : \mathbb{x}$ for typing variables and values.
- $\Gamma \vdash_S z : \mathbb{x}, \mathsf{L} \triangleright \Gamma'$ for typing expressions with side effects, where S is the set of parameters of the current method body being typed, L is the behavioral type of the expression, and Γ' is the environment obtained by updating Γ to reflect possible future creations.
- $\Gamma \vdash_S s : \mathsf{L} \triangleright \Gamma'$ for typing statements, where S and L are as before, and Γ' is obtained by updating Γ to reflect possible variable assignment.

Since Γ is a function, we use the standard predicates $x \in \mathrm{dom}(\Gamma)$ or $x \notin \mathrm{dom}(\Gamma)$. We define some additional auxiliary function on Γ in Figure 4. $\Gamma(f)^{\rightarrow}$ is defined similarly to $\Gamma(f)^{\checkmark}$. $\mathtt{Fut}(\Gamma)$ collects all the futures stored in Γ, $\mathtt{AFut}(\Gamma)$ collects all the futures that are not tagged with a $\checkmark$ or $\rightarrow$, i.e. not-yet-synchronized futures, and $\mathit{unsync}(\Gamma)$ performs the parallel composition of the behavioral types of not-yet-synchronized method invocations, it collects the unsynchronized future of all the methods running in parallel.

3.2 Typing Rules

The typing rules are presented in Figure 5 and the most significant ones are discussed below. In general, a statement has a behavior that is a sum of behaviors. Each term of the sum is a parallel composition of synchronization dependencies and unsynchronized behaviors. We propagate this way the set of methods running in parallel as a set of not-yet-synchronized futures all along the type analysis (see the role of $\mathit{unsync}(\Gamma')$ in rules (T-SYNC), (T-INVK), (T-RETURN)). The statements that create no synchronization at all (i.e. that do not access a future, nor call a method, nor return from a method) have behavior 0 and an unsynchronized behavior that is the same as for the previous statement. We omit the unsynchronized part in this case as no deadlock can be created at those steps.

Rule (T-SYNC) types the synchronization of a not-yet-synchronized value v, namely when v has type $\mathbb{r}_f$. The rule dereferences the future f by assigning to v the type $\mathbb{r}$. The corresponding behavioral type is $f_\alpha \,\&\, \mathit{unsync}(\Gamma')$, where α is the synchronizing actor and the not-yet-synchronized method invocations are added in parallel. The environment Γ is updated in order to record the synchronization in the possible aliases of v and to record that f has been computed (the tag $\checkmark$). Notice that in case v is already associated to a value type, namely $\mathbb{r}$, no synchronization and no environment update is performed – see rule (T-SYNC-VAL). Rule (T-INVK) creates a new future and stores it in Γ. The receiving actor v must have a type α, meaning that the value cannot be an unresolved future, while the parameters $\overline{v}$ may have future types. The resulting behavioral type is the sum of behavioral type resulting from the possible synchronization on v and the not-yet-synchronized method invocations – i.e., the current one on f and those that are running in parallel, represented by $\mathit{unsync}(\Gamma')$. Rules (T-ASSIGN-FIELD), (T-ASSIGN-VAL), and (T-ASSIGN-EXP) type the assignment. The first one constrains fields to be assigned to (already synchronized) integers; the second one types the assignment of values to local variables, without performing any synchronization, thus supporting the aliasing of future variables; the third rule types the assignment of an expression $e \oplus e'$ to a local variable. In the last case, a synchronization might be required.

Rule (T-SEQ), given the types L_1 and L_2 of the two subsequent statements, correctly associates a new composite type to the sequential composition of the statements. In rule (T-IF) the behavioral type of a conditional statement is the sum of the behavioral type resulting from the typing of the expression e and the behavioral types of the two branches. The rule can be applied provided that the futures created in the branches are either returned (by a *return* statement) or synchronized (constraint on line 2). The environment is updated with the changes that occur in the two branches, when they are equal, and the futures created in one of the branches, when they are synchronized or returned $(\Gamma_1|_{\mathtt{Fut}(\Gamma_1)\setminus\mathtt{Fut}(\Gamma')} \cup \Gamma_2|_{\mathtt{Fut}(\Gamma_2)\setminus\mathtt{Fut}(\Gamma')})$. Variables that are modified in different ways by the two branches are restricted to be almost the same. The only allowed difference is if one future is synchronized in one of the branch and not in the other; in this case the merge retains the unsynchronized behavior.

In rule (T-RETURN) we check that the return type value corresponds to the value type of `destiny` unless it is a fresh name (not belonging to the formal parameters of the method). In Γ, `future` contains the handler variable X (see below). The corresponding future is tagged with a $\rightarrow$ in the resulting environment (because it is returned to the caller). In case v is already associated to a value type, namely $\mathbb{r}$, no synchronization and no environment update is performed – see rule (T-RETURN-VAL).

In rule (T-METHOD), the behavioral type of the method body is extended with a parallel pair (X, α) which will be instantiated by the actor performing a synchronization on this method. As mentioned earlier, the rule constrains method bodies to either return futures created in the body or to synchronize on them – predicate $\mathtt{AFut}(\Gamma') = \varnothing$. The environment $\Theta_{\mathtt{m}}$ is defined by comprehension, collecting all the futures created in the method (without synchronization information).

Example 3.1. Let us discuss the typing of the program containing the factorial method of Listing 1 in page 2 and the following main statement

```
Act x; Int y; Int z;
```

$$
\begin{array}{rcll}
\mathbb{r} & ::= & \square \mid \alpha & \text{basic type} \\
\mathbb{x} & ::= & \mathbb{r} \mid \mathbb{r}_f & \text{extended type} \\
\kappa & ::= & \star \mid \alpha \mid X & \text{synchronizers} \\
\mathsf{L} & ::= & \mathtt{0} \mid (\kappa, \alpha) \mid f_\kappa \mid \mathsf{L} + \mathsf{L} \mid \mathsf{L} \,\&\, \mathsf{L} & \text{behavioral type} \\
\lambda X.\mathtt{m}(\alpha, \overline{\mathbb{x}}, X)^{[\checkmark]} & ::= & \lambda X.\mathtt{m}(\alpha, \overline{\mathbb{x}}, X) \mid \lambda X.\mathtt{m}(\alpha, \overline{\mathbb{x}}, X)^{\checkmark} \mid \lambda X.\mathtt{m}(\alpha, \overline{\mathbb{x}}, X)^{\rightarrow} & \text{future behavior}
\end{array}
$$

Figure 3: Syntax of behavioral types.

$$
\mathtt{Merge}(\Gamma_1, \Gamma_2)(x) = \begin{cases} \mathbb{r}_f & \text{if } \Gamma_i(x) = \mathbb{r}_f \wedge \Gamma_j(x) = \mathbb{r}, \\ & i, j \in \{1, 2\} \\ \mathbb{x} & \text{if } \Gamma_1(x) = \Gamma_2(x) = \mathbb{x} \\ \mathit{undef}. & \text{otherwise} \end{cases}
\qquad
\Gamma[x \mapsto \mathbb{x}](y) = \begin{cases} \mathbb{x} & \text{if } y = x \\ \Gamma(y) & \text{otherwise} \end{cases}
\qquad
\Gamma|_S(x) = \begin{cases} \Gamma(x) & \text{if } x \in S \\ \mathit{undefined} & \text{otherwise} \end{cases}
$$

$$
(\Gamma \setminus x)(y) = \begin{cases} \Gamma(y) & \text{if } x \neq y \\ \mathit{undef}. & \text{if } x = y \end{cases}
\qquad
(\Gamma + \Gamma')(x) = \begin{cases} \Gamma(x) & \text{if } x \in \mathit{dom}(\Gamma) \\ \Gamma'(x) & \text{if } x \in \mathit{dom}(\Gamma') \end{cases}
$$

Note: $\Gamma + \Gamma'$ is defined only if $\forall\, x \in \mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma').\, \Gamma(x) = \Gamma'(x)$

$$
\Gamma[x \mapsto \mathbb{x}']^{\Gamma(x)=\mathbb{x}} \stackrel{def}{=} \Gamma[x_1 \mapsto \mathbb{x}] \cdots [x_n \mapsto \mathbb{x}] \text{ if } \{y \,|\, \Gamma(y) = \mathbb{x}\} = \{x_1, \cdots, x_n\}
$$

$$
\Gamma(f)^{\checkmark} = \begin{cases} \lambda X.\mathtt{m}(\alpha, \overline{\mathbb{x}}, X)^{\checkmark} & \text{if } \Gamma(f) = \lambda X.\mathtt{m}(\alpha, \overline{\mathbb{x}}, X)^{[\checkmark]} \\ \mathit{undef}. & \text{if } f \notin \mathrm{dom}(\Gamma) \end{cases}
\qquad
\Gamma(f)^{\times} = \begin{cases} \lambda X.\mathtt{m}(\alpha, \overline{\mathbb{x}}, X) & \text{if } \Gamma(f) = \lambda X.\mathtt{m}(\alpha, \overline{\mathbb{x}}, X)^{[\checkmark]} \\ \mathit{undef}. & \text{if } f \notin \mathrm{dom}(\Gamma) \end{cases}
$$

$$
\mathtt{Fut}(\Gamma) \stackrel{def}{=} \{f \mid f \in \mathrm{dom}(\Gamma)\}
\qquad
\mathtt{AFut}(\Gamma) \stackrel{def}{=} \{f \in \mathtt{Fut}(\Gamma) \mid \Gamma(f) = \Gamma(f)^{\times}\}
\qquad
\mathit{unsync}(\Gamma) \stackrel{def}{=} \&_{f \in \mathtt{AFut}(\Gamma)}\, f_\star,
$$

Figure 4: Auxiliary definitions

```
x = new Act(); y = x.fact(3,1); z = y + 0;
```

We begin with the typing of the main body.
Let $\Gamma = [\mathtt{fact} \mapsto (\alpha_{\mathtt{fact}}, \square_{f_{\mathtt{fact}}}, \square_{g_{\mathtt{fact}}}, X) \to \square]$. By the rule (T-PROGRAM) we need to verify $\Gamma + \mathtt{this} : \mathit{main} \vdash_{\varnothing}$ $\mathtt{x = newAct(); y = x.fact(3,1); z = y + 0;} \;:\; \mathsf{L} \;\triangleright\; \Gamma''$. By rules (T-SEQ), (T-ASSIGN-EXP), (T-NEW), (T-INVK), and (T-SYNCH) we get

$\Gamma + \mathtt{this} : \mathit{main} \vdash_{\varnothing} \mathtt{x=new\ Act()} : \mathtt{0} \triangleright \Gamma + \mathtt{this} : \mathit{main} + x : \alpha$,

$\Gamma + \mathtt{this} : \mathit{main} + x : \alpha \vdash_{\varnothing} \mathtt{y = x.fact(3,1);} : f''_\star \triangleright \Gamma'$, and

$\Gamma' \vdash_{\varnothing} \mathtt{z = y + 0;} : f''_{\mathit{main}} \triangleright \Gamma''$

where,

$\Gamma' = \Gamma + \mathtt{this} : \mathit{main} + x : \alpha + y : \square_{f''} + f' : \lambda X.\mathtt{fact}(\alpha, \square, \square, X)$,

and $\Gamma'' = \Gamma + \mathtt{this} : \mathit{main} + x : \alpha + y : \square_{f''} + z : \square$
$+ f'' : \lambda X.\mathtt{fact}(\alpha, \square, \square, X)^{\checkmark}$.

Thus, $\mathsf{L} = f''_\star + f''_{\mathit{main}}$ and $\Theta = \{f'' \mapsto \lambda X.\mathtt{fact}(\alpha, \square, \square, X)\}$, by rule (T-PROGRAM).
By rule (T-METHOD), we type the body of method fact with the following judgement:

$\Gamma + \mathtt{this} : \alpha + \mathtt{n} : \square_{f_{\mathtt{fact}}} + \mathtt{r} : \square_{g_{\mathtt{fact}}} + \mathtt{destiny} : \square + \mathtt{future} : X \vdash_{\{\alpha, \square_{f_{\mathtt{fact}}}, \square_{g_{\mathtt{fact}}}\}} s_{\mathtt{fact}} : \mathsf{L}_{\mathtt{fact}} \triangleright \Gamma'''$

Following a similar reasoning as above, we get

$$
\begin{array}{rcl}
\Theta_{\mathtt{fact}} & = & \{f' \mapsto \lambda X.\, \mathtt{fact}(\beta, \square, \square, X)\} \\
\mathsf{L}_{\mathtt{fact}} & = & (f_\alpha + g_X + g_\alpha + f'_\star + f'_X) \,\&\, (X, \alpha)
\end{array}
$$

and the behavioral type program becomes:

$$
\big(\mathtt{fact}(\alpha, \square_f, \square_g, X) = (\nu\, \beta, f')(\Theta_{\mathtt{fact}} \cdot \mathsf{L}_{\mathtt{fact}}), \Theta \cdot \mathsf{L}\big)
$$

The type $\mathsf{L}_{\mathtt{fact}}$ is a sum of five types (we omit 0 types): the synchronization of the n parameter, the return of the result in the if branch, the synchronization of the r parameter, the reference to a new future associated in $\Theta_{\mathtt{fact}}$ to the recursive invocation of fact on a new actor, and the last one is the return of the future of the method invocation on the else branch. This sum is in conjunction with the pair (X, α) which represents the possible synchronization made by a synchronizer X on the current method. The type L of the main statement contains the reference f'', which is linked in Θ to the method invocation of fact, followed by its synchronization.

Example 3.2. Let us now consider the program of Listing 2 in page 4. The associated behavioral type program is

$$
(\mathtt{fact\_d}(\alpha, \square_f, X) = (\nu\, f')(\Theta_{\mathtt{fd}} \cdot \mathsf{L}_{\mathtt{fd}}), \Theta \cdot \mathsf{L})
$$

where

$$
\begin{array}{rcl}
\Theta_{\mathtt{fd}} & = & \{f' \mapsto \lambda X.\, \mathtt{fact\_d}(\alpha, \square, X)\} \\
\mathsf{L}_{\mathtt{fd}} & = & (f_\alpha + f'_\star + f'_\alpha) \,\&\, (X, \alpha) \\
\Theta & = & \{f'' \mapsto \lambda X.\mathtt{fact\_d}(\beta, \square, X)\} \\
\mathsf{L} & = & f''_\star + f''_{\mathit{main}}
\end{array}
$$

In Section 4, we will see that the synchronization f'_α causes a deadlock, because the corresponding method invocation is performed on the actor α, which amounts to instantiate the pair (X, α) into (α, α).

Example 3.3. The behavioral type of the program in Listing 3 is

$$
(\mathtt{fact\_d2}(\alpha, \square_f, \beta_g, X) = (\nu\, f')(\Theta_{\mathtt{fd2}} \cdot \mathsf{L}_{\mathtt{fd2}}), \Theta \cdot \mathsf{L}))
$$

where

$$
\begin{array}{rcl}
\Theta_{\mathtt{fd2}} & = & \{f' \mapsto \lambda X.\mathtt{fact\_d2}(\beta, \square, \alpha, X)\} \\
\mathsf{L}_{\mathtt{fd2}} & = & (f_\alpha + g_\alpha + f'_\star + f'_\alpha) \,\&\, (X, \alpha) \\
\Theta & = & \{f'' \mapsto \lambda X.\mathtt{fact\_d2}(\alpha', \square, \beta', X)\} \\
\mathsf{L} & = & f''_\star
\end{array}
$$

3.3 Type Soundness

The correctness of the type system in Section 3 is demonstrated by means of a subject reduction theorem expressing that if a runtime configuration *cn* is well typed and $cn \rightarrow cn'$ then cn' is well typed as well. While the theorem is almost standard, in our case we cannot demonstrate a statement guaranteeing standard type-preservation (the equality of types of *cn* and *cn*′) because our types are behavioral. However, it is critical for the correctness of the analysis that there is a relation between the type of *cn*, let it be $\Theta \cdot \mathsf{L}$, and the type of cn', let it be $\Theta' \cdot \mathsf{L}'$. Therefore, a subject reduction for the type system of Section 3 requires the extension

expressions and addresses $\Gamma \vdash v : \mathbb{x}$

$$\text{(T-VAR)}\ \frac{\Gamma(x) = \mathbb{x}}{\Gamma \vdash x : \mathbb{x}} \qquad \text{(T-VAL)}\ \frac{v \quad \textit{integer-value} \text{ or } \texttt{null}}{\Gamma \vdash v : \square} \qquad \text{(T-METHOD-SIGN)}\ \frac{\Gamma(\mathtt{m}) = (\alpha, \overline{\mathbb{x}}, X) \to \mathbb{r} \quad \sigma \text{ renaming} \quad \sigma(\square) = \square \quad \mathbb{r} \notin \{\square, \alpha, \overline{\mathbb{x}}\} \implies \sigma(\mathbb{r}) \text{ fresh}}{\Gamma \vdash m : (\sigma(\alpha), \sigma(\overline{\mathbb{x}}), X) \to \sigma(\mathbb{r})}$$

expressions with side effects $\Gamma \vdash_S e : \mathbb{x}, \mathsf{L} \triangleright \Gamma'$

$$\text{(T-SYNC)}\ \frac{\Gamma \vdash v : \mathbb{r}_f \quad \Gamma(\texttt{this}) = \alpha \quad \Gamma' = (\Gamma[y \mapsto \mathbb{r}]^{\Gamma(y)=\mathbb{r}_f})[f \mapsto \Gamma(f)^{\checkmark}]}{\Gamma \vdash_S v : \mathbb{r},\ f_\alpha \,\&\, unsync(\Gamma') \triangleright \Gamma'} \qquad \text{(T-SYNC-VAL)}\ \frac{\Gamma \vdash v : \mathbb{r}}{\Gamma \vdash_S v : \mathbb{r},\ 0 \triangleright \Gamma} \qquad \text{(T-EXPRESSION)}\ \frac{\Gamma \vdash_S e : \square,\ \mathsf{L}_1 \triangleright \Gamma' \quad \Gamma' \vdash_S e' : \square,\ \mathsf{L}_2 \triangleright \Gamma''}{\Gamma \vdash_S e \oplus e' : \square,\ \mathsf{L}_1 + \mathsf{L}_2 \triangleright \Gamma''}$$

$$\text{(T-NEW)}\ \frac{\alpha \text{ fresh} \quad \Gamma \vdash \overline{v} : \overline{\square}}{\Gamma \vdash_S \texttt{new Act}(\overline{v}) : \alpha,\ 0 \triangleright \Gamma} \qquad \text{(T-INVK)}\ \frac{\Gamma \vdash_S v : \alpha,\ \mathsf{L} \triangleright \Gamma' \quad \Gamma' \vdash \overline{v} : \overline{\mathbb{x}} \quad \Gamma' \vdash \mathtt{m} : (\alpha, \overline{\mathbb{x}}, X) \to \mathbb{r} \quad f \text{ fresh} \quad \Gamma'' = \Gamma'[f \mapsto \lambda X.\mathtt{m}(\alpha, \overline{\mathbb{x}}, X)]}{\Gamma \vdash_S v.\mathtt{m}(\overline{v}) : \mathbb{r}_f,\ \mathsf{L} + f_\star \,\&\, unsync(\Gamma') \triangleright \Gamma''}$$

statements $\Gamma \vdash_S s : \mathsf{L} \triangleright \Gamma'$

$$\text{(T-ASSIGN-FIELD)}\ \frac{x \in \mathit{fields}(\texttt{Act}) \setminus \mathrm{dom}(\Gamma) \quad \Gamma \vdash v : \square}{\Gamma \vdash_S x = v : 0 \triangleright \Gamma} \qquad \text{(T-ASSIGN-VAL)}\ \frac{x \notin \mathit{fields}(\texttt{Act}) \setminus \mathrm{dom}(\Gamma) \quad \Gamma \vdash v : \mathbb{x}}{\Gamma \vdash_S x = v : 0 \triangleright \Gamma[x \mapsto \mathbb{x}]} \qquad \text{(T-ASSIGN-EXP)}\ \frac{x \notin \mathit{fields}(\texttt{Act}) \setminus \mathrm{dom}(\Gamma) \quad z \text{ is not a value } v \quad \Gamma \vdash_S z : \mathbb{x},\ \mathsf{L} \triangleright \Gamma'}{\Gamma \vdash_S x = z : \mathsf{L} \triangleright \Gamma'[x \mapsto \mathbb{x}]}$$

$$\text{(T-SEQ)}\ \frac{\Gamma \vdash_S s_1 : \mathsf{L}_1 \triangleright \Gamma_1 \quad \Gamma_1 \vdash_S s_2 : \mathsf{L}_2 \triangleright \Gamma_2}{\Gamma \vdash_S s_1; s_2 : \mathsf{L}_1 + \mathsf{L}_2 \triangleright \Gamma_2} \qquad \text{(T-RETURN)}\ \frac{\Gamma(\texttt{destiny}) = \mathbb{r} \quad \Gamma(\texttt{future}) = X \quad \Gamma \vdash v : \mathbb{r}'_{f'} \quad \mathbb{r}' \in S \vee \mathbb{r} \in S \implies \mathbb{r} = \mathbb{r}' \quad \Gamma' = \Gamma[f' \mapsto \Gamma(f')^{\rightarrow}]}{\Gamma \vdash_S \texttt{return}\ v : f'_X \,\&\, unsync(\Gamma') \triangleright \Gamma'} \qquad \text{(T-RETURN-VAL)}\ \frac{\Gamma(\texttt{destiny}) = \mathbb{r} \quad \Gamma \vdash v : \mathbb{r}' \quad \mathbb{r}' \in S \vee \mathbb{r} \in S \implies \mathbb{r} = \mathbb{r}'}{\Gamma \vdash_S \texttt{return}\ v : 0 \triangleright \Gamma}$$

$$\text{(T-SKIP)}\ \Gamma \vdash_S \texttt{skip} : 0 \triangleright \Gamma \qquad \text{(T-IF)}\ \frac{\Gamma \vdash_S e : \square,\ \mathsf{L} \triangleright \Gamma' \quad \Gamma' \vdash_S s_1 : \mathsf{L}_1 \triangleright \Gamma_1 \quad \Gamma' \vdash_S s_2 : \mathsf{L}_2 \triangleright \Gamma_2 \quad (\texttt{AFut}(\Gamma_1) \cup \texttt{AFut}(\Gamma_2)) \setminus \texttt{AFut}(\Gamma') = \varnothing \quad \Gamma'' = \texttt{Merge}(\Gamma_1, \Gamma_2) \cup \Gamma_1|_{\texttt{Fut}(\Gamma_1) \setminus \texttt{Fut}(\Gamma')} \cup \Gamma_2|_{\texttt{Fut}(\Gamma_2) \setminus \texttt{Fut}(\Gamma')}}{\Gamma \vdash_S \texttt{if}\ e\ \{\ s_1\ \}\ \texttt{else}\ \{\ s_2\ \} : \mathsf{L} + \mathsf{L}_1 + \mathsf{L}_2 \triangleright \Gamma''}$$

Figure 5: Typing rules for expressions, expressions with side-effects and statements.

$$\text{(T-METHOD)}\ \frac{\Gamma(\mathtt{m}) = (\alpha, \overline{\mathbb{x}}, X) \to \mathbb{r} \quad \Gamma + \texttt{this} : \alpha + \overline{x : \mathbb{x}} + \texttt{destiny} : \mathbb{r} + \texttt{future} : X \vdash_{\{\alpha, \overline{\mathbb{x}}\}} s : \mathsf{L} \triangleright \Gamma' \quad \texttt{AFut}(\Gamma') = \varnothing \quad \overline{\varphi} = var(\mathsf{L}) \setminus \{\alpha, \overline{\mathbb{x}}\} \quad \Theta_{\mathtt{m}} = [f \mapsto \Gamma'(f)^{\times}]^{f \in \texttt{Fut}(\Gamma')} \quad \mathsf{L}_{\mathtt{m}} = \mathsf{L}\,\&(X, \alpha)}{\Gamma \vdash \mathtt{m}\ (\overline{T\ x})\{\overline{T\ y};\ s\} : \mathtt{m}(\alpha, \overline{\mathbb{x}}, X) = (\nu\,\overline{\varphi})(\Theta_{\mathtt{m}} \cdot \mathsf{L}_{\mathtt{m}})}$$

$$\text{(T-PROGRAM)}\ \frac{\Gamma \vdash \overline{M} : \mathcal{L} \quad \Gamma + \texttt{this} : main \vdash_{\varnothing} s : \mathsf{L} \triangleright \Gamma' \quad \Theta = [f \mapsto \Gamma'(f)^{\times}]^{f \in \texttt{Fut}(\Gamma')}}{\Gamma \vdash \{\overline{\texttt{Int}\ x}, \overline{M}\}\ \{\overline{T\ z}\ ;\ s\} : (\mathcal{L}, \Theta \cdot \mathsf{L})}$$

Figure 6: Typing rules methods and programs.

of the typing to configurations; and the definition of a *later-stage* relation between behavioral types.

In order to state the subject reduction theorem, we first have to extend the type syntax of Figure 3 to be able to type the runtime configurations. We, thus, introduce the *behavioral type for configuration* K defined as follows:

$$\mathsf{K} \quad ::= \quad (\nu\,\overline{\varphi})(\Theta \cdot \mathsf{L}) \ \mid\ \mathsf{K}\,\&\,\mathsf{K}$$

and we extend the typing environment Γ to a *runtime typing environment* Δ which additionally maps method names to a pair of element (i.e $\Delta(m) = ((\alpha, \overline{\mathbb{x}}, X) \to \mathbb{r}, \mathsf{K})$) that are respectively the method signature and its runtime behavioral type.

The *later-stage* relation $\succeq_\Delta$ is a syntactic relationship between behavioral types whose basic laws are that a method invocation is larger than the instantiation of its method behavior, and a sum type is larger than each element of the sum. Formally, the later-stage relation is the least congruence with respect to runtime behavioral type that contains the rules in Figure 7.

We are now ready to state the Subject Reduction theorem.

Theorem 3.4 (Subject Reduction)**.** *Let $\Delta \vdash_R cn : \mathsf{K}$ and $cn \to cn'$. Then there exist Δ', K', and an injective renaming of actor names i such that*

- $\Delta' \vdash_R cn' : \mathsf{K}'$ *and*
- $i(\mathsf{K}) \succeq_\Delta \mathsf{K}'$

The proof is a case analysis on the reduction rule used in $cn \to cn'$, and can be found in Appendix A.

$$\text{LS-RuntimeEmpty}\quad \mathsf{K} \succeq_\Delta 0$$

$$\text{LS-Global}\quad \frac{\mathsf{K}_1 \succeq_\Delta \mathsf{K}_1'}{\mathsf{K}_1 \,\&\, \mathsf{K} \succeq_\Delta \mathsf{K}_1' \,\&\, \mathsf{K}}$$

$$\text{LS-Behavior}\quad \frac{\mathsf{K} = (\nu\,\overline{\varphi})(\Theta \cdot \mathsf{L}) \qquad \mathsf{K}' = (\nu\,\overline{\varphi'})(\Theta \cdot \mathsf{L}') \qquad \mathsf{L} \succeq_\Delta \mathsf{L}'}{\mathsf{K} \succeq_\Delta \mathsf{K}'}$$

$$\text{LS-Empty}\quad \mathsf{L} \succeq_\Delta 0$$

$$\text{LS-Invk}\quad \frac{\begin{array}{c}\Delta(f) = \lambda X.\mathtt{m}(\alpha', \overline{\mathtt{x}'}, X) \\ \Delta(m) = \big((\alpha, \overline{\mathtt{x}}, X) \to \mathtt{r}, \mathsf{K}_\mathtt{m}\big) \qquad \Delta \vdash \mathtt{m} : (\alpha', \overline{\mathtt{x}'}, X) \to \mathtt{r}' \\ \overline{\alpha} = fn(\mathsf{K}) \setminus fn(\alpha, \overline{\mathtt{x}}, \mathtt{r}) \qquad \overline{\alpha'} \cup fn(\alpha', \overline{\mathtt{x}'}, \mathtt{r}') = \varnothing\end{array}}{(\nu\,\overline{\varphi})(\Theta \cdot (f_\kappa \,\&\, \mathsf{L}) + \mathsf{L}_s) \succeq_\Delta (\nu\,\overline{\varphi})(\Theta \cdot \mathsf{L}_s) \,\&\, \mathsf{K}_\mathtt{m}[{}^{\overline{\alpha'}}/_{\overline{\alpha}}][{}^{\alpha', \overline{\mathtt{x}'}, \mathtt{r}'}/_{\alpha, \overline{\mathtt{x}}, \mathtt{r}}]}$$

$$\text{LS-Plus}\quad \mathsf{L}_1 + \mathsf{L}_2 \succeq_\Delta \mathsf{L}_i$$

$$\text{LS-Parallel}\quad \frac{\mathsf{L}_1 \succeq \mathsf{L}_1'}{\mathsf{L}_1 \,\&\, \mathsf{L} \succeq_\Delta \mathsf{L}_1' \,\&\, \mathsf{L}}$$

Figure 7: The later-stage relation.

4. Behavioral Type Analysis

The behavioral types of Section 3 are actually an extension of the so-called *lam* model – deadLock Analysis Model [14] – and in this section we correspondingly extend the theory.

The operational semantics of a behavioral type program $\big(\mathcal{L}, \Theta \cdot \mathsf{L}\big)$ is a transition system where *states* are pairs of a *future environment* Θ and a behavioral type. The definition of transitions requires some notation. *Contexts*, noted $\mathcal{C}[\,]$, are defined by the syntax

$$\mathcal{C}[\,] \quad ::= \quad [\,] \quad | \quad \mathsf{L} \,\&\, \mathcal{C}[\,] \quad | \quad \mathsf{L} + \mathcal{C}[\,]$$

As usual, $\mathcal{C}[\mathsf{L}]$ is the type where the hole of $\mathcal{C}[\,]$ is replaced by L. The *environment update of futures* (with $f' \notin \mathrm{dom}(\Theta)$) and *extended type substitution* are respectively defined as follows:

$$(\Theta[{}^{f'}/_{f}])(g) \stackrel{def}{=} \begin{cases} \Theta(f) & \text{if } g = f' \\ \Theta(g) & \text{if } g \neq f \end{cases}$$

$$\mathsf{L}[{}^{\mathtt{r}}/_{\mathtt{r}'_f}] = \mathsf{L}[{}^{0}/_{f_\kappa}]$$

The last substitution replaces all occurrences of a future synchronization by a null behavior. It is used when a (synchronized) value $\mathtt{r}$, i.e. a value that is not a future, is passed to a method. Indeed, to have the most generic signature, in method behaviors (stored in Θ), all formal parameters are assumed to be potentially a non synchronized future $\mathtt{r}'_f$.

The *transition relation* is the least one satisfying the rule

$$\text{BT-red}\quad \frac{\begin{array}{c}\Theta(f) = \lambda X.\mathtt{m}(\alpha, \overline{\mathtt{x}}, X) \\ \mathtt{m}(\alpha', \overline{\mathtt{x}'}, X) = (\nu\,\overline{\varphi})(\Theta_\mathtt{m} \cdot \mathsf{L}_\mathtt{m}) \in \mathcal{L} \qquad \kappa \neq X \qquad \overline{\varphi'} \text{ fresh} \\ \mathsf{L}' = \mathsf{L}_\mathtt{m}[{}^{\overline{\varphi'}}/_{\overline{\varphi}}][{}^{\alpha, \overline{\mathtt{x}}, \kappa}/_{\alpha', \overline{\mathtt{x}'}, X}] \qquad \Theta' = \Theta \cup \Theta_\mathtt{m}[{}^{\overline{\varphi'}}/_{\overline{\varphi}}][{}^{\alpha, \overline{\mathtt{x}}}/_{\alpha', \overline{\mathtt{x}'}}]\end{array}}{\Theta \cdot \mathcal{C}[\,f_\kappa\,] \to \Theta' \cdot \mathcal{C}[\,\mathsf{L}'\,]}$$

The initial state of $(\mathcal{L}, \Theta \cdot \mathsf{L})$ is $\Theta \cdot \mathsf{L}$. We write $\to^*$ for the reflexive and transitive closure of $\to$.

A behavioral type L is evaluated by successively replacing each future dereference with the corresponding type instance. Name creation is handled by replacing bound names of method bodies with fresh names. Notice that when $\kappa = X$ the rule does not apply meaning that the behavior of the return of a future (f_X) is not evaluated until that future is synchronized by some actor (f_β).

Example 4.1. Let us consider the behavioral type program of Example 3.1. $(\mathtt{fact}(\alpha, \Box_f, \Box_g, X) = (\nu\,\beta, f')(\Theta_\mathtt{fact} \cdot \mathsf{L}_\mathtt{fact}), \Theta \cdot \mathsf{L})$ where

$$\begin{array}{lcl} \Theta_\mathtt{fact} & = & \{f' \mapsto \lambda X.\,\mathtt{fact}(\beta, \Box, \Box, X)\} \\ \mathsf{L}_\mathtt{fact} & = & \Big(\big(f_\alpha + g_X + g_\alpha + f'_\star + f'_X\big) \,\&\, (X, \alpha)\Big) \\ \Theta & = & \{f'' \mapsto \lambda X.\mathtt{fact}(\alpha, \Box, \Box, X)\} \\ \mathsf{L} & = & f''_\star + f''_{main} \end{array}$$

By rule (BT-RED),

$$\begin{array}{l} \Theta \cdot \mathsf{L} \to \Theta' \cdot \big((g'_\star + g'_\star) \,\&\, (\star, \alpha)\big) + f''_{main} \\ \quad \to \Theta'' \cdot \big((g'_\star + g'_\star) \,\&\, (\star, \alpha)\big) + \big((g''_{main} + g''_{main}) \,\&\, (main, \alpha)\big) \\ \quad \to \cdots \end{array}$$

and so on, where

$$\begin{array}{lcl} \Theta' & = & \Theta \cup \{g' \mapsto \lambda X.\,\mathtt{fact}(\beta', \Box, \Box, X)\} \\ \Theta'' & = & \Theta' \cup \{g'' \mapsto \lambda X.\,\mathtt{fact}(\beta'', \Box, \Box, X)\}. \end{array}$$

The term g''_{main} will be replaced by the instantiated behavioral type of the method, producing the pair $(main, \beta'')$, reflecting the fact that the actor $main$ is synchronizing also the nested recursive calls.

Example 4.2. Let us consider the behavioral type program of Example 3.3. $(\mathtt{fact\_d2}(\alpha, \Box_f, \beta_g, X) = (\nu\,f')(\Theta_\mathtt{fd2} \cdot \mathsf{L}_\mathtt{fd2}), \Theta \cdot \mathsf{L}))$ where

$$\begin{array}{lcl} \Theta_\mathtt{fd2} & = & \{f' \mapsto \lambda X.\mathtt{fact\_d2}(\beta, \Box, \alpha, X)\} \\ \mathsf{L}_\mathtt{fd2} & = & \big(f_\alpha + g_\alpha + f'_\star + f'_\alpha\big) \,\&\, (X, \alpha) \\ \Theta & = & \{f'' \mapsto \lambda X.\mathtt{fact\_d2}(\alpha', \Box, \beta', X)\} \\ \mathsf{L} & = & f''_\star \end{array}$$

By rule (BT-RED),

$$\begin{array}{l} \Theta \cdot \mathsf{L} \to \Theta_1 \cdot \big((g'_\star + g'_{\beta'}) \,\&\, (\star, \alpha')\big) \\ \quad \to \Theta_2 \cdot \big((g'_\star + (g''_\star + g''_{\alpha'}) \,\&(\beta', \alpha')) \,\&\, (\star, \alpha')\big) \\ \quad \to \Theta_3 \cdot \big((g'_\star + (g''_\star + (g'''_\star + g'''_{\beta'}) \,\&(\alpha', \beta')) \,\&(\beta', \alpha')) \,\&\, (\star, \alpha')\big) \\ \quad \to \cdots \end{array}$$

and so on, where

$$\begin{array}{lcl} \Theta_1 & = & \Theta \cup \{g' \mapsto \lambda X.\,\mathtt{fact\_d2}(\beta', \Box, \alpha', X)\} \\ \Theta_2 & = & \Theta_1 \cup \{g'' \mapsto \lambda X.\,\mathtt{fact\_d2}(\alpha', \Box, \beta', X)\} \\ \Theta_3 & = & \Theta_2 \cup \{g''' \mapsto \lambda X.\,\mathtt{fact\_d2}(\beta', \Box, \alpha', X)\}. \end{array}$$

The occurrence of pairs (α', β') and (β', α') in parallel corresponds to the presence of a deadlock (See Definitions 4.3 and 4.4).

Notice that both computations of Examples 4.1 and 4.2 never end: the type grows in the number of "+"-terms, which in turn become larger and larger as the evaluation progresses. In principle, we do not know whether the former will produce a deadlock at some further evaluation step. Since the computation is infinite, we do not know when to stop looking for it. The fixpoint technique of Section 4.1 offers us a finite way to discriminate between deadlocked and deadlock-free programs.

4.1 Flattening, circularities and fixpoint definition of the interpretation function.

In this section we report the definitions from [14] slightly adapted to our current model. Let $\mathtt{R}$ be a set whose elements are either pairs (κ, β), where κ ranges over actor, future names, and variables X – see Figure 3 – or terms f_κ. We observe that, if the set of names is finite, then every set $\mathtt{R}$ built with such names is finite as well. In addition, the collection of all sets $\mathtt{R}$ is also finite. We use $\mathcal{R}, \mathcal{R}', \cdots$ to range over sets of relations $\{\mathtt{R}_1, \cdots, \mathtt{R}_m\}$. Let

- $\mathtt{R}^+$ be the *transitive closure* of $\mathtt{R}$ (namely $\mathtt{R}^+$ is the least relation such that $\mathtt{R} \subseteq \mathtt{R}^+$ and such that $(\kappa, \alpha), (\alpha, \beta) \in \mathtt{R}^+$ implies $(\kappa, \beta) \in \mathtt{R}^+$).
- $\{\mathtt{R}_1, \cdots, \mathtt{R}_m\} \Subset \{\mathtt{R}'_1, \cdots, \mathtt{R}'_n\}$ if and only if, for all $\mathtt{R}_i$, there is $\mathtt{R}'_j$ such that $\mathtt{R}_i \subseteq {\mathtt{R}'_j}^+$.

– $(\alpha_0,\alpha_1),\cdots,(\alpha_{n-1},\alpha_n) \Subset \{\mathtt{R}_1,\cdots,\mathtt{R}_m\}$ if and only if there is $\mathtt{R}_i$ such that $(\alpha_0,\alpha_1),\cdots,(\alpha_{n-1},\alpha_n) \in \mathtt{R}_i$.
– $\{\mathtt{R}_1,\cdots,\mathtt{R}_m\} \& \{\mathtt{R}'_1,\cdots,\mathtt{R}'_n\} \stackrel{def}{=} \{\mathtt{R}_i \cup \mathtt{R}'_j \mid 1 \leq i \leq m \text{ and } 1 \leq j \leq n\}$.

Definition 4.3. *A set* $\mathtt{R}$ has a circularity *if* $(\alpha,\alpha) \in \mathtt{R}^+$ *for some* α. *A set of elements* $\mathtt{R}$*, noted* $\mathcal{R}$*,* has a circularity *if there is* $\mathtt{R} \in \mathcal{R}$ *that has a circularity.*

For instance, $\{\{(\alpha,\beta),(\beta,\gamma)\},\ \{(\gamma,\beta),(\delta,\beta),(\beta,\gamma)\},\ \{(\eta,\delta)\}\}$ has a circularity because of the second element of the set.

Behavioral types define sets $\mathcal{R}$. This is displayed by the following function. Let $\mathcal{L}$ be a set of method definitions and let $I(\cdot)$, called *flattening*, be a function either on future environments and behavioral types or on method names that (*i*) maps a method name $\mathtt{m}$ defined in $\mathcal{L}$ to elements $\mathcal{R}$ and (*ii*) is defined on behavioral types as follows

$$\begin{array}{rll} I(\Theta\cdot\mathtt{0}) = & \{\varnothing\} & \\ I(\Theta\cdot(\kappa,\beta)) = & \{\{(\kappa,\beta)\}\} & \\ I(\Theta\cdot f_\kappa) = & I(\mathtt{m})[^{\alpha,\overline{\mathtt{x}},\kappa}/_{\alpha',\overline{\mathtt{x}}',X}] & \text{if } \Theta(f) = \lambda X.\mathtt{m}(\alpha,\overline{\mathtt{x}},X)] \\ & & \text{and } \mathtt{m}(\alpha',\overline{\mathtt{x}}',X) \text{ is defined in } \mathcal{L} \\ I(\Theta\cdot f_\kappa) = & \{\{f_\kappa\}\} & \text{if } f \notin \mathrm{dom}(\Theta) \\ I(\Theta\cdot\mathtt{L}\,\&\,\mathtt{L}') = & I(\Theta\cdot\mathtt{L})\,\&\,I(\Theta\cdot\mathtt{L}') & \\ I(\Theta\cdot\mathtt{L}+\mathtt{L}') = & I(\Theta\cdot\mathtt{L}) \cup I(\Theta\cdot\mathtt{L}') & \end{array}$$

Note that $I(\Theta\cdot\mathtt{L})$ is unique up to a renaming of names that do not occur free in $\mathtt{L}$. Let $I^\perp$ be the map such that, for every $\mathtt{m}$, $I^\perp(\mathtt{m}) = \{\varnothing\}$.

For example, let $\mathcal{L}$ define $\mathtt{m}(\alpha,\beta,\gamma,X)$ and $\mathtt{n}(\alpha,\beta,X)$ and let

$$\begin{array}{l} I(\mathtt{m}) = \{\{(\alpha,\beta),(X,\gamma)\}\} \qquad I(\mathtt{n}) = \{\{(\beta,\alpha)\}\} \\ \Theta = \{f \mapsto \lambda X.\mathtt{m}(\alpha,\beta,\gamma,X), f' \mapsto \lambda X.\mathtt{n}(\beta,\gamma,X), \\ \qquad f'' \mapsto \lambda X.\mathtt{m}(\delta,\beta,\gamma,X)\} \\ \mathtt{L} = f_\star \,\&\, f''_\star + (\alpha,\beta) \,\&\, f'_X \,\&\, f''_\alpha + (\eta,\delta). \end{array}$$

Then

$$\begin{array}{rl} I(\Theta\cdot\mathtt{L}) = & \{\{(\alpha,\beta),(\star,\gamma),(\delta,\beta),(\star,\gamma)\}, \\ & \{(\alpha,\beta),(\gamma,\beta),(\delta,\beta),(\alpha,\gamma)\},\ \{(\eta,\delta)\}\} \\ I^\perp(\Theta\cdot\mathtt{L}) = & \{\varnothing,\{(\alpha,\beta)\},\{(\eta,\delta)\}\}\ . \end{array}$$

Definition 4.4. *A state* $\Theta\cdot\mathtt{L}$ has a circularity *if* $I^\perp(\Theta\cdot\mathtt{L})$ *has a circularity. A behavioral type program* $(\mathcal{L},\Theta\cdot\mathtt{L})$ has a circularity *if there exist* Θ' *and* $\mathtt{L}'$ *such that* $\Theta\cdot\mathtt{L} \to^* \Theta'\cdot\mathtt{L}'$ *and* $\Theta'\cdot\mathtt{L}'$ *has a circularity.*

The basic item of our algorithm is the computation of methods' interpretation. This computation is performed by means of a standard fixpoint technique that is detailed below.

Let $(\mathcal{L},\Theta\cdot\mathtt{L})$ be a program such that pairwise different method definitions in $\mathcal{L}$ have disjoint formal parameters. Let A be the set of (*i*) formal parameters of definitions in $\mathcal{L}$, of (*ii*) free names in $\Theta\cdot\mathtt{L}$ and (*iii*) containing a special fresh name $\varkappa$. Since A is finite, then every set $\mathtt{R}_A$ built with names in A is finite and similarly for $\mathcal{R}_A$. In particular, the sets $\mathcal{R}_A$ are ordered by the $\subseteq$ relation and form a *finite* lattice [8].

Definition 4.5. *Let* $\mathcal{L} = \{\mathtt{m}_i(\alpha_i,\overline{\mathtt{x}}_i,X_i) = (\nu\,\overline{\varphi}_i)(\Theta_i\cdot\mathtt{L}_i) \mid i \in \{1..k\}\}$. *The family of flattening functions* $I^{(k)}$ *is defined as follows*

$$\begin{array}{rcl} I^{(0)}(\mathtt{m}_i) & = & \{\varnothing\} \\ I^{(k+1)}(\mathtt{m}_i) & = & \{\ \mathtt{proj}_{\alpha_i,\overline{\mathtt{x}}_i,X_i}(R^+) \mid R \in I^{(k)}(\Theta_i\cdot\mathtt{L}_i)\} \end{array}$$

where $\mathtt{proj}_{\alpha,\overline{\mathtt{x}},X}(R) \stackrel{def}{=} \{(\beta,\gamma) \mid (\beta,\gamma) \in R \text{ and } \beta,\gamma \in \alpha,\overline{\mathtt{x}},X\} \bigcup \{(\varkappa,\varkappa) \mid (\delta,\delta) \in R \text{ and } \delta \notin \alpha,\overline{\mathtt{x}},X\}$.

We notice that $I^{(0)}$ is the function $I^\perp$ of the previous section. Since, for every k, $I^{(k)}(\mathtt{m}_i)$ ranges over a finite lattice, by the fixpoint theory [8], there exists m such that $I^{(m)}$ is a fixpoint, namely $I^{(m)} \approx I^{(m+1)}$ where $\approx$ is the equivalence relation induced by $\Subset$. In the following, we let I, called the *interpretation function* (of a behavioral type), be the least fixpoint $I^{(m)}$.

The following theorem states the correctness and completeness of our algorithm. Similarly to [14], there is a relation between the circularities of the set $I^{(k)}(\Theta\cdot\mathtt{L})$ and, whenever $\Theta\cdot\mathtt{L} \to \Theta'\cdot\mathtt{L}'$, between the circularities of $I^{(k)}(\Theta\cdot\mathtt{L})$ and of $I^{(k)}(\Theta'\cdot\mathtt{L}')$.

Theorem 4.6. *A behavioural type program* $(\mathcal{L},\Theta\cdot\mathtt{L})$ *has a circularity if and only if* $I_{\mathcal{L}}(\Theta\cdot\mathtt{L})$ *has a circularity.*

The proof can be found in Appendix B.

Example 4.7. Let us consider the behavioral type program of Example 3.1. $(\mathtt{fact}(\alpha,\square_f,\square_g,X) = (\nu\,\beta,f')(\Theta_{\mathtt{fact}}\cdot\mathtt{L}_{\mathtt{fact}}),\Theta\cdot\mathtt{L})$ where

$$\begin{array}{l} \Theta_{\mathtt{fact}} = \{f' \mapsto \lambda X.\,\mathtt{fact}(\beta,\square,\square,X)\} \\ \mathtt{L}_{\mathtt{fact}} = \Big((f_\alpha + g_X + g_\alpha + f'_\star + f'_X)\,\&\,(X,\alpha)\Big) \\ \Theta = \{f'' \mapsto \lambda X.\mathtt{fact}(\alpha,\square,\square,X)\} \\ \mathtt{L} = f''_\star + f''_{main} \end{array}$$

By Definition 4.5,

$$\begin{array}{l} I^{(0)}(\mathtt{fact}) = \{\varnothing\} \\ I^{(1)}(\mathtt{fact}) = \{\{f_\alpha,(X,\alpha)\},\{g_X,(X,\alpha)\},\{g_\alpha,(X,\alpha)\}\} \\ I^{(2)}(\mathtt{fact}) = \{\{f_\alpha,(X,\alpha)\},\{g_X,(X,\alpha)\},\{g_\alpha,(X,\alpha)\},\{(X,\alpha)\}\} \\ I^{(3)}(\mathtt{fact}) = \{\{f_\alpha,(X,\alpha)\},\{g_X,(X,\alpha)\},\{g_\alpha,(X,\alpha)\},\{(X,\alpha)\}\} \end{array}$$

Notice that $I^{(2)}(\mathtt{fact}) \approx I^{(3)}(\mathtt{fact})$, thus $I^{(2)}$ is a fixpoint for $\mathtt{fact}$. We then compute

$$I^{(2)}(\Theta\cdot\mathtt{L}) = \{\{(\star,\alpha)\},\{(main,\alpha)\}\}.$$

Example 4.8. Let us consider the behavioral type in Example 3.3. for the program in Listing 3 $(\mathtt{fact\_d2}(\alpha,\square_f,\beta_g,X) = (\nu\,f')(\Theta_{\mathtt{fd2}}\cdot\mathtt{L}_{\mathtt{fd2}}),\Theta\cdot\mathtt{L}))$ where

$$\begin{array}{l} \Theta_{\mathtt{fd2}} = \{f' \mapsto \lambda X.\mathtt{fact\_d2}(\beta,\square,\alpha,X)\} \\ \mathtt{L}_{\mathtt{fd2}} = \big(f_\alpha + g_\alpha + f'_\star + f'_\alpha\big)\,\&\,(X,\alpha) \\ \Theta = \{f'' \mapsto \lambda X.\mathtt{fact\_d2}(\alpha',\square,\beta',X)\} \\ \mathtt{L} = f''_\star \end{array}$$

It turns out that $I^{(3)}(\mathtt{fact\_d2}) = \{\{f_\alpha,(X,\alpha)\},\{g_\alpha,(X,\alpha)\},\{(\varkappa,\varkappa),(X,\alpha)\}\}$ is the fixpoint and, if we compute $I^{(3)}(\Theta\cdot\mathtt{L})$ we get the set $\{\{(\star,\alpha)\},\{(\varkappa,\varkappa),(\star,\alpha)\}\}$ which contains a circularity.

5. Correctness

The correctness of our system guarantees that, if the deadlock-freedom of a behavioral type program associated to a gASP program is assessed, then also the corresponding gASP program is guaranteed to be deadlock-free. In other words we are proving that if the analysis shows that no deadlock is present in the behavioral type of the original program, then none of its executions can lead to a deadlock. To this end, we prove that if there is no circularity in the type of a runtime configuration then this configuration exhibits no deadlock, and that if a configuration reduces to a configuration with a circularity then the original configuration already had a circularity. This ensures that if no circularity is found in the behavioral type of a gASP program then there is no deadlock in the original program.

Theorem 5.1. *Let* P *be a* gASP *program and cn be a configuration of its operational semantics, with behavioral type* $\Theta\cdot\mathtt{L}$.

1. *If* $\Theta\cdot\mathtt{L}$ *has no circularity then cn is deadlock-free;*
2. *if* $cn \to cn'$ *and the behavioral type* $\Theta'\cdot\mathtt{L}'$ *of* cn' *has a circularity, then a circularity is already present in* $\Theta\cdot\mathtt{L}$*, the behavioral type of cn;*

The theorem is proven by relying on Theorem 3.4 (subject reduction) and on a crucial property of the later stage relation:

Theorem 5.2. *If* $\Theta\cdot\mathtt{L} \succeq \Theta'\cdot\mathtt{L}'$*, then* $I_{\mathcal{L}}(\Theta'\cdot\mathtt{L}') \Subset I_{\mathcal{L}}(\Theta\cdot\mathtt{L})$.

The proof of Theorem 5.2, as well as the proofs of the foregoing theorems, is very similar to the corresponding one in [14].

6. Example

In this section we present a more detailed example in which we show our analysis applied on the dining philosophers problem (Listing 4). For the sake of simplicity we present a version with only two philosophers in order to better focus on the steps of the analysis. We start detailing the execution of the program showing how a deadlocked configuration can be reached. Then we present the behavioral type of the program given by our type system and finally we apply the analysis to detect circularities.

Listing 4: Dining Philosopher

```
1  class Philosopher{
2  Int behave(Fork fork1, Fork fork2) {
3      Int fut, aux, c;
4      fut = fork1.grab(fork2);
5      aux = fut + 0;
6      c = this.behave(fork1, fork2);
7      return c;
8  }
9  }
10
11 class Fork{
12 Int grab(Fork z) {
13     Int aux, f;
14     f = z.grab_second();
15     aux = f + 0;
16     return aux;
17 }
18
19 Int grab_second() {
20     return 0;
21   }
22 }}
23
24 // MAIN //
25 {
26   Fork fL, fR;
27   Philosopher p1, p2;
28   Int fut1, fut2, aux;
29   fL = new Fork() ;
30   fR = new Fork() ;
31   p1 = new Philosopher();
32   p2 = new Philosopher();
33   fut1 = p1.behave(fL, fR);
34   fut2 = p2.behave(fR, fL);
35   aux = fut1 + fut2;
36 }
```

In Figure 8 we detail the evaluation of the program. The step number 1 refers to the four reductions corresponding to the lines of code from 29 to 32. Four actors are created: two of them represent the two philosophers, respectively actors γ and δ, and the two others represent the two forks, actors α and β. For simplicity we will call p_γ the philosopher running on actor γ, p_δ the philosopher running on actor δ; similarly, we refer to the two forks as $fork_\alpha$ and $fork_\beta$. The second and third steps correspond to the two invocations of the method `behave` (line 33, 34). This method encodes the behavior of a philosopher that wants to grab the two forks and then start eating. The two invocations of `behave` are handled by p_γ and by p_δ. Both method invocations are immediately served because both actors have no running process. At this point, notice that the computation in actor $main$ can not continue because it needs the results of the two methods just invoked. The step number 4 shows how the two philosophers grab the fork on their right[2], respectively $fork_\alpha$ for p_γ and $fork_\beta$ for p_δ, invoking the method `grab` on the actor α or β respectively. As before the executions of the two methods `grab` can immediately be served because the actors α and β have no running process; additionally the computation in p_1 and p_2 can not proceed waiting the result of the `grab` method. At this point, as shown in step 5, $fork_\alpha$ invokes the method `grab_second` on $fork_\beta$ and $fork_\beta$ does the same on $fork_\alpha$. Both requests are put in the queue of the corresponding actor instead but cannot be served because both actor α and β have already a running process. It is trivial to see that no other rule can be applied and the reached configuration is deadlocked.

[2] We detail here the execution that leads to a deadlock, of course another scheduling is also possible.

Here is the behavioral types of the program of Listing 4:

$$\begin{array}{l}(\texttt{behave}(\gamma,\alpha_b,\beta_{b'},X) = (\nu\, g,g')(\Theta_{\texttt{behave}}\cdot \mathsf{L}_{\texttt{behave}}),\\ \texttt{grab}(\alpha,\beta_{d'},X) = (\nu\, d)(\Theta_{\texttt{grab}}\cdot \mathsf{L}_{\texttt{grab}}),\\ \texttt{grab\_second}(\alpha,X) = (\nu\,)(\varnothing\cdot(X,\alpha)),\Theta\cdot\mathsf{L})\end{array}$$

in which

$$\begin{array}{lcl}\Theta & = & \{f \mapsto \lambda X.\texttt{behave}(\gamma,\alpha,\beta,X)\\ & & \quad f' \mapsto \lambda X.\texttt{behave}(\delta,\beta,\alpha,X)\},\\ \mathsf{L} & = & f_\star + f'_\star \,\&\, f_\star + f_{main} \,\&\, f'_\star + f'_{main},\\ \Theta_{\texttt{behave}} & = & \{g \mapsto \lambda X.\texttt{grab}(\alpha,\beta_{b'},X)\\ & & \quad g' \mapsto \lambda X.\texttt{behave}(\gamma,\alpha,\beta_{b'},X)\},\\ \mathsf{L}_{\texttt{behave}} & = & (b_\gamma + g_\star + g_\gamma + g'_\star)\,\&\,(X,\gamma),\\ \Theta_{\texttt{grab}} & = & \{d \mapsto \lambda X.\texttt{grab\_second}(\beta,X)\},\\ \mathsf{L}_{\texttt{grab}} & = & (d'_\alpha + d_\star + d_\alpha)\,\&\,(X,\alpha).\end{array}$$

Figure 9 shows the reduction of the behavioral type done applying the rule BT-RED presented in section 4. Starting from the behavioral type of the main function $\Theta\cdot\mathsf{L}$, at each reduction step, applying BT-RED, we replace one term of the behavioral types that refers to the execution of a method by the behavioral type of the body of that method adequately instantiated. Knowing that $\Theta(f) = \lambda X.\texttt{behave}(\gamma,\alpha,\beta,X)$ and the behavioral type of the method `behave` is $\texttt{behave}(\gamma,\alpha_b,\beta_b,X) = (\nu\, g,g')(\Theta_{\texttt{behave}}\cdot\mathsf{L}_{\texttt{behave}})$, the first reduction replaces f_{main} with $\mathsf{L}_{\texttt{behave}}$ on which we replace the formal parameters with the actual parameters with $\mathsf{L}_{\texttt{behave}}[^{\gamma,\alpha,\beta,main}/_{\gamma,\alpha_b,\beta_{b'},X}] = (g_\star + g_\gamma + g'_\star)\,\&(main,\gamma)$. Similarly, we compute the other steps of reduction. In Figure 9 we do not show the complete reduction of the behavioral type, that can be infinite; instead we guide the reduction through some steps that are relevant to reach the significant state shown in the last line. After simplification, we obtain a type in which one term has the following shape $\Theta_4\cdot(\cdots+(\alpha,\beta)\,\&(\gamma,\alpha)\,\&(main,\gamma)\,\&(\beta,\alpha)\,\&(\delta,\beta)\,\&(\star,\delta)+\cdots)$ in which we can identify a circularity caused by the presence of the pairs (α,β) and (β,α).

Knowing the behavioral type of our program we can evaluate the flattening function for each method by Definition 4.5.

$$\begin{array}{l}I^{(0)}(\texttt{grab\_second}) = \{\varnothing\},\\ I^{(1)}(\texttt{grab\_second}) = \{\{(X,\alpha)\}\}\\ I^{(0)}(\texttt{grab}) = \{\varnothing\}\\ I^{(1)}(\texttt{grab}) = \{\{d'_\alpha,(X,\alpha)\},\{(X,\alpha)\},\{(\alpha,\beta),(X,\alpha)\}\}\\ I^{(0)}(\texttt{behave}) = \{\varnothing\}\\ I^{(1)}(\texttt{behave}) = \{\{b_\gamma,(X,\gamma)\},\{b'_\beta,(X,\gamma)\},\{(X,\gamma)\},\\ \qquad\qquad \{(\alpha,\beta),(X,\gamma)\},\{b'_\beta,(\gamma,\alpha),(X,\gamma)\},\\ \qquad\qquad \{(\gamma,\alpha),(X,\gamma)\},\{(\alpha,\beta),(\gamma,\alpha),(X,\gamma)\}\}\end{array}$$

Finally we compute $I(\Theta\cdot\mathsf{L})$. We show below only a small relevant fragment of the result and the complete version is shown in Figure 10.

$$\begin{array}{l}I(\Theta\cdot\mathsf{L}) = \{\cdots,\{(\alpha,\beta),(\beta,\alpha),(\star,\delta),(main,\gamma)\},\\ \qquad\qquad \{(\alpha,\beta),(main,\gamma),(\beta,\alpha),(\delta,\beta),(\star,\delta)\},\\ \qquad\qquad \{(\alpha,\beta),(\gamma,\alpha),(main,\gamma),(\beta,\alpha),(\delta,\beta),(\star,\delta)\},\cdots\}\end{array}$$

As we have mentioned above we can see three states that compose the entire result, we take these three states because in all of them is possible to find the circularity that cause the deadlock in the dining philosopher problem. More precisely we can see in each of the three states the pairs (α,β) and (β,α) that are exactly the

$main(\varnothing, \{[\text{destiny} \mapsto f_{main}, \text{fL}, \text{fR}, \text{p1}, \text{p2}, \text{fut1}, \text{fut2}, \text{aux} \mapsto \bot] \mid \texttt{fL = newFork(); fR = newFork(); } \cdots\}, \varnothing)$
1. $(New) \to (New) \to (New) \to (New) \to$
$main(\varnothing, \{[\text{destiny} \mapsto f_{main}, \text{fL} \mapsto \alpha, \text{fR} \mapsto \beta, \text{p1} \mapsto \gamma, \text{p2}, \text{fut1}, \text{fut2}, \text{aux} \mapsto \bot] \mid \texttt{fut1 = p1.behave(fL, fR); p2 = newPhilosopher(); } \cdots\}, \varnothing)$
$\alpha(\varnothing, \varnothing, \varnothing) \quad \beta(\varnothing, \varnothing, \varnothing) \quad \gamma(\varnothing, \varnothing, \varnothing) \quad \delta(\varnothing, \varnothing, \varnothing)$
2. $(Invk) \to (Serve) \to$
$main(\varnothing, \{[\text{destiny} \mapsto f_{main}, \text{fL} \mapsto \alpha, \text{fR} \mapsto \beta, \text{p1} \mapsto \gamma, \text{fut1} \mapsto f_0, \text{p2}, \text{fut2}, \text{aux} \mapsto \bot] \mid \texttt{p2 = newPhilosopher(); fut2 = p2.behave(fR, fL); } \cdots\}, \varnothing)$
$\alpha(\varnothing, \varnothing, \varnothing) \quad \beta(\varnothing, \varnothing, \varnothing) \quad \gamma(\varnothing, \{[\text{destiny} \mapsto f_\gamma, \text{fork1} \mapsto \alpha, \text{fork2} \mapsto \beta, \text{fut}, \text{aux}, \text{c} \mapsto \bot] \mid \{\textit{body-of-}\texttt{grab}\}, \varnothing) \quad f_0(\bot) \quad \delta(\varnothing, \varnothing, \varnothing)$
3. $(Invk) \to (Serve) \to$
$main(\varnothing, \{[\text{destiny} \mapsto f_{main}, \text{fL} \mapsto \alpha, \text{fR} \mapsto \beta, \text{p1} \mapsto \gamma, \text{fut1} \mapsto f_0, \text{p2} \mapsto \delta, \text{fut2} \mapsto f_1, \text{aux} \mapsto \bot] \mid \texttt{aux = fut1 + fut2; } \}, \varnothing)$
$\alpha(\varnothing, \varnothing, \varnothing) \quad \beta(\varnothing, \varnothing, \varnothing)$
$\gamma(\varnothing, \{[\text{destiny} \mapsto f_0, \text{fork1} \mapsto \alpha, \text{fork2} \mapsto \beta, \text{fut}, \text{aux}, \text{c} \mapsto \bot] \mid \texttt{fut = fork1.grab(fork2); aux = fut + 0;} \cdots\}, \varnothing) \quad f_0(\bot)$
$\delta(\varnothing, \{[\text{destiny} \mapsto f_1, \text{fork1} \mapsto \beta, \text{fork2} \mapsto \alpha, \text{fut}, \text{aux}, \text{c} \mapsto \bot] \mid \texttt{fut = fork1.grab(fork2); aux = fut + 0;} \cdots\}, \varnothing) \quad f_1(\bot)$
4. $(Invk) \to (Serve) \to (Invk) \to (Serve) \to$
$main(\varnothing, \{[\text{destiny} \mapsto f_{main}, \text{fL} \mapsto \alpha, \text{fR} \mapsto \beta, \text{p1} \mapsto \gamma, \text{fut1} \mapsto f_0, \text{p2} \mapsto \delta, \text{fut2} \mapsto f_1, \text{aux} \mapsto \bot] \mid \texttt{aux = fut1 + fut2; } \}, \varnothing)$
$\alpha(\varnothing, \{[\text{destiny} \mapsto f_2, \text{z} \mapsto \alpha, \text{aux}, \text{f} \mapsto \bot] \mid \texttt{f = z.grab\_second(); aux = f + 0;} \cdots\}, \varnothing) \quad f_2(\bot)$
$\beta(\varnothing, \{[\text{destiny} \mapsto f_3, \text{z} \mapsto \beta, \text{aux}, \text{f} \mapsto \bot] \mid \texttt{f = z.grab\_second(); aux = f + 0;} \cdots\}, \varnothing) \quad f_3(\bot)$
$\gamma(\varnothing, \{[\text{destiny} \mapsto f_0, \text{fork1} \mapsto \alpha, \text{fork2} \mapsto \beta, \text{fut} \mapsto f_2, \text{aux}, \text{c} \mapsto \bot] \mid \texttt{aux = fut + 0; c = this.behave(fork1, fork2);} \cdots\}, \varnothing) \quad f_0(\bot)$
$\delta(\varnothing, \{[\text{destiny} \mapsto f_1, \text{fork1} \mapsto \beta, \text{fork2} \mapsto \alpha, \text{fut} \mapsto f_3, \text{aux}, \text{c} \mapsto \bot] \mid \texttt{aux = fut + 0; c = this.behave(fork1, fork2);} \cdots\}, \varnothing) \quad f_1(\bot)$
5. $(Invk) \to (Invk)$
$main(\varnothing, \{[\text{destiny} \mapsto f_{main}, \text{fL} \mapsto \alpha, \text{fR} \mapsto \beta, \text{p1} \mapsto \gamma, \text{fut1} \mapsto f_0, \text{p2} \mapsto \delta, \text{fut2} \mapsto f_1, \text{aux} \mapsto \bot] \mid \texttt{aux = fut1 + fut2; } \}, \varnothing)$
$\alpha(\varnothing, \{[\text{destiny} \mapsto f_2, \text{z} \mapsto \alpha, \text{f} \mapsto f_4, \text{aux} \mapsto \bot] \mid \texttt{aux = f + 0; return aux;} \cdots\}, \{\textit{body-of-}\texttt{grab\_second}\}) \quad f_2(\bot) \quad f_5(\bot)$
$\beta(\varnothing, \{[\text{destiny} \mapsto f_3, \text{z} \mapsto \beta, \text{f} \mapsto f_5, \text{aux} \mapsto \bot] \mid \texttt{aux = f + 0; return aux;} \cdots\}, \{\textit{body-of-}\texttt{grab\_second}\}) \quad f_3(\bot) \quad f_4(\bot)$
$\gamma(\varnothing, \{[\text{destiny} \mapsto f_0, \text{fork1} \mapsto \alpha, \text{fork2} \mapsto \beta, \text{fut} \mapsto f_2, \text{aux}, \text{c} \mapsto \bot] \mid \texttt{aux = fut + 0; c = this.behave(fork1, fork2);} \cdots\}, \varnothing) \quad f_0(\bot)$
$\delta(\varnothing, \{[\text{destiny} \mapsto f_1, \text{fork1} \mapsto \beta, \text{fork2} \mapsto \alpha, \text{fut} \mapsto f_3, \text{aux}, \text{c} \mapsto \bot] \mid \texttt{aux = fut + 0; c = this.behave(fork1, fork2);} \cdots\}, \varnothing) \quad f_1(\bot)$

Figure 8: Reduction of dining philosopher problem.

actors representing the two forks that are waiting one the result of the other.

7. Related work

The behavioural type model has been introduced in [11, 12] for detecting deadlocks in a concurrent object-oriented language; the decision algorithm for the circularity of behavioural types has been defined in [14]. This technique improves the previous deadlock-freedom analysis [20] in a significative way, as demonstrated in [14, 21]. In [6] circular dependencies among processes are detected as erroneous configurations, but dynamic creation of names is not treated. An alternative model checking technique is proposed in [4] for multi-threaded asynchronous communication languages with futures (as ABS). This technique addresses infinite-state programs that admit thread creation but not dynamic resource creation. The problem of verifying deadlocks in infinite state models has been studied in other contributions. For example, [23] compare a number of unfolding algorithms for Petri Nets with techniques for safely cutting potentially infinite unfoldings. Also in this work, dynamic resource creation is not addressed. We refer to [12, 14] for further related works about deadlock analysis and for comparisons with our technique.

The language gASP is a subcalculus of ASP [7]. ASP adheres to the *active object paradigm* that aims at easing the distributed programming by abstracting away the notions of concurrency and location. In fact, active objects are mono-threaded, thus data race-conditions are prevented without using either locks or synchronized blocks. An extension of ASP, called ProActive, has been prototyped by using Java libraries [17]. Studying deadlock detection of ProActive systems will be a relevant application of the work presented in this paper.

It is worth to mention the presence of active object languages that do not use wait-by-necessity synchronizations, such as Creol [19], which did not admit to communicate future references to other actors, and its extension ABS where futures are first-class entities [9]. In particular, in the case of ABS, futures are explicitly typed and the user is fully aware of the number of indirections to unroll the future before accessing the actual value. Another calculus with futures as first-class entities is $\lambda(\mathsf{Fut})$ [22], which admits to pass futures as arguments of invocations. In particular, the invocation arguments may be *future handlers*, which allow to delegate another task the fulfilment of a future. The problematic issue is that several tasks may try to use the same handler to fill a future, which is an error that is not easy to detect in $\lambda(\mathsf{Fut})$.

A powerful operation on futures has been recently investigated in the Encore language [5]. This operation, called *future chaining*, allows programmers to compose futures with closures and returns a handler that is executed when a future becomes available. The future chaining would probably permit an encoding of a variable number of future unrolling, thus enabling the compilation of implicit wait-by-necessity synchronizations into explicit ones – see Section 1. This is an open problem that definitely requires detailed investigations.

8. Conclusion

In this paper we have studied deadlock detection for gASP, a basic actor calculus with wait-by-necessity synchronization. We aggregate two complementary techniques: a type system for extracting behavioural descriptions out of programs and a fixpoint technique for computing dependency models of behavioural descriptions. The work builds on and extends previous work where a similar technique has been used to detect deadlocks in pi-calculus [14, 21] and in an object-oriented language [11–13, 15]. In particular, the extension addresses the possible unbounded nesting of futures and the corresponding management in the behavioural descriptions.

The technical contribution of this paper highlights the differences between explicit and implicit futures: while explicit futures enable the synchronization upon the end of the execution of a method, implicit futures trigger synchronization upon access to some data. The data-flow implicit synchronization makes programming easier and execution more efficient as the program is only blocked if data is really needed and in an automatic way. However reasoning and finding deadlocks on a program with data-flow synchronization is more difficult, both for the programmer and for automatic tools. This paper shows that the analysis of such data-flow synchronization is possible and that the programming model can be at the same time easier and more efficient to program, while enabling automatic detection of deadlocks.

$$
\begin{aligned}
&\Theta \cdot (f_\star + f'_\star \,\&\, f_\star + \boxed{f_{main}} \,\&\, f'_\star + f'_{main}) \\
&\to \Theta_1 \cdot (\cdots + (g_\star + g_\gamma + g'_\star)\,\&(main,\gamma)\,\&\, \boxed{f'_\star} + \cdots) \\
&\to \Theta_2 \cdot (\cdots + (g_\star + \boxed{g_\gamma} + g'_\star)\,\&(main,\gamma)\ \&\,(g''_\star + g''_\delta + g'''_\star)\,\&(\star,\delta) + \cdots) \\
&\to \Theta_3 \cdot (\cdots + (g_\star + (d_\star + d_\alpha)\,\&(\gamma,\alpha) + g'_\star)\,\&(main,\gamma)\,\&(g''_\star + \boxed{g''_\delta} + g'''_\star)\,\&(\star,\delta) + \cdots) \\
&\to \Theta_4 \cdot (\cdots + (g_\star + (d_\star + \boxed{d_\alpha})\,\&(\gamma,\alpha) + g'_\star)\,\&(main,\gamma)\,\&(g''_\star + (d'_\star + d'_\beta)\,\&(\delta,\beta) + g'''_\star)\,\&(\star,\delta) + \cdots) \\
&\to \Theta_4 \cdot (\cdots + (g_\star + (d_\star + (\alpha,\beta))\,\&(\gamma,\alpha) + g'_\star)\,\&(main,\gamma)\,\&(g''_\star + (d'_\star + \boxed{d'_\beta})\,\&(\delta,\beta) + g'''_\star)\,\&(\star,\delta) + \cdots) \\
&\to \Theta_4 \cdot (\cdots + (g_\star + (d_\star + \boxed{(\alpha,\beta)})\,\&\,\boxed{(\gamma,\alpha)} + g'_\star)\,\&\,\boxed{(main,\gamma)}\,\&(g''_\star + (d'_\star + \boxed{(\beta,\alpha)})\,\&\,\boxed{(\delta,\beta)} + g'''_\star)\,\&(\star,\delta) + \cdots) \\
&\to \cdots
\end{aligned}
$$

$$
\begin{aligned}
&\Theta = \{f \mapsto \lambda X.\texttt{behave}(\gamma,\alpha,\beta,X),\, f' \mapsto \lambda X.\texttt{behave}(\delta,\beta,\alpha,X)\} \\
&\Theta_1 = \Theta \cup \{g \mapsto \lambda X.\texttt{grab}(\alpha,\beta,X),\, g' \mapsto \lambda X.\texttt{behave}(\gamma,\alpha,\beta,X)\} \\
&\Theta_2 = \Theta_1 \cup \{g'' \mapsto \lambda X.\texttt{grab}(\beta,\alpha,X),\, g''' \mapsto \lambda X.\texttt{behave}(\delta,\beta,\alpha,X)\} \\
&\Theta_3 = \Theta_2 \cup \{d \mapsto \lambda X.\texttt{grab\_second}(\beta,X)\} \\
&\Theta_4 = \Theta_3 \cup \{d' \mapsto \lambda X.\texttt{grab\_second}(\alpha,X)\}
\end{aligned}
$$

Figure 9: Behavioral types reduction.

$$
\begin{aligned}
I(\Theta \cdot \mathsf{L}) = \{&\{(\star,\gamma)\},\{(\alpha,\beta),(\star,\gamma)\},\{(\gamma,\alpha),(\star,\gamma)\},\{(\alpha,\beta),(\gamma,\alpha),(\star,\gamma)\},\\
&\{(\star,\gamma),(\star,\delta)\},\{(\alpha,\beta),(\star,\gamma),(\star,\delta)\},\{(\gamma,\alpha),(\star,\gamma),(\star,\delta)\},\{(\alpha,\beta),(\gamma,\alpha),(\star,\gamma),(\star,\delta)\},\\
&\{(\star,\gamma),(\beta,\alpha),(\star,\delta)\},\{(\alpha,\beta),(\star,\gamma),(\beta,\alpha),(\star,\delta)\},\{(\gamma,\alpha),(\star,\gamma),(\beta,\alpha),(\star,\delta)\},\{(\alpha,\beta),(\gamma,\alpha),(\star,\gamma),(\beta,\alpha),(\star,\delta)\},\\
&\{(\star,\gamma),(\delta,\beta),(\star,\delta)\},\{(\alpha,\beta),(\star,\gamma),(\delta,\beta),(\star,\delta)\},\{(\gamma,\alpha),(\star,\gamma),(\delta,\beta),(\star,\delta)\},\{(\alpha,\beta),(\gamma,\alpha),(\star,\gamma),(\delta,\beta),(\star,\delta)\},\\
&\{(\star,\gamma),(\beta,\alpha),(\delta,\beta),(\star,\delta)\},\{(\alpha,\beta),(\star,\gamma),(\beta,\alpha),(\delta,\beta),(\star,\delta)\},\{(\gamma,\alpha),(\star,\gamma),(\beta,\alpha),(\delta,\beta),(\star,\delta)\},\\
&\{(\alpha,\beta),(\gamma,\alpha),(\star,\gamma),(\beta,\alpha),(\delta,\beta),(\star,\delta)\},\\
&\{(main,\gamma),(\star,\delta)\},\{(\alpha,\beta),(main,\gamma),(\star,\delta)\},\{(\gamma,\alpha),(main,\gamma),(\star,\delta)\},\{(\alpha,\beta),(\gamma,\alpha),(main,\gamma),(\star,\delta)\},\\
&\{(main,\gamma),(\beta,\alpha),(\star,\delta)\},\{(\alpha,\beta),(main,\gamma),(\beta,\alpha),(\star,\delta)\},\{(\gamma,\alpha),(main,\gamma),(\beta,\alpha),(\star,\delta)\},\\
&\{(\alpha,\beta),(\gamma,\alpha),(main,\gamma),(\beta,\alpha),(\star,\delta)\},\\
&\{(main,\gamma),(\delta,\beta),(\star,\delta)\},\{(\alpha,\beta),(main,\gamma),(\delta,\beta),(\star,\delta)\},\{(\gamma,\alpha),(main,\gamma),(\delta,\beta),(\star,\delta)\},\\
&\{(\alpha,\beta),(\gamma,\alpha),(main,\gamma),(\delta,\beta),(\star,\delta)\},\\
&\{(main,\gamma),(\beta,\alpha),(\delta,\beta),(\star,\delta)\},\{(\alpha,\beta),(main,\gamma),(\beta,\alpha),(\delta,\beta),(\star,\delta)\},\{(\gamma,\alpha),(main,\gamma),(\beta,\alpha),(\delta,\beta),(\star,\delta)\},\\
&\{(\alpha,\beta),(\gamma,\alpha),(main,\gamma),(\beta,\alpha),(\delta,\beta),(\star,\delta)\},\\
&\{(main,\delta)\},\{(\beta,\alpha),(main,\delta)\},\{(\delta,\beta),(main,\delta)\},\{(\beta,\alpha),(\delta,\beta),(main,\delta)\}\}
\end{aligned}
$$

Figure 10: Flattening.

In order to simplify our arguments, we focussed on a sublanguage where (i) fields do not contain futures, (ii) nor actors, and (iii) futures are either returned or synchronized within a method body.

For allowing futures to be stored in fields, and relaxing restriction (i), we would need to track the identities of those futures, since they could be synchronized by any actor who has (direct or indirect) access to them. Thus, the type of an actor must be extended to a tuple containing also the types of all its field. Moreover if we relax also restriction (ii), the tuple is not sufficient anymore, because each of the field can contain in turn an actor with field whose content we want to track. In this case we have to resort to record types (as done in [15]). This would allow us to safely analyse *immutable* fields of any type. For enabling also the field update we would need to track also the effect of each method on the fields of the actors taken as parameter. So method signatures should also record in the output type, how field content has been modified. Parallel modification of the same field would result in a conflict that we must detect.

To remove restriction (iii) and allow pending unsynchronized futures at the end of a method execution, the type system would need a minor modification in the method typing rule, in order to collect also the unsynchronized behavior corresponding to those unsynchronized futures. Then dealing with the new type system will affect mostly the complexity of the analysis, as shown in [15].

References

[1] G. Agha. The structure and semantics of actor languages. In *REX Workshop*, pages 1–59, 1990.

[2] G. Agha, I. Mason, S. Smith, and C. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7:1–72, 1997.

[3] J. Armstrong. Erlang. *Communications of ACM*, 53(9):68–75, 2010.

[4] A. Bouajjani and M. Emmi. Analysis of recursively parallel programs. In *POPL 2012*, pages 203–214, 2012. doi: 10.1145/2103656.2103681.

[5] S. Brandauer, E. Castegren, D. Clarke, K. Fernandez-Reyes, E. Broch-Johnsen, K. Pun, L. T. Tarifa, T. Wrigstad, and A. Yang. Parallel objects for multicores: A glimpse at the parallel language encore. In *SFM*, volume 9104 of *LNCS*, pages 1–56. Springer, 2015.

[6] R. Carlsson and H. Millroth. On cyclic process dependencies and the verification of absence of deadlocks in reactive systems, 1997.

[7] D. Caromel, L. Henrio, and B. P. Serpette. Asynchronous sequential processes. *Inf. Comput.*, 207(4):459–495, 2009.

[8] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 2002.

[9] F. S. de Boer, D. Clarke, and E. Broch Johnsen. A complete guide to the future. In *ESOP*, pages 316–330, 2007.

[10] F. S. de Boer, M. M. Jaghoori, C. Laneve, and G. Zavattaro. Decidability problems for actor systems. *Logical Methods in Computer Science*, 10(4), 2014.

[11] E. Giachino and C. Laneve. A beginner's guide to the deadLock Analysis Model. In *TGC'2012*, volume 8191 of *LNCS*, pages 49–63. Springer-Verlag, 2013.

[12] E. Giachino and C. Laneve. Deadlock detection in linear recursive programs. In *Proceedings of SFM-14:ESM*, volume 8483 of *LNCS*, pages 26–64. Springer-Verlag, 2014.

[13] E. Giachino, C. A. Grazia, C. Laneve, M. Lienhardt, and P. Y. H. Wong. Deadlock analysis of concurrent objects: Theory and practice.

In *Proceedings of IFM 2013*, volume 7940 of *LNCS*, pages 394–411. Springer, 2013.

[14] E. Giachino, N. Kobayashi, and C. Laneve. Deadlock analysis of unbounded process networks. In *Proceedings of CONCUR 2014*, volume 8704, pages 63–77. Springer, 2014.

[15] E. Giachino, C. Laneve, and M. Lienhardt. A framework for deadlock detection in core ABS. *Software and Systems Modeling*, 2015. doi: 10.1007/s10270-014-0444-y. URL `https://hal.inria.fr/hal-01229046`.

[16] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2-3): 202–220, 2009.

[17] L. Henrio, F. Huet, and Z. István. A language for multi-threaded active objects. INRIA Research Report RR-8021, 2012.

[18] C. Hewitt. Procedural embedding of knowledge in planner. In *Proc. the 2nd International Joint Conference on Artificial Intelligence*, pages 167–184, 1971.

[19] E. B. Johnsen, O. Owe, and I. C. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theoretical Computer Science*, 365(1–2):23–66, 2006.

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

[21] N. Kobayashi and C. Laneve. Deadlock analysis of unbounded process networks. *Inf. and Comput. (to appear)*, 2016.

[22] J. Niehren, J. Schwinghammer, and G. Smolka. A Concurrent Lambda Calculus with Futures. *Theoretical Computer Science*, 364(3):338–356, nov 2006.

[23] C. Schröter and J. Esparza. Reachability analysis using net unfoldings. In *Proc. of Workshop of Concurrency, Specification & Programming 2000 (CS&P'2000*. Citeseer, 2000.

A. Proofs of Section 3

A.1 Syntax for runtime typing configuration

In order to type the configurations we use a runtime type system. To this aim we extend the syntax of contracts in Figure 3 and define *extended futures* F and *behavioral type for configuration* K as follows:

$\mathbb{r}$	$::=$	$\Box \mid \alpha$	basic type
$\mathbb{x}$	$::=$	$\mathbb{r}_F \mid \mathbb{r}$	extended type
F	$::=$	$f \mid f^s$	future type
κ	$::=$	$\star \mid \alpha \mid X$	synchronizers
L	$::=$	$\mathtt{0} \mid (\kappa, \alpha) \mid F_\kappa \mid \mathsf{L}+\mathsf{L} \mid \mathsf{L}\,\&\,\mathsf{L}$	behavioural type
K	$::=$	$(\nu\,\overline{\varphi})(\Theta \cdot \mathsf{L}) \mid \mathsf{K}\,\&\,\mathsf{K}$	behavioral type for config.

As regards F, they are introduced for distinguishing two kinds of future names: i) f that has been used in the type system as a static time representation of a future, but it is now used as its runtime representation; ii) f^s now replacing f in its role of static time future (it is typically used to reference a future that is not created yet).

A.2 Typing rules for runtime configurations

The typing rules for the runtime configuration are given in Figures 11 and 12. Except for a few rules (in particular, those in Figures 11 which type the runtime objects of a configuration), all the typing rules have a corresponding one in the type system defined in Section 3.

Additionally, the typing judgments are identical to the corresponding one in the type system, except for some minor differences:

1) the typing environment, that now maps method names to a pair of elements (i.e $\Delta(m) = ((\alpha, \overline{\mathbb{x}}, X) \to \mathbb{r}, \mathsf{K})$) that are respectively the method signature and its behavioral type, is called Δ;
2) the $rt\_unsync(\cdot)$ function on environments Δ is similar to $unsync(\cdot)$ in Section 3, except that it now grabs all f^s and all futures f that were created by the current thread f .

   More precisely we define $\mathtt{Fut}_R(\Delta)$, $\mathtt{AFut}_R(\Delta)$, and $rt\_unsync(\Delta)$ to be the functions

$$\mathtt{Fut}_R(\Delta) \stackrel{def}{=} \{F \mid F \in \mathrm{dom}(\Delta)\}$$

$$\mathtt{AFut}_R(\Delta) \stackrel{def}{=} \{F \in \mathtt{Fut}(\Delta) \mid \Delta(F) = \Delta(F)^\times\}$$

$$rt\_unsync(\Delta) \stackrel{def}{=} \&_{F \in \mathtt{AFut}_R(\Delta)} F_\star,$$

   where $\mathtt{Fut}_R(\Gamma)$ collects all the (static and runtime) futures stored in Δ, $\mathtt{AFut}_R(\Delta)$ collects all the (static and runtime) futures that are not tagged with a $\checkmark$ or $\to$, and $rt\_unsync(\Delta)$ performs the parallel composition of the behavioral types of such not-yet-synchronized method invocations.

A.3 Proof of Theorem 3.4 (Subject Reduction)

Lemma A.1. *$\Delta \vdash_R e : \mathbb{x}$, $\mathsf{L} \triangleright \Delta'$ and $[\![e]\!]_\ell = w$ for some ℓ, imply that $\Delta' \vdash_R w : \mathbb{x}'$, $\mathtt{0} \triangleright \Delta'$ where $\mathbb{x}' = \mathbb{x}$ or $\mathbb{x} = \mathbb{r}_f$ and $\mathbb{x}' = \mathbb{r}$.*

Proof. We can prove it by structural induction on e.

Base Cases:

Case b.1: e can be an integer value, an actor name or variable containing an integer value or an actor name ($e = v$ and $[\![e]\!]_\ell = k$ where $k = \alpha$ or k is an integer value).
By rules TR-SYNC-VAL and TR-VAL we can say that $\Delta \vdash_R e : \mathbb{r}$, $\mathtt{0} \triangleright \Delta$ and again by TR-SYNC-VAL and TR-VAL we can conclude that $\Delta \vdash_R k : \mathbb{r}$, $\mathtt{0} \triangleright \Delta$.

Case b.2: e is a future variable x where $\Delta(x) = \mathbb{r}_F$.
By rules TR-SYNC and TR-VAR we can say that $\Delta \vdash_{\{\alpha, \mathbb{x}\}} e : \Box$, $F_\alpha \,\&\, rt\_unsync(\Delta') \triangleright \Delta'$ where Δ' has the shape $\Delta' = (\Delta[y \mapsto \mathbb{r}]^{\Delta(y) = \mathbb{r}_F})[F \mapsto \Delta(F)^\checkmark]$, by rules TR-SYNC-VAL and TR-VAL we can conclude that using Δ' we can type the evaluation of e saying that $\Delta' \vdash [\![e]\!]_\ell : \Box$, $\mathtt{0} \triangleright \Delta'$.

Induction step: e_1 and e_2 are two expressions such that $[\![e_1]\!]_\ell = k_1$ and $[\![e_2]\!]_\ell = k_2$ where k_1 and k_2 are integer values, by induction hypothesis and the rule TR-EXPRESSION we can say that $\Delta \vdash_R e_1 : \Box$, $\mathsf{L}_1 \triangleright \Delta'$ implies $\Delta' \vdash_R [\![e_1]\!]_\ell : \Box$, $\mathtt{0} \triangleright \Delta'$ and $\Delta' \vdash_R e_2 : \Box$, $\mathsf{L}_2 \triangleright \Delta''$ implies $\Delta'' \vdash_R [\![e_2]\!]_\ell : \Box$, $\mathtt{0} \triangleright \Delta''$.
By rules TR-EXPRESSION and TR-VAL we can infer $\Delta \vdash_R e_1 \oplus e_2 : \Box$, $\mathsf{L}_1 + \mathsf{L}_2 \triangleright \Delta''$ and this implies $\Delta'' \vdash_R k_1 \oplus k_2 : \Box$, $\mathtt{0} + \mathtt{0} \triangleright \Delta''$.

□

Theorem 3.4. (Subject Reduction) *Let $\Delta \vdash_R cn : K$ and $cn \to cn'$. Then there exist Δ', K', and an injective renaming of actor names i such that*

- *$\Delta' \vdash_R cn' : K'$ and*
- *$i(K) \succeq K'$*

Proof. The proof is a case analysis on the reduction rule used in $cn \to cn'$ and we assume that the evaluation of an expression $[\![w]\!]$ always terminates.

Case SERVE**.**

$$\frac{}{\alpha(a, \varnothing, \overline{q} \cup \{p\}) \to \alpha(a, p, \overline{q})}\ \textsc{Serve}$$

Bt hypothesis $\Delta \vdash \alpha(a, \varnothing, \overline{q} \cup \{p\}) : \mathsf{K}$ applying the rule TR-ACTOR there exist $\mathsf{K}_1, \cdots, \mathsf{K}_n$ and K_p such that $\mathsf{K} = (\overset{n}{\underset{i=1}{\&}} \mathsf{K}_i) \,\&\, \mathsf{K}_p$ with the same Δ we can type $\alpha(a, p, \overline{q})$ and we have $\Delta \vdash \alpha(a, p, \overline{q}) : \mathsf{K}_p \,\&\, (\overset{n}{\underset{i=1}{\&}} \mathsf{K}_i)$. By commutativity of $\&$ we can easily prove that $(\overset{n}{\underset{i=1}{\&}} \mathsf{K}_i) \,\&\, \mathsf{K}_p \succeq \mathsf{K}_p \,\&\, (\overset{n}{\underset{i=1}{\&}} \mathsf{K}_i)$.

Case UPDATE**.**

$$\frac{(a + \ell)(x) = f \quad (a + \ell)[x \mapsto w] = a' + \ell'}{\alpha(a, \{\ell \mid s\}, \overline{q})\ f(w) \to \alpha(a', \{\ell' \mid s\}, \overline{q})\ f(w)}\ \textsc{Update}$$

Ex hypothesis we know that $\Delta \vdash \alpha(a, \{\ell | s\}, \overline{q})\ f(w) : \mathsf{K}$, by TR-PARALLEL, TR-ACTOR and TR-PROCESS there exist $\mathsf{K}_1, \cdots, \mathsf{K}_n, \mathsf{L}, \overline{\varphi}$ and Θ such that $\mathsf{K} = (\nu\,\overline{\varphi})(\Theta \cdot \mathsf{L}) \,\&\, (\overset{n}{\underset{i=1}{\&}} \mathsf{K}_i) \,\&\, \mathtt{0}$ accordingly with the restrictions described for our type system we have that $x \in \ell$ then $a' = a$ and $\ell' = \ell[x \mapsto w]$. Given $\Delta \vdash w : \mathbb{x}$ we can choose $\Delta' = \Delta[x \mapsto \mathbb{x}]$ to type $\alpha(a, \{\ell' | s\}, \overline{q})\ f(w)$. By rules TR-PARALLEL, TR-ACTOR and TR-PROCESS we obtain that $\Delta' \vdash \alpha(a, \{\ell[x \mapsto w] | s\}, \overline{q})\ f(w) : \mathsf{K}$. It's trivial to see that $\mathsf{K} \succeq \mathsf{K}$.

$$\frac{\text{(TR-FUTURE)}\quad -}{\Delta \vdash f(v) : \mathsf{0}} \qquad \frac{\text{(TR-ACTOR)}\quad \Delta \vdash^{\alpha} p : \mathsf{K}_1 \qquad \Delta \vdash^{\alpha} \overline{q} : \mathop{\&}_{i=2}^{n} \mathsf{K}_i}{\Delta \vdash \alpha(a, p, \overline{q}) : \mathop{\&}_{i=1}^{n} \mathsf{K}_i}$$

$$\frac{\begin{array}{c}\text{(TR-PROCESS)}\\ \Delta \vdash \mathtt{m} : (\alpha, \overline{\mathbb{x}}, X) \to \mathbb{r} \qquad \Delta(f) = \mathtt{m}(\alpha, \mathbb{x}, X)\\ \Delta + \overline{x : \mathbb{x}} + \mathtt{destiny} : \mathbb{r} + \mathtt{future} : X \vdash^{\alpha}_{\{\alpha, \overline{\mathbb{x}}\}} s : \mathsf{L} \triangleright \Delta'\\ \mathtt{AFut}(\Delta') = \varnothing \qquad \overline{\varphi} = var(\mathsf{L}) \setminus \{\alpha, \overline{\mathbb{x}}\}\\ \Theta_{\mathtt{m}} = [f \mapsto \Delta'(f)^{\times}]^{f \in \mathtt{Fut}(\Delta')} \qquad \mathsf{L}_{\mathtt{m}} = \mathsf{L} \,\&(X, \alpha)\end{array}}{\Delta \vdash^{\alpha} \{destiny \mapsto f, \overline{x \mapsto v}, \overline{y \mapsto v'} \mid s\} : (\nu\, \overline{\varphi})(\Theta_{\mathtt{m}} \cdot \mathsf{L}_{\mathtt{m}})} \qquad \frac{\text{(TR-PARALLEL)}\quad \Delta \vdash cn_1 : \mathsf{K}_1 \qquad \Delta \vdash cn_2 : \mathsf{K}_2}{\Delta \vdash cn_1\ cn_2 : \mathsf{K}_1 \,\&\, \mathsf{K}_2}$$

Figure 11: Typing rules for runtime configurations.

Case NEW**.**

$$\frac{\text{NEW}\quad \overline{w} = [\![\overline{v}]\!]_{a+\ell} \qquad \beta \text{ fresh} \qquad \overline{z} = \mathit{fields}(\mathtt{Act})}{\begin{array}{c}\alpha(a, \{\ell \mid x = \mathtt{new}\ \mathtt{Act}(\overline{v}); s\}, \overline{q})\\ \to \alpha(a, \{\ell \mid x = \beta; s\}, \overline{q})\ \ \beta([\overline{z} \mapsto \overline{w}], \varnothing, \varnothing)\end{array}}$$

Ex hypothesis we know that $\Delta \vdash \alpha(a, \{\ell | x = \mathtt{new}\ \mathtt{Act}(\overline{v}); s\}, \overline{q}) : \mathsf{K}$, by TR-ACTOR, TR-PROCESS, TR-SEQ, TR-NEW and TR-VAL there exist $\mathsf{K}_1, \cdots, \mathsf{K}_n$, L, $\overline{\varphi}$ and Θ such that $\mathsf{K} = (\nu\, \overline{\varphi}, \beta)(\Theta \cdot \mathsf{0} + \mathsf{L}) \,\&\, (\mathop{\&}_{i=1}^{n} \mathsf{K}_i)$.
With the same Δ by TR-PROCESS, TR-ACTOR, TR-PROCESS, TR-SEQ, TR-ASSIGN-VAL and TR-ACT we have: $\Delta \vdash \alpha(a, \{\ell | x = \beta; s\}, \overline{q})\ \beta([z \mapsto w], \varnothing, \varnothing) : \mathsf{K} \,\&\, \mathsf{0} \,\&\, \mathsf{0}$.
It's trivial to see that $\mathsf{K} \succeq \mathsf{K} \,\&\, \mathsf{0} \,\&\, \mathsf{0}$.

Case RETURN**.**

$$\frac{\text{RETURN}\quad w = [\![v]\!]_{a+\ell} \qquad \ell(\mathtt{destiny}) = f}{\alpha(a, \{\ell \mid \mathtt{return}\ v;\}, \overline{q})\ f(\bot) \to \alpha(a, \varnothing, \overline{q})\ f(w)}$$

By hypothesis we know that $\Delta \vdash \alpha(a, \{\ell \mid \mathtt{return}\ v;\}, \overline{q})\ f(\bot) : \mathsf{K}$, applying TR-PARALLEL, TR-ACTOR and TR-PROCESS there exist L, $\mathsf{K}_1, \cdots, \mathsf{K}_n$, $\overline{\varphi}$, X and Θ such that $\mathsf{K} = (\nu\, \overline{\varphi})(\Theta \cdot \mathsf{L} \,\&(X, \alpha)) \,\&\, (\mathop{\&}_{i=1}^{n} \mathsf{K}_i) \,\&\, \mathsf{0}$ we can distinguish two cases:
Case 1: $\Delta(v) = \mathbb{r}$, by rule TR-RETURN-VAL we have $\mathsf{L} = \mathsf{0}$
Case 2: $\Delta(v) = \mathbb{r}_f$ by rule TR-RETURN we have $\mathsf{L} = f_X \,\&\, rt\_unsync((\Delta \setminus f))$
In both cases, with the same Δ by rules TR-PARALLEL, TR-ACTOR and TR-PROCESS we can say that $\Delta \vdash \alpha(a, \varnothing, \overline{q})\ f(w) : \mathsf{0} \,\&\, (\mathop{\&}_{i=1}^{n} \mathsf{K}_i) \,\&\, \mathsf{0}$.
It's trivial to see that $(\nu\, \overline{\varphi})(\Theta \cdot \mathsf{0} \,\&(X, \alpha)) \,\&\, (\mathop{\&}_{i=1}^{n} \mathsf{K}_i) \,\&\, \mathsf{0} \succeq \mathsf{0} \,\&\, (\mathop{\&}_{i=1}^{n} \mathsf{K}_i) \,\&\, \mathsf{0}$ or
$(\nu\, \overline{\varphi})(\Theta \cdot f_X \,\&\, rt\_unsync((\Delta \setminus f)) \,\&(X, \alpha)) \,\&\, (\mathop{\&}_{i=1}^{n} \mathsf{K}_i) \,\&\, \mathsf{0} \succeq \mathsf{0} \,\&\, (\mathop{\&}_{i=1}^{n} \mathsf{K}_i) \,\&\, \mathsf{0}$.

Case IF-TRUE**.**

$$\frac{\text{IF-TRUE}\quad [\![e]\!]_{a+\ell} \neq 0}{\begin{array}{c}\alpha(a, \{\ell \mid \mathtt{if}\ e\ \{\, s_1 \,\}\ \mathtt{else}\ \{\, s_2 \,\};\ s\}, \overline{q})\\ \to \alpha(a, \{\ell \mid s_1\ ;\ s\}, \overline{q})\end{array}}$$

Ex hypothesis we know that $\Delta \vdash \alpha(a, \{\ell \mid \mathtt{if}\ v\ \{\, s_1 \,\}\ \mathtt{else}\ \{\, s_2 \,\};\ s\}, \overline{q}) : \mathsf{K}$, by TR-PARALLEL, TR-ACTOR, TR-PROCESS and TR-SEQ there exist L, L_s $\mathsf{K}_1, \cdots, \mathsf{K}_n$, $\overline{\varphi}$, X and Θ such that $\mathsf{K} = (\nu\, \overline{\varphi})\big(\Theta \cdot (\mathsf{L} + \mathsf{L}_s) \,\&(X, \alpha)\big) \,\&\, (\mathop{\&}_{i=1}^{n} \mathsf{K}_i)$.
Let $\Delta_S = \overline{x : \mathbb{x}} + \mathtt{destiny} : \mathbb{r} + \mathtt{future} : X$ by role TR-IF we have that:
$\Delta + \Delta_S \vdash^{\alpha}_R e : \mathsf{L}_e \triangleright \Delta' + \Delta_S$,
$\Delta' + \Delta_S \vdash^{\alpha}_R s_1 : \mathsf{L}_1 \triangleright \Delta_1 + \Delta_S$,
$\Delta_1 + \Delta_S \vdash^{\alpha}_R s_2 : \mathsf{L}_2 \triangleright \Delta_2 + \Delta_S$ and
$\Delta + \Delta_S \vdash^{\alpha}_R \mathtt{if}\ e\ \{\, s_1 \,\}\ \mathtt{else}\ \{\, s_2 \,\} : \mathsf{L}_e + \mathsf{L}_1 + \mathsf{L}_2 \triangleright \Delta'' + \Delta_S$ where $\Delta'' = \mathtt{Merge}(\Delta_1, \Delta_2) \cup \Delta_1|_{\mathtt{Fut}(\Delta_1) \setminus \mathtt{Fut}(\Delta)} \cup \Delta_2|_{\mathtt{Fut}(\Delta_2) \setminus \mathtt{Fut}(\Delta)}$, then $\mathsf{L} = \mathsf{L}_e + \mathsf{L}_1 + \mathsf{L}_2$.
Thanks to the Lemma A.1 we can say that $\Delta + \Delta_S \vdash e : \mathbb{x}$, $\mathsf{L}_e \triangleright \Delta' + \Delta_S$ implies that $\Delta' + \Delta_S \vdash w : \square$, $\mathsf{0} \triangleright \Delta' + \Delta_S$ where $w = [\![e]\!]_{\ell}$.
Now we can use Δ' and the rules TR-PARALLEL, TR-ACTOR, TR-PROCESS and TR-SEQ to say that $\Delta' \vdash \alpha(a, \{\ell | s_1; s\}, \overline{q}) : (\nu\, \overline{\varphi})\big(\Theta \cdot (\mathsf{L}_1 + \mathsf{L}_s) \,\&(X, \alpha)\big) \,\&\, (\mathop{\&}_{i=1}^{n} \mathsf{K}_i)$.
It's trivial to prove by the rules LS-RBEHAVIOR and LS-PLUS that

$$(\nu\, \overline{\varphi})\big(\Theta \cdot (\mathsf{L}_e + \mathsf{L}_1 + \mathsf{L}_2 + \mathsf{L}_s) \,\&(X, \alpha)\big) \,\&\, (\mathop{\&}_{i=1}^{n} \mathsf{K}_i) \succeq (\nu\, \overline{\varphi})\big(\Theta \cdot (\mathsf{L}_1 + \mathsf{L}_s) \,\&(X, \alpha)\big) \,\&\, (\mathop{\&}_{i=1}^{n} \mathsf{K}_i).$$

Case INVK**.**

$$\frac{\text{INVK}\quad [\![v]\!]_{a+\ell} = \beta \qquad [\![\overline{v}]\!]_{a+\ell} = \overline{w} \qquad \beta \neq \alpha \qquad f \text{ fresh} \qquad \mathrm{bind}(\beta, m, \overline{w}, f) = p''}{\begin{array}{c}\alpha(a, \{\ell \mid x = v.\mathtt{m}(\overline{v}); s\}, \overline{q})\ \beta(a', p', \overline{q'})\\ \to \alpha(a, \{\ell \mid x = f; s\}, \overline{q})\ \beta(a', p', \overline{q'} \cup \{p''\})\ f(\bot)\end{array}}$$

By hypothesis we know that

$$\Delta \vdash \alpha(a, \{\ell | x = v.\mathtt{m}(\overline{v}); s\}, \overline{q})\ \beta(a', p', q') : \mathsf{K}$$

, applying TR-PARALLEL, TR-ACTOR, TR-PROCESS and TR-SEQ there exist L_s, $\mathsf{K}_1, \cdots, \mathsf{K}_n$, K'_p and $\mathsf{K}'_1, \cdots, \mathsf{K}'_n$, f, $\overline{\varphi}$, X and Θ such that

$$\mathsf{K} = (\nu\, \overline{\varphi}, f)\big(\Theta \cdot (\mathsf{L} + \mathsf{L}_s) \,\&(X, \alpha)\big) \,\&\, (\mathop{\&}_{i=1}^{n} \mathsf{K}_i) \,\&\, \mathsf{K}'_p \,\&\, (\mathop{\&}_{i=1}^{n} \mathsf{K}'_i).$$

Let $\Delta_S = \overline{x : \mathbb{x}} + \mathtt{destiny} : \mathbb{r} + \mathtt{future} : X$ by role TR-SYNC we have that
$\Delta + \Delta_S \vdash^{\alpha}_R v : \beta$, $\mathsf{L}' \triangleright \Delta' + \Delta_S$ and by role TR-INVK we have
$\Delta + \Delta_S \vdash^{\alpha}_R v.\mathtt{m}(\overline{v}) : \mathbb{r}_{f^s}$, $\mathsf{L}' + f^s_\star \,\&\, rt\_unsync(\Delta') \triangleright \Delta'' + \Delta_S$ than $\mathsf{L} = \mathsf{L}' + f^s_\star \,\&\, rt\_unsync(\Delta')$. Using Δ''

expressions and addresses $\Delta \vdash v : \mathbb{x}$

$$\text{(TR-VAR)}\ \frac{\Delta(x) = \mathbb{x}}{\Delta \vdash^{\alpha} x : \mathbb{x}} \qquad \text{(TR-VAL)}\ \frac{v \quad \textit{integer-value or } \texttt{null}}{\Delta \vdash^{\alpha} v : \mathbb{r}} \qquad \text{(TR-ACT)}\ \frac{-}{\Delta \vdash^{\alpha} \beta : \beta} \qquad \text{(TR-FUT)}\ \frac{\Delta(f) = \lambda X.\mathtt{m}(\beta, \overline{\mathbb{x}}, X) \quad \Delta \vdash \mathtt{m} : (\beta, \overline{\mathbb{x}}, X) \to \mathbb{r}}{\Delta \vdash^{\alpha} f : \mathbb{r}_f}$$

$$\text{(TR-METHOD-SIGN)}\ \frac{\Delta(\mathtt{m}) = (\nu\,\overline{\varphi})(\Theta_{\mathtt{m}} \cdot \mathsf{L}_{\mathtt{m}}) \quad \sigma \text{ renaming} \quad \sigma(\square) = \square \quad \mathbb{r} \notin \{\square, \alpha, \overline{\mathbb{x}}\} \implies \sigma(\mathbb{r}) \text{ fresh}}{\Delta \vdash m : (\sigma(\alpha), \sigma(\overline{\mathbb{x}}), X) \to \sigma(\mathbb{r})}$$

expressions with side effects $\Delta \vdash^{\alpha}_{R} e : \mathbb{x}\ ,\ \mathsf{L} \triangleright \Delta'$

$$\text{(TR-SYNC)}\ \frac{\Delta \vdash v : \mathbb{r}_F \quad \Delta(\mathtt{this}) = \alpha \quad \Delta' = (\Delta[y \mapsto \mathbb{r}]^{\Delta(y)=\mathbb{r}_F})[F \mapsto \Delta(F)^{\checkmark}]}{\Delta \vdash^{\alpha}_{R} v : \mathbb{r}\ ,\ F_{\alpha}\,\&\,rt\_unsync(\Delta') \triangleright \Delta'} \qquad \text{(TR-SYNC-VAL)}\ \frac{\Delta \vdash^{\alpha} v : \mathbb{r}}{\Delta \vdash^{\alpha}_{R} v : \mathbb{r}\ ,\ 0 \triangleright \Delta} \qquad \text{(TR-EXPRESSION)}\ \frac{\Delta \vdash^{\alpha}_{R} e : \square\ ,\ \mathsf{L}_1 \triangleright \Delta' \quad \Delta' \vdash^{\alpha}_{R} e' : \square\ ,\ \mathsf{L}_2 \triangleright \Delta''}{\Delta \vdash^{\alpha}_{R} e \oplus e' : \square\ ,\ \mathsf{L}_1 + \mathsf{L}_2 \triangleright \Delta''}$$

$$\text{(TR-NEW)}\ \frac{\alpha \text{ fresh} \quad \Delta \vdash \overline{v} : \overline{\square}}{\Delta \vdash^{\alpha}_{R} \mathtt{new\ Act}(\overline{v}) : \alpha\ ,\ 0 \triangleright \Delta} \qquad \text{(TR-INVK)}\ \frac{\Delta \vdash^{\alpha}_{R} v : \alpha\ ,\ \mathsf{L} \triangleright \Delta' \quad \Delta' \vdash \overline{v} : \overline{\mathbb{x}} \quad \Delta' \vdash \mathtt{m} : (\alpha, \overline{\mathbb{x}}, X) \to \mathbb{r} \quad f^s \text{ fresh} \quad \Delta'' = \Delta'[f^s \mapsto \lambda X.\mathtt{m}(\alpha, \overline{\mathbb{x}}, X)]}{\Delta \vdash^{\alpha}_{R} v.\mathtt{m}(\overline{v}) : \mathbb{r}_{f^s}\ ,\ \mathsf{L} + f^s_{\star}\,\&\,rt\_unsync(\Delta') \triangleright \Delta''}$$

statements $\Delta \vdash^{\alpha}_{R} s : \mathsf{L} \triangleright \Delta'$

$$\text{(TR-ASSIGN-FIELD)}\ \frac{x \in \mathit{fields}(\mathtt{Act}) \setminus \mathrm{dom}(\Delta) \quad \Delta \vdash^{\alpha} v : \square}{\Delta \vdash^{\alpha}_{R} x = v : 0 \triangleright \Delta} \qquad \text{(TR-ASSIGN-VAL)}\ \frac{x \notin \mathit{fields}(\mathtt{Act}) \setminus \mathrm{dom}(\Delta) \quad \Delta \vdash^{\alpha} w : \mathbb{x}}{\Delta \vdash^{\alpha}_{R} x = w : 0 \triangleright \Delta[x \mapsto \mathbb{x}]} \qquad \text{(TR-ASSIGN-EXP)}\ \frac{x \notin \mathit{fields}(\mathtt{Act}) \setminus \mathrm{dom}(\Delta) \quad z \text{ is not a value } v \quad \Delta \vdash^{\alpha}_{R} z : \mathbb{x}\ ,\ \mathsf{L} \triangleright \Delta'}{\Delta \vdash^{\alpha}_{R} x = z : \mathsf{L} \triangleright \Delta'[x \mapsto \mathbb{x}]}$$

$$\text{(TR-SEQ)}\ \frac{\Delta \vdash^{\alpha}_{R} s_1 : \mathsf{L}_1 \triangleright \Delta_1 \quad \Delta_1 \vdash^{\alpha}_{R} s_2 : \mathsf{L}_2 \triangleright \Delta_2}{\Delta \vdash^{\alpha}_{R} s_1; s_2 : \mathsf{L}_1 + \mathsf{L}_2 \triangleright \Delta_2} \qquad \text{(TR-RETURN)}\ \frac{\Delta(\mathtt{destiny}) = \mathbb{r} \quad \Delta(\mathtt{future}) = X \quad \Delta \vdash v : \mathbb{r}'_{f'} \quad \mathbb{r} \in R \implies \mathbb{r} = \mathbb{r}' \quad \Delta' = \Delta[f' \mapsto \Delta(f')^{\checkmark}]}{\Delta \vdash^{\alpha}_{R} \mathtt{return}\ v : f'_X\,\&\,rt\_unsync(\Delta') \triangleright \Delta'} \qquad \text{(TR-RETURN-VAL)}\ \frac{\Delta(\mathtt{destiny}) = \mathbb{r} \quad \Delta \vdash v : \mathbb{r}' \quad \mathbb{r} \in R \implies \mathbb{r} = \mathbb{r}'}{\Delta \vdash^{\alpha}_{R} \mathtt{return}\ v : 0 \triangleright \Delta}$$

$$\text{(TR-SKIP)}\ \Delta \vdash^{\alpha}_{R} \mathtt{skip} : 0 \triangleright \Delta \qquad \text{(T-IF)}\ \frac{\Delta \vdash^{\alpha}_{R} e : \square\ ,\ \mathsf{L} \triangleright \Delta' \quad \Delta' \vdash^{\alpha}_{R} s_1 : \mathsf{L}_1 \triangleright \Delta_1 \quad \Delta' \vdash^{\alpha}_{R} s_2 : \mathsf{L}_2 \triangleright \Delta_2 \quad (\mathtt{AFut}(\Delta_1) \cup \mathtt{AFut}(\Delta_2)) \setminus \mathtt{AFut}(\Delta') = \varnothing \quad \Delta'' = \mathtt{Merge}(\Delta_1, \Delta_2) \cup \Delta_1|_{\mathrm{Fut}(\Delta_1)\setminus\mathrm{Fut}(\Delta)} \cup \Delta_2|_{\mathrm{Fut}(\Delta_2)\setminus\mathrm{Fut}(\Delta)}}{\Delta \vdash^{\alpha}_{R} \mathtt{if}\ e\ \{\ s_1\ \}\ \mathtt{else}\ \{\ s_2\ \} : \mathsf{L} + \mathsf{L}_1 + \mathsf{L}_2 \triangleright \Delta''}$$

Figure 12: Typing rules at runtime for expressions, expressions with side-effects and statements.

by rules TR-PARALLEL, TR-ACTOR, TR-PROCESS, TR-SEQ and TR-FUT we finally obtain that $\Delta'' \vdash \alpha(a, \{\ell \mid x = f; s\}, \overline{q})\ \beta(a', p', \overline{q'} \cup \{p''\})\ f(\bot) : \mathsf{K}'$ where $\mathsf{K}' = (\nu\,\overline{\varphi}, f)\big(\Theta \cdot (0 + \mathsf{L}_s)\,\&(X, \alpha)\big)\,\&\,(\overset{n}{\underset{i=1}{\&}} \mathsf{K}_i)\,\&\,\mathsf{K}'_p\,\&\,(\overset{n}{\underset{i=1}{\&}} \mathsf{K}'_i)\,\&\,\mathsf{K}_{p''}$.

where $\mathsf{L}_{p''} = (\nu\,\overline{\varphi'})(\Theta_{\mathtt{m}} \cdot \mathsf{L}_{\mathtt{m}})$ is the behavioral type of the method $\mathtt{m}$ instantiated with β and $\overline{\mathbb{x}}$.

Case ASSIGN**.**

$$\text{ASSIGN}\ \frac{x \in \mathrm{dom}(a + \ell) \quad w = [\![e]\!]_{a+\ell} \quad (a + \ell)[x \mapsto w] = a' + \ell'}{\alpha(a, \{\ell \mid x = e; s\}, \overline{q}) \to \alpha(a', \{\ell' \mid s\}, \overline{q})}$$

Ex hypothesis we know that applying TR-ACTOR, TR-PROCESS, TR-SEQ and TR-ASSIGN-EXP we have that there exist L_e, L_s, $\mathsf{K}_1, \cdots, \mathsf{K}_n$, $\overline{\varphi}$, X and Θ such that
$\Delta \vdash \alpha(a, \{\ell \mid x = e; s\}, \overline{q}) : (\nu\,\overline{\varphi})\big(\Theta \cdot (\mathsf{L}_e + \mathsf{L}_s)\,\&(X, \alpha)\big)\,\&\,(\overset{n}{\underset{i=1}{\&}} \mathsf{K}_i)$.
Let $\Delta_S = \overline{x : \mathbb{x}} + \mathtt{destiny} : \mathbb{r} + \mathtt{future} : X$, by hypothesis we can also know that $\Delta + \Delta_S \vdash e : \mathbb{x}\ ,\ \mathsf{L}_e \triangleright \Delta' + \Delta_S$ Now thanks to the Lemma A.1 we know that $\Delta + \Delta_S \vdash e : \mathbb{x}\ ,\ \mathsf{L}_e \triangleright \Delta' + \Delta_S$ implies $\Delta' + \Delta_S \vdash w : \square\ ,\ 0 \triangleright \Delta' + \Delta_S$ where $w = [\![e]\!]_{\ell}$, and we can finally conclude that by rules TR-ACTOR and TR-PROCESS $\Delta' \vdash \alpha(a, \{\ell' \mid s\}, \overline{q}) : (\nu\,\overline{\varphi})\big(\Theta \cdot \mathsf{L}_s\,\&(X, \alpha)\big)\,\&\,(\overset{n}{\underset{i=1}{\&}} \mathsf{K}_i)$.
It's trivial prove that by rules LS-GLOBAL and LS-PLUS

$$(\nu\,\overline{\varphi})\big(\Theta \cdot (\mathsf{L}_e + \mathsf{L}_s)\,\&(X, \alpha)\big)\,\&\,(\overset{n}{\underset{i=1}{\&}} \mathsf{K}_i) \succeq (\nu\,\overline{\varphi})\big(\Theta \cdot \mathsf{L}_s\,\&(X, \alpha)\big)\,\&\,(\overset{n}{\underset{i=1}{\&}} \mathsf{K}_i).$$

□

B. Proofs of Section 4

Proposition B.1. *Let* $\mathtt{m}(\alpha, \overline{\mathbb{x}}, X) = (\nu\,\overline{\varphi})(\Theta \cdot \mathsf{L}_{\mathtt{m}}) \in \mathcal{L}$.
(i) for every k, $I^{(k)}(\mathtt{m})$ *is an element in the lattice of* $\mathcal{R}_A$.

(ii) for every k, $I^{(k)}(\mathtt{m}) \Subset I^{(k+1)}(\mathtt{m})$.

Proof. (i) is immediate by definition. To see (ii) , observe that $I(\Theta \cdot \mathsf{L})$ is monotonic on I (i.e., $I(\mathtt{m}) \Subset I'(\mathtt{m})$ for every $\mathtt{m}$ implies $I(\Theta \cdot \mathsf{L}) \Subset I'(\Theta \cdot \mathsf{L})$), which follows by a straightforward structural induction on L. Then, a standard induction on k gives $I^{(k)}(\mathtt{m}) \Subset I^{(k+1)}(\mathtt{m})$. □

Since, for every k, $I^{(k)}(\mathtt{m}_i)$ ranges over a finite lattice, by the fixpoint theory [8], there exists m such that $I^{(m)}$ is a fixpoint, namely $I^{(m)} \approx I^{(m+1)}$ where $\approx$ is the equivalence relation induced by $\Subset$. In the following, we let I, called the *interpretation function* (of a behavioral type), be the least fixpoint $I^{(m)}$.

The following three lemmas are preparatory to the theorem of correctness and completeness of our algorithm Theorem 5.1. They establish a relation between the circularities of the approximants $I^{(k)}(\Theta \cdot \mathsf{L})$ - Lemma B.3 and, whenever $\Theta \cdot \mathsf{L} \to \Theta \cdot \mathsf{L}'$, between circularities of $I(\Theta \cdot \mathsf{L})$ and $I(\Theta \cdot \mathsf{L}')$ - Lemma B.4.

Lemma B.2. *Let $\mathcal{C}$ be a context, Θ be a future environment, L be a behavioral type and $I(\cdot)$ be a* flattening. *Then we have:*

1) *$I(\Theta \cdot \mathcal{C}[\,\mathsf{L}\,])$ has a circularity if and only if $\mathcal{C}[\,\mathtt{R}\,]$ has a circularity, for some $\mathtt{R} \in I(\Theta \cdot \mathsf{L})$.*
2) *Let $\mathtt{R}$ be a binary relation on names and $\overline{a}$ be the names in both $\mathcal{C}$ and $\mathtt{R}$. Then $I(\Theta \cdot \mathcal{C}[\,\mathsf{L}\,])$ has a circularity if and only if $I(\Theta \cdot \mathcal{C}[\,\mathtt{proj}_{\overline{a}}(R^+)\,])$ has a circularity.*

Proof. Both the properties follow almost immediately from the definitions. To see 1, note that it follows by a straightforward induction on $\mathcal{C}$ that $\mathtt{R} \in I(\Theta \cdot \mathcal{C}[\,\mathsf{L}\,])$ if and only if $\mathtt{R} \in I(\Theta \cdot \mathcal{C}[\,\mathtt{R}\,])$ for some $\mathtt{R} \in I(\Theta \cdot \mathsf{L})$. □

Lemma B.3. *Let*

$$\begin{array}{rcl} \big(\mathtt{m}_1(\alpha_1, \overline{\mathtt{x}}_1, X_1) & = & (\nu\, \overline{\varphi}_1)(\Theta_1 \cdot \mathsf{L}_1), \\ \cdots, & & \\ \mathtt{m}_n(\alpha_n, \overline{\mathtt{x}}_n, X_n) & = & (\nu\, \overline{\varphi}_n)(\Theta_n \cdot \mathsf{L}_n), \mathsf{L}\big) \end{array}$$

be a behavioral type program and let $\Theta \cdot \mathcal{C}[\,f^{i_1}_{\kappa_1}\,] \cdots [\,f^{i_m}_{\kappa_m}\,] \to^m \Theta' \cdot \mathcal{C}[\,\mathsf{L}'_{i_1}\,] \cdots [\mathsf{L}'_{i_m}]$ where $\mathcal{C}[\,\cdot\,] \cdots [\,\cdot\,]$ is a multiple context without function invocations, $\Theta(f^{i_j}) = \lambda X'_j.\mathtt{m}_{i_j}(\alpha'_j, \overline{\mathtt{x}'}_j, X'_j)$, $\Theta' = \bigcup_{j=1}^{m} \Theta_j[{}^{\overline{\varphi'_j}}/_{\overline{\varphi_{i_j}}}][{}^{\alpha'_j, \overline{\mathtt{x}'}_j, \kappa_j}/_{\alpha_{i_j}, \overline{\mathtt{x}}_{i_j}, X_{i_j}}] \cup \Theta$ and $\mathsf{L}'_{i_j} = \mathsf{L}_{i_j}[{}^{\overline{\varphi'_j}}/_{\overline{\varphi_{i_j}}}][{}^{\alpha'_j, \overline{\mathtt{x}'}_j, \kappa_j}/_{\alpha_{i_j}, \overline{\mathtt{x}}_{i_j}, X_{i_j}}]$.

Then, the following two properties are equivalent:

(1) $I^{(k+1)}(\Theta \cdot \mathcal{C}[\,f^{i_1}_{\kappa_1}\,] \cdots [f^{i_m}_{\kappa_m}])$ has a circularity,

(2) $I^{(k)}(\Theta' \cdot \mathcal{C}[\,\mathsf{L}'_{i_1}\,] \cdots [\mathsf{L}'_{i_m}])$ has a circularity.

Proof. To show the implication 1 ⇒ 2, suppose that

$$I^{(k+1)}(\Theta \cdot \mathcal{C}[\,f^{i_1}_{\kappa_1}\,] \cdots [f^{i_m}_{\kappa_m}])$$

has a circularity. By repeated applications of Lemma B.2(1), there exists $\mathtt{R}_j \in I^{(k+1)}(\Theta \cdot f^{i_j}_{\kappa_j})$ with $1 \le j \le m$ such that $I^{(k+1)}(\Theta \cdot \mathcal{C}[\mathtt{R}_1\,] \cdots [\mathtt{R}_m])$ has a circularity. By the definition of $I^{(k+1)}(\Theta \cdot f^{i_j}_{\kappa_j})$ and $I^{(k+1)}(\mathtt{m}_i)$, where $\Theta(f) = \lambda X.\mathtt{m}_i(\alpha, \overline{\mathtt{x}}, X)$

$$\mathtt{R}_j = \mathtt{proj}_{\overline{a}}(R^+)[{}^{\alpha'_j, \overline{\mathtt{x}'}_j, \kappa_j}/_{\alpha_{i_j}, \overline{\mathtt{x}}_{i_j}, X_{i_j}}]$$

with $\mathtt{R}'_j \in I^{(k)}(\mathsf{L}_{i_j})$. This implies that

$$I^{(k+1)}(\mathcal{C}[\,\mathtt{proj}_{\overline{a'_1}}((\mathtt{R}'_1[{}^{\alpha'_1, \overline{\mathtt{x}'}_1, \kappa_1}/_{\alpha_1, \overline{\mathtt{x}}_1, X_1}])^+)\,] \cdots [\mathtt{proj}_{\overline{a'_m}}((\mathtt{R}'_m[{}^{\alpha'_m, \overline{\mathtt{x}'}_m, \kappa_m}/_{\alpha_m, \overline{\mathtt{x}}_m, X_m}])^+)])$$

also has a circularity. By repeated applications of Lemma B.2(2),

$$I^{(k+1)}(\mathcal{C}[\,\mathtt{R}'_1[{}^{\alpha'_1, \overline{\mathtt{x}'}_1, \kappa_1}/_{\alpha_1, \overline{\mathtt{x}}_1, X_1}]\,] \cdots [\mathtt{R}'_m[{}^{\alpha'_m, \overline{\mathtt{x}'}_m, \kappa_m}/_{\alpha_m, \overline{\mathtt{x}}_m, X_m}]])$$

has also a circularity. Since $\mathcal{C}$ contains no function invocations,

$$I^{(k)}(\mathcal{C}[\,\mathtt{R}'_1[{}^{\alpha'_1, \overline{\mathtt{x}'}_1, \kappa_1}/_{\alpha_1, \overline{\mathtt{x}}_1, X_1}]\,] \cdots [\mathtt{R}'_m[{}^{\alpha'_m, \overline{\mathtt{x}'}_m, \kappa_m}/_{\alpha_m, \overline{\mathtt{x}}_m, X_m}]])$$

has a circularity, and by repeated applications of Lemma B.2(1), $I^{(k)}(\Theta' \cdot \mathcal{C}[\,\mathsf{L}'_{i_1}\,] \cdots [\mathsf{L}'_{i_m}])$ also has a circularity.

The convert is similar. □

Lemma B.4. *Let $(\mathcal{L}, \mathsf{L})$ be a behavioral type program and $\Theta \cdot \mathcal{C}[\,f_\kappa\,] \to \Theta' \cdot \mathcal{C}[\,\mathsf{L}'\,]$, where $\Theta(f) = \lambda X.\mathtt{m}(\alpha', \overline{\mathtt{x}}', X)$, $\mathsf{L}' = \mathsf{L}_{\mathtt{m}}[{}^{\overline{\varphi}'}/_{\overline{\varphi}}][{}^{\alpha', \overline{\mathtt{x}}', \kappa}/_{\alpha, \overline{\mathtt{x}}, X}]$ and $\Theta' = \Theta \cup \Theta_{\mathtt{m}}[{}^{\overline{\varphi}'}/_{\overline{\varphi}}][{}^{\alpha', \overline{\mathtt{x}}', \kappa}/_{\alpha, \overline{\mathtt{x}}, X}]$.*

The following two properties are equivalent:

(1) $I(\Theta \cdot \mathcal{C}[\,f_\kappa\,])$ has a circularity,

(2) $I(\Theta' \cdot \mathcal{C}[\,\mathsf{L}'\,])$ has a circularity.

Proof. To show the implication 1 ⇒ 2, suppose that $I(\Theta \cdot \mathcal{C}[\,f_\kappa\,])$ has a circularity. Then by Lemma B.2(1), there exists $\mathtt{R} \in I(\Theta \cdot f_\kappa)$ such that $I(\Theta \cdot \mathcal{C}[\mathtt{R}\,])$ has a circolarity. By definition of I, there exist k such that $\mathtt{R} \in I^{(k+1)}(\Theta \cdot f_\kappa)$. Thus, by a reasoning similar to the one in Lemma B.3, there exists

$$\mathtt{R}' \in I^{(k)}(\Theta \cdot \mathsf{L}') \Subset I(\Theta \cdot \mathsf{L}')$$

such that $I(\mathtt{R}')$ has a circularity. by Lemma B.2(1), therefore, $I(\Theta \cdot \mathsf{L}')$ has a circularity.

The converse is similar. □

The following theorem states the correctness and completeness of our algorithm. Similarly to [14], there is a relation between the circularities of the set $I^{(k)}(\Theta \cdot \mathsf{L})$ and, whenever $\Theta \cdot \mathsf{L} \to \Theta' \cdot \mathsf{L}'$, between the circularities of $I^{(k)}(\Theta \cdot \mathsf{L})$ and of $I^{(k)}(\Theta' \cdot \mathsf{L}')$. Proofs are omitted because they are similar to those of [14].

Theorem 4.6. *A behavioural type program $(\mathcal{L}, \Theta \cdot \mathsf{L})$ has a circularity if and only if $I_{\mathcal{L}}(\Theta \cdot \mathsf{L})$ has a circularity.*

Proof. (*If* direction) By definition, $(\mathcal{L}, \mathsf{L})$ has a circularity if there is $\Theta \cdot \mathsf{L} \to^* \Theta' \cdot \mathsf{L}'$ such that $I^{\perp}(\Theta' \cdot \mathsf{L}')$ has a circularity. By induction on the lenght of $\mathsf{L} \to^* \mathsf{L}'$. When the lenght is 0 then $I^{\perp}(\Theta' \cdot \mathsf{L}')$ has a circularity implies $I(\Theta' \cdot \mathsf{L}')$ has a circularity (by $I^{\perp}(\Theta' \cdot \mathsf{L}') = I^{(0)}(\Theta' \cdot \mathsf{L}')$ and Lemma B.1(2)). Assume $\Theta \cdot \mathsf{L} \to^* \Theta' \cdot \mathsf{L}'$ be equale to $\Theta \cdot \mathsf{L} \to \Theta'' \cdot \mathsf{L}'' \to^* \Theta' \cdot \mathsf{L}'$. By inductive hypothesis, we assume that the theorem holds on the computation $\Theta'' \cdot \mathsf{L}'' \to^* \Theta' \cdot \mathsf{L}'$. Then, by Lemma B.4, if $I(\Theta'' \cdot \mathsf{L}'')$ has a circularity then $I(\Theta \cdot \mathsf{L})$ has a circularity. Therefore the theorem.

(*Only-if* direction) We demostrate that, if $I(\Theta \cdot \mathsf{L})$ has a circularity then there is $\Theta \cdot \mathsf{L} \to^* \Theta' \cdot \mathsf{L}'$ such that $I^{\perp}(\Theta' \cdot \mathsf{L}')$ has a circularity. Let m be the least natural number such that $I = I^{(m)}$. Let $\mathsf{L} = \mathcal{C}[\,f^{i_1}_{\kappa_1}\,] \cdots [f^{i_n}_{\kappa_n}]$ where $\Theta(f^{i_j}) = \mathtt{m}_{i_j}(\alpha_j, \overline{\mathtt{x}}_j, X_j)$, such that $\mathcal{C}[\,\,] \cdots [\,\,]$ does not contain function invocations. Then

$$\Theta \cdot \mathsf{L} \to^n \Theta \cdot \mathcal{C}[\,\mathsf{L}_{i_1}\,] \cdots [\mathsf{L}_{i_n}] = \Theta'' \cdot \mathsf{L}''$$

Additionally, by Lemma B.3, $I^{(m-1)}(\Theta'' \cdot \mathsf{L}'')$ has a circularity because $I^{(m)}(\Theta' \cdot \mathsf{L}')$ has a circularity. Now, we reapply the same argument to $\Theta'' \cdot \mathsf{L}''$ since $I^{(m-1)}(\Theta'' \cdot \mathsf{L}'')$ has a circularity. After m steps we get $\Theta' \cdot \mathsf{L}'$ such that $I^{(0)}(\Theta' \cdot \mathsf{L}') = I^{\perp}(\Theta' \cdot \mathsf{L}')$ has a circularity. □