Science of Computer Programming 181 (2019) 64–81

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

A lightweight deadlock analysis for programs with threads and reentrant locks

Cosimo Laneve [1]

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

ARTICLE INFO

Article history:
Received 5 October 2018
Received in revised form 11 June 2019
Accepted 14 June 2019
Available online 21 June 2019

Keywords:
Deadlock analysis
Threads and reentrant locks
Lams
Circularities
Static semantics

ABSTRACT

Deadlock analysis of multi-threaded programs with reentrant locks is complex because these programs may have infinitely many states. We define a simple calculus featuring recursion, threads and synchronizations that guarantee exclusive access to objects. We detect deadlocks by associating an abstract model to programs – the *extended lam* model – and we define an algorithm for verifying that a problematic object dependency (e.g. a *circularity*) between threads will not be manifested.

The analysis is lightweight because the deadlock detection problem is fully reduced to the corresponding one in lams (without using other models). In fact, the technique is intended to be an effective tool for the deadlock analysis of programming languages by defining ad-hoc extraction processes. We demonstrate this effectivity by applying our analysis to a core calculus featuring shared objects, threads and `Java`-like synchronization primitives. We also discuss a prototype verifier, called `JaDA`, that covers several features of `Java` and deliver initial assessments of the tool.

1. Introduction

Threads and locks are a common model of concurrent programming that is nowadays widely used by the mainstream programming languages (`Java`, `C#`. `C++`, `Objective C`, etc.). In these languages, deadlocks are flaws that occur when two or more threads are blocked because each one is attempting to acquire an object's lock held by another one. As an example, consider the following method

```
setTable(C x,C y,int n) = (new C z)(   if (n=0) then sync(y){ sync(x){ 0 } }
                                       else fork{ sync(x){ sync(z){ 0 } } }
                                       z.setTable(z,y,n-1)
                                   )
```

where `new C z` creates a new object of class `C`, `fork{`P`}` Q creates a new thread whose body is P and runs it in parallel with the continuation Q, and `sync(x){ P }` is the operation locking the object `x`, performing P, and releasing `x` when P terminates. The method `setTable` creates a table of $n+1$ threads – the *philosophers* – each one sharing an object – the *fork* – with the nearby philosopher. Every philosopher, except one, grabs the fork on his left – the first argument – and on his right – the second argument – in this order and then releases them. The exceptional case is the `then`-branch (`n=0`) where the grabbing strategy is opposite. It is well-known that, when the method is invoked with `x.setTable(x,x,n)`,

E-mail address: cosimo.laneve@unibo.it.

[1] Research partly supported by the H2020-MSCA-RISE project ID 778233 "Behavioural Application Program Interfaces (BEHAPI)".

https://doi.org/10.1016/j.scico.2019.06.002
0167-6423/

no deadlock will ever occur because at least one philosopher has a strategy that is different from the other ones. On the contrary, if we change the `then`-branch into `sync(this){ sync(y){ 0 } }` a deadlock may occur because philosophers' strategies are all symmetric. It is worth to notice that `x.setTable(x,x,0)` is deadlock-free because it just locks twice the object `x`, which is admitted in the model with threads and locks – a thread may acquire a same lock several times (*lock-reentrancy*).

In order to ensure termination, current analyzers [4,7,1,16,27,28] use finite approximate models representing the dependencies between object names. The corresponding algorithms usually return false positives with input `x.setTable(x,x,n)` because they are not powerful enough to manage structures that are not statically bounded.

In [11,17] we solved this problem for value-passing CCS [21] and pi-calculus [22]. In that case, the technique used two formal models: Petri Nets [25] and *deadLock Analysis Models* – lams [12]. The former ones are a well-known model of concurrent and distributed systems; the latter ones are basic recursive models that collect dependencies and features recursion and dynamic name creation. In the pi-calculus analyzer, Petri Nets were used to verify the consistency of the communication protocol of every channel, while lams were used for guaranteeing the correctness of the dependencies between different channels. The analysis algorithm of [11,17] required a tool for verifying the reachability of Petri Nets that model channels' behaviors (which has exponential computational complexity with respect to the size of the net [15]) and a tool for analyzing lams (which has exponential computational complexity with respect to the number of arguments of functions).

In this paper we demonstrate that it is possible to define a deadlock analyzer for programs with threads and (reentrant) locks by only using an extension of lams. For example, the lam function corresponding to `setTable` is[2]

$$\texttt{setTable}(t,x,y) = (\nu\, s,z)\left((y,x)_t + ((x,z)_s \,\&\, \texttt{setTable}(t,z,y))\right).$$

The term $(y,x)_t$, called *dependency*, indicates that the thread t, which owns the lock of y, is going to grab the lock of x. The operation "+" and "&" are disjunction and conjunctions of dependencies, respectively. The index t of $(y,x)_t$ was missing in [12,11,17]; it has been necessary for modeling reentrant locks. In particular, (x,x) is a circularity in the standard lam model, whilst $(x,x)_t$ is not a circularity in the *extended lam model* because t can acquire x *twice*. Therefore, $\texttt{setTable}(t,x,x)$ manifests a circularity in the model of [12,11,17] and it does not in the extended model. A circularity in the extended model is $(y,x)_t\&(x,y)_s$, which denotes that two *different* threads are attempting to acquire two objects in different order. This lam gives $(x,x)_{\checkmark}$, which represents a circularity.

Because of the foregoing extension, the algorithm for detecting circularities in extended lams is different than the one in [11,17]. In particular, while there is a *decision algorithm* for the presence/absence of circularities in standard lams, in Section 2 we define an algorithm that verifies the absence and is imprecise in some cases (it may return that a lam will manifest a circularity while it will not be the case – a false positive).

We also define a simple language of concurrent programs featuring recursion, threads, shared objects and synchronization primitives inspired to those of `Java`. (The foregoing method `setTable` is a term of our calculus.) The syntax, semantics, and examples of the calculus are in Section 3. Section 4 reports a type system that associates lams to processes. For example, the type system returns the lam function `setTable` for the method `setTable`. As a byproduct of the type system and the algorithm for detecting circularities in lams, our technique can detect deadlocks of programs like `setTable`.

In order to deliver initial assessments of our technique, we have extended the basic language in Section 3 to cover several features of `Java`. In Section 5 we focus on inheritance that allows classes to be defined as sub-classes of other ones by deriving the corresponding method (and field) definitions and eventually redefine them (method overriding). In a language with inheritance, locating a method definition is performed at run-time through a mechanism called *dynamic dispatch* that uses the actual value of the called object. Since our technique extracts lams from programs *at static time*, the precise modeling of dynamic dispatch is not feasible. We therefore over-approximate the type by returning the lam functions of every possible overridden method in the subclasses. In Section 6 we also analyze assignments and recursive data-types. The analysis of other features of `Java` can be found in Garcia's PhD thesis [9]. This analysis does not address coordination primitives (methods `wait`, `notify` and `notifyAll`) and `Java` concurrent libraries, such as `java.util.concurrent`, – see Section 9 for a discussion.

Our theoretical developments are complemented by an analyzer prototype called `JaDA` [10,19]. While the type system in this paper simply checks static information, `JaDA` infers the behavioral types from the code. Inference is important in practice because it lightens the analysis but checking is crucial for type safety. Section 7 reports on initial assessments of the tool.

We discuss a few weaknesses of the techniques in Section 8 and we point to related works and give some concluding remarks in Section 9. The paper also includes an appendix that contains the demonstration of type safety for the language of Section 3 and the type system of Section 4.

This paper collects two conference contributions about deadlock analysis of `Java`-like languages: [18] and [19]. Here we extend [18] with the proof of type soundness and with the discussion on the extensions of the basic calculus in Section 3.

[2] Actually, this is a simplified form of `setTable`. The lam function associated to `setTable` by the type system in Section 4 has two additional names that record the last name synchronized by the thread t and the carrier of the invocation.

With respect to [19], we adopt a different presentation style. We discuss typing rules for the `Java` language, while in the conference paper the focus was on `Java` bytecode.

2. Lams and the algorithm for detecting circularities

This section extends the theory developed in [11,17] to cover thread reentrancy. In particular, the new definitions are those of transitive closure and Definition 2.3. Theorem 2.8 is new.

Preliminaries. We use an infinite set $\mathscr{A}$ of *names*, ranged over by x, y, t, s, $\cdots$. A relation on $\mathscr{A}$, denoted $\mathtt{R}, \mathtt{R}', \cdots$, is an element of $\mathscr{P}(\mathscr{A} \times \mathscr{A} \times (\mathscr{A} \cup \{\checkmark, \bullet\}))$, where $\mathscr{P}(\cdot)$ is the standard powerset operator, $\cdot \times \cdot$ is the cartesian product, and $\checkmark, \bullet \notin \mathscr{A}$ are two *special* names. The elements of $\mathtt{R}$, called *dependencies*, are denoted by $(x, y)_t$, where t is called *thread*. The name $\checkmark$ indicates that the dependency is due to the contributions of two or more threads; $\bullet$ indicates that the dependency is due to a thread whose name is unknown.

Let

- $\mathtt{R}^+$ be the least relation containing $\mathtt{R}$ and closed under the operations:
  1. if $(x, y)_t, (y, z)_{t'} \in \mathtt{R}^+$ and $t \neq t'$ then $(x, z)_\checkmark \in \mathtt{R}^+$;
  2. if $(x, y)_t, (y, z)_t \in \mathtt{R}^+$, $t \in \mathscr{A} \cup \{\checkmark\}$, then $(x, z)_t \in \mathtt{R}^+$;
  3. if $(x, y)_\bullet, (y, z)_\bullet \in \mathtt{R}^+$, $(x, y)_\bullet \neq (y, z)_\bullet$, then $(x, z)_\checkmark \in \mathtt{R}^+$.
- $\{\mathtt{R}_1, \cdots, \mathtt{R}_m\} \Subset \{\mathtt{R}'_1, \cdots, \mathtt{R}'_n\}$ if and only if, for all $\mathtt{R}_i$, there is $\mathtt{R}'_j$ such that
  1. if $(x, y)_t \in \mathtt{R}_i$, $t \in \mathscr{A}$, then $(x, y)_t \in {\mathtt{R}'_j}^+$
  2. if $(x, y)_\checkmark \in \mathtt{R}_i$ then either $(x, y)_\checkmark \in {\mathtt{R}'_j}^+$ or there is $t \in \mathscr{A}$ such that $(x, y)_t \in {\mathtt{R}'_j}^+$;
  3. if $(x, y)_\bullet \in \mathtt{R}_i$ then either $(x, y)_\bullet \in {\mathtt{R}'_j}^+$ or there is $t \in \mathscr{A}$ such that $(x, y)_t \in {\mathtt{R}'_j}^+$.
- $\{\mathtt{R}_1, \cdots, \mathtt{R}_m\} \& \{\mathtt{R}'_1, \cdots, \mathtt{R}'_n\} \stackrel{def}{=} \{\mathtt{R}_i \cup \mathtt{R}'_j \mid 1 \leq i \leq m \text{ and } 1 \leq j \leq n\}$.

We use $\mathscr{R}, \mathscr{R}', \cdots$ to range over $\{\mathtt{R}_1, \cdots, \mathtt{R}_m\}$, which are elements of $\mathscr{P}(\mathscr{P}(\mathscr{A} \times \mathscr{A} \times (\mathscr{A} \cup \{\checkmark, \bullet\})))$.

The names $\checkmark$ and $\bullet$ are managed in an ad-hoc way in the transitive closure $\mathtt{R}^+$ and in the relation $\Subset$. In particular, if $(x, y)_t$ and $(y, z)_{t'}$ belong to a relation and $t \neq t'$, the dependency obtained by transitivity, e.g. $(x, z)_\checkmark$, records that it has been produced by a contribution of two different threads – this is important for separating circularities, e.g. $(x, x)_\checkmark$, from lock reentrancy, e.g. $(x, x)_t$. The name $\bullet$ copes with another issue: it allows us to abstract away thread names that are created inside methods. For this reason the transitive dependency of $(x, y)_\bullet$ and $(y, z)_\bullet$ is $(x, z)_\checkmark$ because the threads producing $(x, y)_\bullet$ and $(y, z)_\bullet$ might be different. The meaning of $\mathscr{R} \Subset \mathscr{R}'$ is that $\mathscr{R}'$ is "*more precise*" with respect to pairs (x, y): if this pair is indexed with either $\checkmark$ or $\bullet$ in some $\mathtt{R} \in \mathscr{R}$ then it may be indexed by a t ($t \neq \checkmark$) in the corresponding (transitive closure) relation of $\mathscr{R}'$. For example $\{\{(x, y)_\bullet, (y, z)_\bullet, (x, z)_\checkmark\}\} \Subset \{\{(x, y)_t, (y, z)_t\}\}$ and $\{\{(x, x)_\bullet\}\} \Subset \{\{(x, x)_t, (x, x)_{t'}\}\}$.

Definition 2.1. A relation $\mathtt{R}$ *has a circularity* if $(x, x)_\checkmark \in \mathtt{R}^+$ for some x. A set of relations $\mathscr{R}$ *has a circularity* if there is $\mathtt{R} \in \mathscr{R}$ that has a circularity.

Lams. In our technique, dependencies are expressed by means of *lams* [12], noted ℓ, whose syntax is

$$\ell ::= 0 \quad | \quad (x, y)_t \quad | \quad (\nu\, x)\, \ell \quad | \quad \ell \,\&\, \ell \quad | \quad \ell + \ell \quad | \quad \mathtt{f}(\overline{x})$$

The term 0 is the empty type; $(x, y)_t$ specifies a dependency between the name x and the name y that has been created by (the thread) t. The operation $(\nu\ x)\ \ell$ creates a new name x whose scope is the type ℓ; the operations $\ell \,\&\, \ell'$ and $\ell + \ell'$ define the conjunction and disjunction of the dependencies in ℓ and ℓ', respectively. The operators $+$ and $\&$ are associative and commutative; the operator $\&$ has precedence over $+$. The term $\mathtt{f}(\overline{x})$ defines the invocation of $\mathtt{f}$ with arguments $\overline{x}$. The argument sequence $\overline{x}$ has always at least two elements in our case: the first element is the thread that performed the invocation, the second element is the last object whose lock has been acquired by it.

A *lam program* is a pair $(\mathscr{L}, \ell)$, where $\mathscr{L}$ is a *finite set* of *function definitions*

$$\mathtt{f}(\overline{x}) = \ell_\mathtt{f}$$

with $\ell_\mathtt{f}$ being the *body* of $\mathtt{f}$, and ℓ is the *main lam*. We always assume that (*i*) $\overline{x}$ is a sequence of pairwise different names and (*ii*) $\ell_\mathtt{f} = (\nu\ \overline{z})\ \ell'_\mathtt{f}$ where $\ell'_\mathtt{f}$ has no ν-binder. The constraint (*ii*) also applies to ℓ and, whenever we write $(\mathscr{L}, (\nu\ \overline{x})\ \ell')$ we always mean that ℓ' has no ν-binder. The function `setTable` in the Introduction is an example of a lam function.

Let a *lam context*, noted $\mathtt{L}[\ ]$, be a term derived by the following syntax:

$$\mathtt{L}[\ ] \quad ::= \quad [\ ] \quad | \quad \ell \,\&\, \mathtt{L}[\ ] \quad | \quad \ell + \mathtt{L}[\ ]$$

As usual $\mathtt{L}[\ell]$ is the lam where the hole of $\mathtt{L}[\ ]$ is replaced by ℓ. We remark that, according to the syntax, lam contexts have no ν-binder.

Since relations $\mathtt{R}$ are collections of dependencies, with an abuse of notation, we often use $\mathtt{R}$ and $\&_{(x,y)_t\in\mathtt{R}}(x,y)_t$ interchangeably, allowing the context to disambiguate whether we mean a relation or a lam. For example, we will often write $\mathtt{L}[\mathtt{R}]$ meaning the lam $\mathtt{L}[\&_{(x,y)_t\in\mathtt{R}}(x,y)_t]$. Similarly, when we write $\mathtt{R}\{^y/_x\}$, we may mean either the relation $\mathtt{R}$ where the occurrences of x in the dependencies have been replaced by y or the lam $(\&_{(x,y)_t\in\mathtt{R}}(x,y)_t)\{^y/_x\}$.

The operational semantics of a lam program $\big(\mathscr{L},(\nu\ \overline{x})\ \ell\big)$ is a transition system where *states* are lams, the *transition relation* is the smallest transition relation induced by the following rule

$$\textsc{(Red)}\quad \frac{\mathtt{f}(\overline{x})=(\nu\ \overline{z})\ \ell_{\mathtt{f}}\ \in\mathscr{L}\qquad \overline{z}'\text{ are fresh}}{\mathtt{L}[\mathtt{f}(\overline{u})]\longrightarrow \mathtt{L}[\ell_{\mathtt{f}}\{^{\overline{z}'}/_{\overline{z}}\}\{^{\overline{u}}/_{\overline{x}}\}]}$$

with initial state ℓ. We write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$.

For example, if $\mathtt{f}(t,x)=(\nu\ s,z)\ ((x,z)_t\&\mathtt{f}(s,z))$ then $\mathtt{f}(t,x)\longrightarrow(x,z')_t\&\mathtt{f}(t',z')$, where t' and z' are fresh names. By continuing the evaluation of $\mathtt{f}(t,x)$, the reader may observe that (*i*) every invocation creates new fresh names and (*ii*) the evaluation does not terminate because $\mathtt{f}$ is recursive. These two points imply that a lam model may have infinite states, which makes the analysis nontrivial.

Flattening and circularities. Lams represent elements of the set $\mathscr{P}(\mathscr{P}(\mathscr{A}\times\mathscr{A}\times(\mathscr{A}\cup\{\checkmark,\bullet\})))$. This property is displayed by the following flattening function. Let $\mathscr{L}$ be a set of function definitions and let $I(\cdot)$, called *flattening*, be a function on lams that (1) maps function name $\mathtt{f}$ defined in $\mathscr{L}$ to elements of $\mathscr{P}(\mathscr{P}(\mathscr{A}\times\mathscr{A}\times(\mathscr{A}\cup\{\checkmark,\bullet\})))$ and (2) is defined on lams as follows

$$\begin{aligned}
&I(0)=\{\varnothing\},\qquad I((x,y)_t)=\{\{(x,y)_t\}\},\qquad I(\ell\&\ell')=I(\ell)\&I(\ell'),\\
&I(\ell+\ell')=I(\ell)\cup I(\ell'),\qquad I((\nu\ x)\ \ell)=I(\ell)\{^{x'}/_x\}\ \text{ with } x'\text{ fresh},\\
&I(\mathtt{f}(\overline{u}))=I(\mathtt{f})\{^{\overline{u}}/_{\overline{x}}\}\ (\text{where }\overline{x}\text{ are the formal parameters of }\mathtt{f}).
\end{aligned}$$

Let $I^{\perp}$ be the map such that, for every $\mathtt{f}$ defined in $\mathscr{L}$, $I^{\perp}(\mathtt{f})=\{\varnothing\}$. For example, let $\mathtt{setTable}$ be the function in the Introduction and let

$$I(\mathtt{setTable})=\{\{(y,x)_t\}\}\quad \ell=\mathtt{setTable}(t,x,y)\&(x,y)_s+(x,y)_s.$$

Then $I(\ell)=\big\{\{(y,x)_t,(x,y)_s\},\ \{(x,y)_s\}\big\},\ I^{\perp}(\ell)=\big\{\{(x,y)_s\}\big\}$.

Definition 2.2. A lam ℓ *has a circularity* if $I^{\perp}(\ell)$ has a circularity. A lam program $\big(\mathscr{L},(\nu\ \overline{x})\ \ell\big)$ *has a circularity* if there is $\ell\longrightarrow^*\ell'$ and ℓ' has a circularity.

For example the above lam ℓ has a circularity because

$$\begin{aligned}
&\mathtt{setTable}(t,x,y)\&(x,y)_s+(x,y)_s\\
&\qquad\longrightarrow((y,x)_t+(x,z)_s\&\mathtt{setTable}(t,z,y))\&(x,y)_s+(x,y)_s\\
&\qquad\quad=\ell'
\end{aligned}$$

and $I^{\perp}(\ell')$ has a circularity.

Fixpoint definition of the interpretation function. Our algorithm relies on the computation of lam functions' interpretation, which is done by a standard fixpoint technique.

Let $\mathscr{L}$ be the set $\mathtt{f}_i(\overline{x}_i)=(\nu\ \overline{z_i})\ \ell_i$, with $i\in 1..n$. Let $A=\bigcup_{i\in 1..n}\overline{x}_i$ and $\varkappa$ be a special name that does not occur in $\big(\mathscr{L},\ell\big)$. We use the domain $\Big(\mathscr{P}(\mathscr{P}((A\cup\{\varkappa\})\times(A\cup\{\varkappa\})\times(A\cup\{\checkmark,\bullet\}))),\ \subseteq\Big)$ which is a *finite* lattice [5].

Definition 2.3. Let $\mathtt{f}_i(\overline{x}_i)=(\nu\ \overline{z_i})\ \ell_i$, with $i\in 1..n$, be the function definitions in $\mathscr{L}$. The family of flattening functions $I^{(k)}_{\mathscr{L}}:\{\mathtt{f}_1,\cdots,\mathtt{f}_n\}\to\mathscr{P}(\mathscr{P}((A\cup\{\varkappa\})\times(A\cup\{\varkappa\})\times(A\cup\{\checkmark,\bullet\})))$ is defined as follows

$$I^{(0)}_{\mathscr{L}}(\mathtt{f}_i)=\{\varnothing\}\qquad I^{(k+1)}_{\mathscr{L}}(\mathtt{f}_i)=\{\mathtt{proj}^{\overline{z_i}}_{\overline{x_i}}(\mathtt{R}^+)\mid \mathtt{R}\in I^{(k)}_{\mathscr{L}}(\ell_i)\}$$

where

$$\begin{aligned}
\mathtt{proj}^{\overline{z}}_{\overline{x}}(\mathtt{R})\stackrel{def}{=}\ &\{(u,v)_t\mid (u,v)_t\in\mathtt{R}\text{ and }u,v\in\overline{x}\text{ and }t\in\overline{x}\cup\{\checkmark\}\}\\
&\cup\{(\varkappa,\varkappa)_{\checkmark}\mid (u,u)_{\checkmark}\in\mathtt{R}\text{ and }u\notin\overline{x}\}\\
&\cup\{(u,v)_{\bullet}\mid (u,v)_t\in\mathtt{R}\text{ and }u,v\in\overline{x}\text{ and }t\in\overline{z}\}
\end{aligned}$$

We notice that $I^{(0)}_{\mathscr{L}}$ is the function $I^{\perp}$. Let us analyze the definition of $I^{(k+1)}_{\mathscr{L}}(\mathtt{f}_i)$ and, in particular, the function $\mathtt{proj}$:

- first of all, notice that $\mathtt{proj}$ applies to the transitive closures of relations, which may have names in A, $\overline{z}_i$, $\checkmark$, $\bullet$ and $\varkappa$;
- the transitive closure operation is crucial because a circularity may follow with the key contribution of fresh names. For instance the model of $\mathtt{f}(x) = (\nu\ t, t', z)\ (x, z)_t \,\&\, (z, x)_{t'}$ is $\{\{(x, x)_{\checkmark}\}\}$; the model of $\mathtt{g}() = (\nu\ t, t', x, y)\ (x, y)_t \,\&\, (y, x)_{t'}$ is $\{\{(\varkappa, \varkappa)_{\checkmark}\}\}$ (this is the reason why we use the name $\varkappa$);
- every dependency $(u, v)_t \in \mathtt{proj}^{\overline{z}}_{\overline{x}}(\mathtt{R})$ is such that $u, v \in \overline{x}$, except for $(\varkappa, \varkappa)_{\checkmark}$. For example, if $\mathtt{f}'(x, y) = (\nu\ s, z)\ ((x, y)_s \,\&\, (x, z)_s)$ then, if we invoke $\mathtt{f}'(u, v)$ we obtain $(u, v)_{t'} \,\&\, (u, z')_{t'}$, where t' and z' are fresh object names. This lam may be simplified because, being z' fresh and unknown elsewhere, the dependency $(u, z')_{t'}$ will never be involved in a circularity. For example, if we have $\ell = (v, u)_t \,\&\, \mathtt{f}'(u, v)$ then we may safely reason on ℓ'-simplified $(u, v)_t \,\&\, (u, v)_{t'}$. For this reason we drop the dependencies containing fresh names *after their contribution to the transitive closure has been computed*;
- the same argument does not apply to names used as threads. For example, in the above ℓ'-simplified lam we cannot drop $(u, v)_{t'}$ because t' is fresh. In fact, the context $(v, u)_t \,\&\, (u, v)_{t'}$ gives a circularity. Therefore, dependencies whose thread names are fresh must be handled in a different way. We take a simple solution: these dependencies all have $\bullet$ as thread name. That is, we assume that they are all generated by the contribution of different threads. For example, $\mathtt{g}'(x, y) = (\nu\ t)\ (x, y)_t$. Then, $I^{(1)}_{\mathscr{L}}(\mathtt{g}') = \{\{(x, y)_{\bullet}\}\}$.

Example 2.4. The flattening functions of `setTable` are

$$I^{(0)}_{\mathscr{L}}(\mathtt{setTable}) = \{\varnothing\}$$
$$I^{(1)}_{\mathscr{L}}(\mathtt{setTable}) = \{\{(y, x)_t\}\}$$

As another example, consider the function $\mathtt{g}(x, y, z) = (\nu\ t, u)\ (x, y)_t \,\&\, \mathtt{g}(y, z, u)$. Then:

$$I^{(0)}_{\mathscr{L}}(\mathtt{g}) = \{\varnothing\}$$
$$I^{(1)}_{\mathscr{L}}(\mathtt{g}) = \{\{(x, y)_{\bullet}\}\}$$
$$I^{(2)}_{\mathscr{L}}(\mathtt{g}) = \{\{(x, y)_{\bullet}, (y, z)_{\bullet}, (x, z)_{\checkmark}\}\}$$

Proposition 2.5. *Let* $\mathtt{f}(\overline{x}) = (\nu\ \overline{z})\ \ell_{\mathtt{f}} \in \mathscr{L}$.

1. *For every* k, $I^{(k)}_{\mathscr{L}}(\mathtt{f}) \in \mathscr{P}(\mathscr{P}((\overline{x} \cup \{\varkappa\}) \times (\overline{x} \cup \{\varkappa\}) \times (\overline{x} \cup \{\checkmark, \bullet\})))$.
2. *For every* k, $I^{(k)}_{\mathscr{L}}(\mathtt{f}) \Subset I^{(k+1)}_{\mathscr{L}}(\mathtt{f})$.

Proof. (1) follows by definition. As regards (2), we observe that $I(\ell)$ is monotonic on I. That is, if I and I' are two flattening functions such that, for every $\mathtt{f}$, $I(\mathtt{f}) \Subset I'(\mathtt{f})$ then it is possible to demonstrate by a standard structural induction that $I(\ell) \Subset I'(\ell)$. An induction on k gives $I^{(k)}_{\mathscr{L}}(\mathtt{f}) \Subset I^{(k+1)}_{\mathscr{L}}(\mathtt{f})$. □

Since, for every k, $I^{(k)}_{\mathscr{L}}(\mathtt{f}_i)$ ranges over a finite lattice, by the fixpoint theory [5], there exists m such that $I^{(m)}_{\mathscr{L}}$ is a fixpoint, namely $I^{(m)}_{\mathscr{L}} \approx I^{(m+1)}_{\mathscr{L}}$ where $\approx$ is the equivalence relation induced by $\Subset$. In the following, we let $I_{\mathscr{L}}$, called the *interpretation function* (of a lam), be the least fixpoint $I^{(m)}_{\mathscr{L}}$. In Example 2.4, $I^{(1)}_{\mathscr{L}}$ is the fixpoint of `setTable` and $I^{(2)}_{\mathscr{L}}$ is the fixpoint of $\mathtt{g}$.

Proposition 2.6. *Let* $\mathtt{L}$ *be a lam context,* ℓ *be a lam with no* ν *binder, and* $I(\cdot)$ *be a flattening. Then we have:*

1. $I(\mathtt{L}[\ell])$ *has a circularity if and only if* $I(\mathtt{L}[\mathtt{R}])$ *has a circularity for some* $\mathtt{R} \in I(\ell)$.
2. *Let* $\mathtt{f}(\overline{x}) = (\nu\ \overline{z})\ \ell_{\mathtt{f}} \in \mathscr{L}$ *and* $\mathtt{R} \in I(\ell_{\mathtt{f}}\{\overline{z'}/\overline{z}\})$ *with* $\overline{z'}$ *fresh. If* $I(\mathtt{L}[\mathtt{R}\{\overline{u}/\overline{x}\}])$ *has a circularity then* $I(\mathtt{L}[(\mathtt{proj}^{\overline{z}}_{\overline{x}}(\mathtt{R}^+))\{\overline{u}/\overline{x}\}])$ *has a circularity.*

Proof. Property 1 follows from the definitions. To see 2, we use a straightforward induction on $\mathtt{L}$. We analyze the basic case $\mathtt{L} = [\,]$: the general case follows by induction. Let $\mathtt{R} \in I(\ell_{\mathtt{f}}\{\overline{z'}/\overline{z}\})$ such that $\mathtt{R}\{\overline{u}/\overline{x}\}$ has a circularity. There are two cases:

- $(v, v)_{\checkmark} \in \mathtt{R}^+$. By definition of $\mathtt{proj}^{\overline{z}}_{\overline{x}}(\mathtt{R}^+)$ either $(v, v)_{\checkmark} \in \mathtt{proj}^{\overline{z}}_{\overline{x}}(\mathtt{R}^+)$, when $v \notin \overline{z'}$, or $(\varkappa, \varkappa)_{\checkmark} \in \mathtt{proj}^{\overline{z}}_{\overline{x}}(\mathtt{R}^+)$, otherwise. In this case the statement 2 follows immediately.
- $(v, v)_{\checkmark} \in \mathtt{R}\{\overline{u}/\overline{x}\}^+$. By definition of transitive closure $\mathtt{R}\{\overline{u}/\overline{x}\}^+ = (\mathtt{R}^+)\{\overline{u}/\overline{x}\}$. Then there is a dependency $(x_1, x_2)_{\checkmark} \in \mathtt{R}^+$ such that $(x_1, x_2)_{\checkmark}\{\overline{u}/\overline{x}\} = (v, v)_{\checkmark}$. By definition of $\mathtt{proj}$, $(x_1, x_2)_{\checkmark} \in \mathtt{proj}^{\overline{z}}_{\overline{x}}(\mathtt{R}^+)$. Therefore $\mathtt{proj}^{\overline{z}}_{\overline{x}}(\mathtt{R}^+)\{\overline{u}/\overline{x}\}$ has also a circularity. □

Lemma 2.7. *Let* $\mathscr{L} = \{\mathtt{f}_1(\overline{x}_1) = (\nu\ \overline{z}_1)\ \ell_1, \cdots, \mathtt{f}_n(\overline{x}_n) = (\nu\ \overline{z}_n)\ \ell_n\}$ *and let*

$$\mathtt{L}[\mathtt{f}_{i_1}(\overline{u}_1)]\cdots[\mathtt{f}_{i_m}(\overline{u}_m)] \longrightarrow^m \mathtt{L}[\ell_{i_1}\{\overline{z}'_1/_{\overline{z}_{i_1}}\}\{\overline{u}_1/_{\overline{x}_{i_1}}\}]\cdots[\ell_{i_m}\{\overline{z}'_m/_{\overline{z}_{i_m}}\}\{\overline{u}_m/_{\overline{x}_{i_m}}\}]$$

where $\mathtt{L}[\cdot]\cdots[\cdot]$ *is a multiple context without function invocations.*

If $I^{(k)}_{\mathscr{L}}(\mathtt{L}[\ell_{i_1}\{\overline{z}'_1/_{\overline{z}_{i_1}}\}\{\overline{u}_1/_{\overline{x}_{i_1}}\}]\cdots[\ell_{i_m}\{\overline{z}'_m/_{\overline{z}_{i_m}}\}\{\overline{u}_m/_{\overline{x}_{i_m}}\}])$ *has a circularity then* $I^{(k+1)}_{\mathscr{L}}(\mathtt{L}[\mathtt{f}_{i_1}(\overline{u}_1)]\cdots[\mathtt{f}_{i_m}(\overline{u}_m)])$ *has also a circularity.*

Proof. To show the implication suppose that

$$I^{(k)}_{\mathscr{L}}(\mathtt{L}[\ell_{i_1}\{\overline{z}'_1/_{\overline{z}_{i_1}}\}\{\overline{u}_1/_{\overline{x}_{i_1}}\}]\cdots[\ell_{i_m}\{\overline{z}'_m/_{\overline{z}_{i_m}}\}\{\overline{u}_m/_{\overline{x}_{i_m}}\}])$$

has a circularity. By repeated applications of Proposition 2.6(1), there exists $\mathtt{R}_j \in I^{(k)}_{\mathscr{L}}(\ell_{i_1}\{\overline{z}'_j/_{\overline{z}_{i_j}}\}\{\overline{u}_j/_{\overline{x}_{i_j}}\})$ with $1 \le j \le m$ such that $I^{(k)}_{\mathscr{L}}(\mathtt{L}[\mathtt{R}_1]\cdots[\mathtt{R}_m])$ has a circularity. It is easy to verify that every $\mathtt{R}_j$ may be written as $\mathtt{R}'_j\{\overline{u}_j/_{\overline{x}_{i_j}}\}$, for some $\mathtt{R}'_j$. By repeated application of Proposition 2.6(2), we have that

$$I^{(k)}_{\mathscr{L}}(\mathtt{L}[\mathtt{proj}^{\overline{z}_{i_1}}_{\overline{x}_{i_1}}(\mathtt{R}'^{+}_1)\{\overline{u}_1/_{\overline{x}_1}\}]\cdots[\mathtt{proj}^{\overline{z}_{i_m}}_{\overline{x}_{i_m}}(\mathtt{R}'^{+}_m)\{\overline{u}_m/_{\overline{x}_m}\}])$$

has a circularity. Since, for every $1 \le j \le m$,

$$\mathtt{proj}^{\overline{z}_{i_j}}_{\overline{x}_{i_j}}(\mathtt{R}'^{+}_j)\{\overline{u}_1/_{\overline{x}_1}\} \in I^{(k+1)}_{\mathscr{L}}(\mathtt{f}_{i_j}(\overline{u}_j))$$

and since $\mathtt{L}$ has no function invocation, we derive that $I^{(k+1)}_{\mathscr{L}}(\mathtt{L}[\mathtt{f}_{i_1}(\overline{u}_1)]\cdots[\mathtt{f}_{i_m}(\overline{u}_m)])$ has also a circularity. □

Unlike [11,17], Lemma 2.7 is strict in our case, for every k. For example, consider

$$\mathtt{h}(x, y, z) = (\nu\ t)\ (x, y)_t \,\&\, (y, z)_t\ .$$

Then, when $k \ge 1$, $I^{(k)}_{\mathscr{L}}(\mathtt{h}) = \{\{(x, y)_\bullet, (y, z)_\bullet\}\}$. Notice that $I^{(k)}_{\mathscr{L}}(\mathtt{h}(x, y, x)) = \{\{(x, y)_\bullet, (y, x)_\bullet\}\}$, which has a circularity – see Definition 2.1. However

$$I^{(k)}_{\mathscr{L}}((x, y)_t \,\&\, (y, x)_t) = \{\{(x, y)_t, (y, x)_t, (x, x)_t\}\}$$

has no circularity (this is a case of reentrant lock).

Theorem 2.8. *Let* $\big(\mathscr{L}, (\nu\ \overline{x})\ \ell\big)$ *be a lam program and* $\ell \longrightarrow^* \ell'$. *If* $I_{\mathscr{L}}(\ell')$ *has a circularity then* $I_{\mathscr{L}}(\ell)$ *has also a circularity. Therefore* $\big(\mathscr{L}, \ell\big)$ *has no circularity if* $I_{\mathscr{L}}(\ell)$ *has no circularity.*

Proof. Let ℓ' have a circularity. Hence, by definition, $I^{\perp}(\ell')$ has a circularity and, since $I^{\perp}(\ell') = I^{(0)}_{\mathscr{L}}(\ell')$, by Proposition 2.5(2) $I^{(0)}_{\mathscr{L}}(\ell') \Subset I_{\mathscr{L}}(\ell')$. Therefore $I_{\mathscr{L}}(\ell')$ has also a circularity. Then, by Lemma 2.7, since $I_{\mathscr{L}}$ is the fixpoint interpretation function, $I_{\mathscr{L}}(\ell)$ has also a circularity. □

Our algorithm for verifying that a lam will never manifest a circularity consists of computing $I_{\mathscr{L}}(\mathtt{f})$, for every $\mathtt{f}$, and $I_{\mathscr{L}}(\ell)$, where ℓ is the main lam. As discussed in this section, $I_{\mathscr{L}}(\mathtt{f})$ uses a saturation technique on names based on a powerset construction. Hence it has a computational complexity that is *exponential* on the number of names. We remind that the names we consider are the *arguments* of lam functions (that corresponds to methods' arguments), which are usually not so many. In fact, this algorithm is quite efficient in practice [10].

3. The language and its semantics

In this section we define a simple programming model of concurrent object-oriented languages featuring threads and object synchronizations (the reader may easily recognize the basic operations of thread creation and synchronization used in `Java` and `C#`). We give a description of how deadlock may be identified, and discuss few examples. The following section contains the type system that associates lams to the programs.

Our model has two disjoint countable sets of names: there are *integer and object names*, ranged over by $x, y, z, t, s, \cdots$, and *method names*, ranged over by $\mathtt{m}$, $\mathtt{n}$, $\cdots$. A *program* is a tuple

$$\Big(\mathtt{C}_1\ \mathscr{D}_1, \cdots, \mathtt{C}_n\ \mathscr{D}_n, P\Big)$$

where $\mathscr{D}_i$ are *finite sets* of *method name definitions* $\mathtt{m}(\overline{\mathtt{T}}\ \overline{x}) = P_{\mathtt{m}}$, with $\overline{\mathtt{T}}\ \overline{x}$ being the *formal parameters* $\overline{x}$ and their *types* $\overline{\mathtt{T}}$ and $P_{\mathtt{m}}$ being the *body* of $\mathtt{m}$; we assume that method name definitions have an implicit formal parameter $\mathtt{C}_i$ *this* which refers to the carrier. The process P is the *main process*. Types $\mathtt{T}$ are either integers `int` or $\mathtt{C}_i$.

The syntax of processes P and expressions e is defined below

$$\begin{array}{rcl} P & ::= & 0 \mid (\texttt{new}\ \texttt{C}\ x)\ P \mid \texttt{fork}\{P\}\ P \mid \texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ P \mid x.\texttt{m}(\overline{e}) \\ & \mid & \texttt{sync}(x)\{\ P\ \} \\ e & ::= & x \mid v \mid e\ \texttt{op}\ e \end{array}$$

A process can be the inert process 0, or a restriction $(\texttt{new}\ \texttt{C}\ x)\ P$ that behaves like P except that the external environment cannot access to the object x of class C, or the spawn $\texttt{fork}\{Q\}\ P$ of a new thread Q by a process P, or a conditional $\texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q$ that evaluates e and behaves either like P or like Q depending on whether the value is $\neq 0$ (*true*) or $= 0$ (*false*), or an invocation $x.\texttt{m}(\overline{e})$ of the method m with carrier x and actual parameters $\overline{e}$. The last process is $\texttt{sync}(x)\{\ P\ \}$ that executes P with exclusive access to x.

An expression e can be a name x, an integer value v, or a generic binary operation on integers $\texttt{op}$, where $\texttt{op}$ ranges over a set including the usual operators like $+$, $\leq$, etc. Integer expressions without names (*constant expressions*) may be evaluated to an integer value (the definition of the evaluation of constant expressions is omitted). Let $[\![e]\!]$ be the evaluation of an expression e: $[\![e]\!]$ is undefined when e is an integer expression that contains integer names; $[\![x]\!] = x$ when x is a non-integer name.

To define the semantics of the above language, we use *states*, ranged over by $\mathbb{P}$, that are multisets of *threads* $P \bullet \sigma$, where σ is a (possibly empty) sequence of object names representing the locks that the process P has already grabbed (in order, from the left to right). When P terminates, the locks of σ must be all released. The thread $P \bullet \sigma$ is *reentrant* on x when σ contains at least two occurrences of x. We usually write states as $P_1 \bullet \sigma_1 \mid \cdots \mid P_n \bullet \sigma_n$ and sometime shorten them into $\prod_{i\in 1..n} P_i \bullet \sigma_i$. We write $x \in \mathbb{P}$ if $\mathbb{P}$ contains σ and x occurs in σ. Since states are multiset, we identify those that are equal up-to associativity and commutativity of $\mid$. Empty sequences σ are written ε. In order to ease the definitions, we will use few notational conventions

- Object names are partitioned into infinitely countable disjoint sets that are addressed by class names. The notation $z = \mathit{fresh}(\texttt{C})$ states that z is a fresh name in the set C (e.g. z does not occur in the state).
- We make explicit that z is a name in the set C by writing $z \in \texttt{C}$.
- If a program Ψ has a pair $\texttt{C}\ \mathscr{D}$ such that $\mathscr{D}$ contains a method $\texttt{m}(\overline{\texttt{T}}\ \overline{x}) = P$ then we write $\texttt{C.m}(\overline{\texttt{T}}\ \overline{x}) = P \in \Psi$.

Definition 3.1. The *operational semantics* of a program $\Psi = \left(\texttt{C}_1\ \mathscr{D}_1, \cdots, \texttt{C}_n\ \mathscr{D}_n, P\right)$ is a transition system where the initial state is $P \bullet \varepsilon$, and the *transition relation* $\longrightarrow_\Psi$ is the least one closed under the rules:

$$\text{(Zero)}\quad 0 \bullet \sigma \longrightarrow_\Psi$$

$$\text{(NewO)}\quad \frac{z = \mathit{fresh}(\texttt{C})}{(\texttt{new}\ \texttt{C}\ x)\ P \bullet \sigma \mid \mathbb{P} \longrightarrow_\Psi P\{^z/_x\} \bullet \sigma \mid \mathbb{P}} \qquad \text{(NewT)}\quad \texttt{fork}\{P\}\ Q \bullet \sigma \longrightarrow_\Psi P \bullet \varepsilon \mid Q \bullet \sigma$$

$$\text{(IfT)}\quad \frac{[\![e]\!] \neq 0}{\texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q \bullet \sigma \longrightarrow_\Psi P \bullet \sigma} \qquad \text{(IfF)}\quad \frac{[\![e]\!] = 0}{\texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ Q \bullet \sigma \longrightarrow_\Psi Q \bullet \sigma}$$

$$\text{(Call)}\quad \frac{[\![\overline{e}]\!] = \overline{v} \quad x \in \texttt{C} \quad \texttt{C.m}(\overline{\texttt{T}}\ \overline{z}) = P \in \Psi}{x.\texttt{m}(\overline{e}) \bullet \sigma \longrightarrow_\Psi P\{^{x,\overline{v}}/_{\mathit{this},\overline{z}}\} \bullet \sigma} \qquad \text{(Sync)}\quad \frac{x \notin \mathbb{P}}{\texttt{sync}(x)\{\ P\ \} \bullet \sigma \mid \mathbb{P} \longrightarrow_\Psi P \bullet \sigma \cdot x \mid \mathbb{P}}$$

We often omit the subscript of $\longrightarrow_\Psi$ when it is clear from the context. We write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$.

We discuss the relevant rules. Rule (Zero) models termination: every lock taken by the thread is released and the thread disappears from the state (which is a multiset). This is because, according to the syntax, there is no continuation in this case. Rule (NewO) creates a new object of class C: we simply take a name that does not occur in the state, replace the bound name with the fresh one, and remove the binder. We observe that the renaming only applies to P: being x new, it cannot have been already locked (σ is not in the scope of the renaming). Rule (NewT) spawns a new thread that has an empty sequence of locks (because it has not performed any synchronization so far). Rule (Call) dispatches the invocation to the right method definition and executes the method body once the formal parameters have been replaced by the actual ones. Rule (Sync) models synchronizations. A thread may lock x provided it is not already locked by another thread in the state – premise $x \notin \mathbb{P}$. We observe that x may occur in σ, that is a thread may lock x several times (thread reentrancy).

Our basic programming model does not retain the explicit instruction for sequential composition, e.g. $P\ ;\ Q$. However, this operation may be easily encoded using synchronizations:

$$(\texttt{new C}\ x)\ \texttt{sync}(x)\{\ \texttt{fork}\{\texttt{sync}(x)\{\ Q\ \}\}\ P\ \}$$

That is, we sequentialize P and Q by letting P to acquire the lock of a new object x *before* spawning Q. In turn Q will begin *after* the termination of P because the lock of x is grabbed by it. According to our rules, we have

$$\begin{array}{l}(\texttt{new C}\ x)\ \texttt{sync}(x)\{\ \texttt{fork}\{\texttt{sync}(x)\{\ Q\ \}\}\ P\ \} \bullet \sigma\\ \qquad\longrightarrow \texttt{sync}(x')\{\ \texttt{fork}\{\texttt{sync}(x')\{\ Q\ \}\}\ P\ \} \bullet \sigma\\ \qquad\longrightarrow \texttt{fork}\{\texttt{sync}(x')\{\ Q\ \}\}\ P \bullet \sigma \cdot x'\\ \qquad\longrightarrow P \bullet \sigma \cdot x' \mid \texttt{sync}(x')\{\ Q\ \} \bullet \varepsilon\end{array}$$

and, by rule (Sync), Q cannot start as long as x' is locked by P.

Definition 3.2 *(Deadlock-freedom).* A program $\left(\texttt{C}_1\ \mathscr{D}_1, \cdots, \texttt{C}_n\ \mathscr{D}_n, P\right)$ is *deadlock-free* if the following condition holds:

whenever $P \bullet \varepsilon \longrightarrow^* \mathbb{P}$ and $\mathbb{P} = \texttt{sync}(x)\{\ Q\ \} \bullet \sigma \mid \mathbb{P}'$ then there exists $\mathbb{P}''$ such that $\mathbb{P} \longrightarrow \mathbb{P}''$.

Example 3.3. We select three processes and discuss their behaviors, highlighting whether they deadlock or not:

- $\texttt{fork}\{sync(x)\{\ sync(y)\{\ 0\ \}\ \}\}\ sync(x)\{\ sync(y)\{\ 0\ \}\ \}$. This process spawns a thread that acquire the locks of x and y in the *same order* of the main thread: no deadlock will ever occur.
- On the contrary, the process $\texttt{fork}\{sync(y)\{\ sync(x)\{\ 0\ \}\ \}\}\ sync(x)\{\ sync(y)\{\ 0\ \}\ \}$ spawns a thread acquiring the locks in reverse order. This is a computation giving a deadlock:

$$\begin{array}{l}\texttt{fork}\{sync(y)\{\ sync(x)\{\ 0\ \}\ \}\}\ sync(x)\{\ sync(y)\{\ 0\ \}\ \} \bullet \varepsilon\\ \qquad\longrightarrow\quad sync(y)\{\ sync(x)\{\ 0\ \}\ \} \bullet \varepsilon \mid sync(x)\{\ sync(y)\{\ 0\ \}\ \} \bullet \varepsilon\\ \qquad\longrightarrow\quad sync(x)\{\ 0\ \} \bullet y \mid sync(x)\{\ sync(y)\{\ 0\ \}\ \} \bullet \varepsilon\\ \qquad\longrightarrow\quad sync(x)\{\ 0\ \} \bullet y \mid sync(y)\{\ 0\ \} \bullet x\end{array}$$

- The following method

$$\begin{array}{l}\texttt{m(C}\ x, \texttt{C}\ y, \texttt{int}\ n) =\\ \qquad \texttt{if}\ (n = 0)\ \texttt{then}\ \texttt{fork}\{sync(y)\{\ sync(x)\{\ 0\ \}\ \}\}\ sync(y)\{\ 0\ \}\\ \qquad \texttt{else}\ sync(x)\{\ this.\texttt{m}(x, y, n-1)\ \}\end{array}$$

  performs n-nested synchronizations on *this* (reentrancy) and then spawns a thread acquiring the locks y and x in this order, while the main thread acquire the lock y. This method deadlocks for every $n \geq 1$, however it never deadlocks when $n \leq 0$.

4. Static semantics

Environments, ranged over by Γ, contain the types of objects, e.g. $x : \texttt{C}$, the type of integer variables e.g. $x : \texttt{int}$, and the types of method names, e.g. $\texttt{C.m} : [\overline{\texttt{T}}]$. Without loss of generality, in method definitions of Fig. 1 we order formal parameters of methods: those of type class are in front of those of type integer. Therefore $\Gamma(\texttt{C.m})$ will have shape $[\overline{\texttt{C}'}, \overline{\texttt{int}}]$.

Let $dom(\Gamma)$ be the domain of Γ and let

- $\Gamma, x{:}\texttt{T}$, when $x \notin dom(\Gamma)$

$$(\Gamma, x{:}\texttt{T})(y) \stackrel{def}{=} \begin{cases}\texttt{T} & \text{if } y = x\\ \Gamma(x) & \text{otherwise}\end{cases}$$

- $\Gamma + \Gamma'$, when $x \in dom(\Gamma) \cap dom(\Gamma')$ implies $\Gamma(x) = \Gamma'(x)$:

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

Processes:

$$\text{(T-Zero)}\ \ \Gamma;\sigma \vdash_t 0 : (\sigma)^t \qquad \text{(T-New)}\ \ \frac{\Gamma, x{:}\mathtt{C};\sigma \vdash_t P : \ell \quad x \notin \sigma}{\Gamma;\sigma \vdash_t (\mathtt{new}\ \mathtt{C}\ x)\ P : (\nu\ x)\ \ell}$$

$$\text{(T-Sync)}\ \ \frac{\Gamma;\sigma \cdot x \vdash_t P : \ell}{\Gamma;\sigma \vdash_t \mathtt{sync}(x)\{\ P\ \} : \ell} \qquad \text{(T-If)}\ \ \frac{\Gamma \vdash_t e : \mathtt{int} \quad \Gamma;\sigma \vdash_t P : \ell \quad \Gamma;\sigma \vdash_t P' : \ell'}{\Gamma;\sigma \vdash_t \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ P' : \ell + \ell'}$$

$$\text{(T-Par)}\ \ \frac{\Gamma;\sigma \vdash_t P : \ell \qquad \Gamma, t' : \mathtt{Thread}, u' : \mathtt{Obj}; u' \vdash_{t'} P' : \ell'}{\Gamma;\sigma \vdash_t \mathtt{fork}\{P'\}\ P : \ell \,\&\, (\nu\ t', u')\ \ell'}$$

$$\text{(T-Call)}\ \ \frac{\Gamma(x) = \mathtt{C} \quad \Gamma(\mathtt{C.m}) = [\overline{\mathtt{C}'}, \overline{\mathtt{int}}] \quad \Gamma \vdash \overline{u} : \overline{\mathtt{C}'} \quad \Gamma \vdash \overline{e} : \overline{\mathtt{int}}}{\Gamma;\sigma \cdot z \vdash_t x.\mathtt{m}(\overline{u}, \overline{e}) : \mathtt{f}_{\mathtt{C.m}}(t, z, x, \overline{u}) \& (\sigma \cdot z)^t}$$

Expressions:

$$\text{(T-Int)}\ \ \Gamma \vdash v : \mathtt{int} \qquad \text{(T-Var)}\ \ \Gamma, x : \mathtt{T} \vdash x : \mathtt{T} \qquad \text{(T-Op)}\ \ \frac{\Gamma \vdash e : \mathtt{int} \quad \Gamma \vdash e' : \mathtt{int}}{\Gamma \vdash e\ \mathtt{op}\ e' : \mathtt{int}}$$

$$\text{(T-Seq)}\ \ \frac{(\Gamma \vdash e_i : \mathtt{T}_i)^{i \in 1..n}}{\Gamma \vdash e_1, \ldots, e_n : \mathtt{T}_1, \ldots, \mathtt{T}_n}$$

Programs:

$$\text{(T-Prog)}\ \ \frac{\begin{array}{c}\Gamma = \{\mathtt{C.m} : [\overline{\mathtt{C}'}, \overline{\mathtt{int}}] \mid \mathtt{C.m}(\overline{\mathtt{C}'}\ \overline{x}, \overline{\mathtt{int}}\ \overline{y}) = P_{\mathtt{C.m}} \in \Psi\} \\ \left(\Gamma, this : \mathtt{C}, \overline{x} : \overline{\mathtt{C}'}, \overline{y} : \overline{\mathtt{int}}, t : \mathtt{Thread}, u : \mathtt{Obj}; u \vdash_t P_{\mathtt{C.m}} : \ell_{\mathtt{C.m}}\right)^{\mathtt{C.m}(\overline{\mathtt{C}'}\ \overline{x}, \overline{\mathtt{int}}\ \overline{y}) = P_{\mathtt{C.m}} \in \Psi} \\ \mathscr{L} = \bigcup_{\mathtt{C.m}(\overline{\mathtt{C}'}\ \overline{x}, \overline{\mathtt{int}}\ \overline{y}) = P_{\mathtt{C.m}} \in \Psi} \{\mathtt{f}_{\mathtt{C.m}}(t, z, this, \overline{x}) = \ell_{\mathtt{C.m}}\} \\ \Gamma, t : \mathtt{Thread}, u : \mathtt{Obj}; u \vdash_t \Psi.main : \ell\end{array}}{\Gamma \vdash \Psi : (\mathscr{L}, \ell)}$$

Fig. 1. The type system (we assume a function name $\mathtt{f}_{\mathtt{C.m}}$ for every method name $\mathtt{m}$ of class $\mathtt{C}$).

Let also

$$(x_1 \cdots x_n \cdot x_{n+1})^t \stackrel{def}{=} \begin{cases} 0 & \text{if } n = 0 \\ (x_1 \cdots x_n)^t & \text{if } x_{n+1} \in \{x_1, \cdots, x_n\} \\ (x_1 \cdots x_n)^t \& (x_n, x_{n+1})_t & \text{otherwise} \end{cases}$$

The static semantics has three judgments:

- $\Gamma \vdash e : \mathtt{T}$ – the expression e has type $\mathtt{T}$ in Γ;
- $\Gamma;\sigma \vdash_t P : \ell$ – the thread P with name t has lam ℓ in $\Gamma;\sigma$. We assume that the class of t is a predefined class `Thread` and Γ contains the bindings $t : \mathtt{Thread}$;
- $\Gamma \vdash \Psi : (\mathscr{L}, \ell)$ the program Ψ has lam program $(\mathscr{L}, \ell)$ in Γ.

In addition to `Thread`, the static semantics also uses another predefined class, called `Obj`. The objects of this class are used for typing new threads (*cf.* rule (T-Par) in Fig. 1; this expedient allows us to have a simpler rule for typing function invocations.

Few notational conventions are in order. When $\Psi = (\mathtt{C}_1\ \mathscr{D}_1, \cdots, \mathtt{C}_n\ \mathscr{D}_n, P)$, we let $\Psi.main = P$ and we write $\mathtt{C}_i.\mathtt{m}(\overline{\mathtt{T}}\ \overline{x}) = P_{\mathtt{C}_i.\mathtt{m}} \in \Psi$ whenever $\mathscr{D}_i$ contains $\mathtt{m}(\overline{T}\ \overline{x}) = P_{\mathtt{C}_i.\mathtt{m}}$.

The type system is defined in Fig. 1. A few key rules are discussed. Rule (T-Zero) types the process 0 in a thread t that has locked the objects in σ in (inverse) order. The lam is the conjunction of dependencies in σ with thread t – *cf.* notation $(\sigma)^t$. Rule (T-Sync) types the critical section P with a sequence of locks extended with x. Rule (T-Par) types a parallel composition of processes by collecting the lams of the components. We use a new name t' for the spawned thread and a new name u' for the object that is already grabbed by t'. This object has class `Obj`. (According to our convention, when we write $\Gamma, t' : \mathtt{Thread}, u' : \mathtt{Obj}$ we assume that $t', u' \notin dom(\Gamma)$.) Rule (T-Call) types a method name invocation in terms of a (lam) function invocation and constrains the types of actual arguments to match with the types of formal parameters in the method declaration. The arguments of the lam function invocation are extended with the thread name of the caller and the name of the last object locked by it (and not yet released). In addition we also conjunct the dependencies $(\sigma \cdot x)^t$ created by the caller because the method invocation run in a thread that holds those locks. This rule must be read in conjunction

with (T-Prog). (T-Prog) associates a lam function to every method – *c.f.* its second and third premises. The body of the lam function is defined by typing the body of the method in the hypothetical context where the thread only owns one lock, which is given as argument of the lam function. Of course, this is not the general case: a method may be invoked by a thread that grabs many locks. To address the general case we record the last object synchronized by the thread and build the dependencies of the invocation starting from it. It turns out that taking these dependencies in conjunction with those of the caller – see rule (T-Call) – does suffice for keeping track of the overall dependencies (see Proposition A.1). Because of this technicality, we require that every thread, even those that are just created, locks at least one object.

Example 4.1. Let us show the typing of the method `setTable` in the Introduction. Let

$$\begin{array}{rcl}\Gamma & = & \mathtt{C}.\mathtt{setTable}:[\mathtt{C},\mathtt{C},\mathtt{int}],\mathit{this}:\mathtt{C},x:\mathtt{C},y:\mathtt{C},n:\mathtt{int},t:\mathtt{Thread},u:\mathtt{Obj}\\ P & = & \mathtt{sync}(y)\{\,\mathtt{sync}(x)\{\,0\,\}\,\}\\ Q & = & \mathtt{fork}\{\mathtt{sync}(x)\{\,\mathtt{sync}(z)\{\,0\,\}\,\}\}\,z.\mathtt{setTable}(z,y,n-1)\end{array}$$

Then

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma,z:\mathtt{C};u\cdot y\cdot x\vdash_t 0:\ell_1 \quad \ell_1=(u,y)_t\&(y,x)_t}{\Gamma,z:\mathtt{C};u\cdot y\vdash_t \mathtt{sync}(x)\{\,0\,\}:\ell_1}}{\Gamma,z:\mathtt{C};u\vdash_t P:\ell_1} \qquad \dfrac{(*)}{\Gamma,z:\mathtt{C};u\vdash_t Q:\ell_2}}{\Gamma,z:\mathtt{C};u\vdash_t \mathtt{if}\ n=0\ \mathtt{then}\ P\ \mathtt{else}\ Q:\ell_1+\ell_2}}{\Gamma;u\vdash_t (\mathtt{new}\ \mathtt{C}\ z)\ (\mathtt{if}\ n=0\ \mathtt{then}\ P\ \mathtt{else}\ Q):(\nu\ z)\ (\ell_1+\ell_2)}$$

where $(*)$ are the two proof trees

$$\dfrac{\cdots}{\Gamma,z:\mathtt{C};u\vdash_t z.\mathtt{setTable}(z,y,n-1):\ell_2'}$$

and

$$\dfrac{\cdots}{\Gamma,z:\mathtt{C},t':\mathtt{Thread},u':\mathtt{Obj};w'\vdash_{t'}\mathtt{sync}(x)\{\,\mathtt{sync}(z)\{\,0\,\}\,\}:\ell_2''}$$

(as an exercise, the reader may try to complete them). By rule (T-Par), we derive $\ell_2=\ell_2'\&(\nu\ t',u')\ \ell_2''$. After completing the proof tree, one obtains the lam function

$$\begin{array}{l}\mathtt{setTable}(t,u,\mathit{this},x,y)=\\ \quad(\nu\ z,t',u')\ \Big((u,y)_t\&(y,x)_t+(u',x)_{t'}\&(x,z)_{t'}\ \&\ \mathtt{setTable}(t,u,x,z,y)\Big)\end{array}$$

which has two additional arguments with respect to the one in the Introduction: u and *this*. In fact, these arguments, as well as u', do not play any role in the analysis of circularities of $\mathtt{setTable}(t,u,\mathit{this},x,y)$.

The following theorem states the soundness of our type system. The technical details of the proof are reported in the Appendix.

Theorem 4.2. *Let* $\Gamma\vdash\big(c_1\ \mathscr{D}_1,\cdots,c_n\ \mathscr{D}_n,P\big):\big(\mathscr{L},\ell\big)$. *If* $\big(\mathscr{L},\ell\big)$ *has no circularity then* $\big(c_1\ \mathscr{D}_1,\cdots,c_n\ \mathscr{D}_n,P\big)$ *is deadlock-free.*

Example 4.3. Let us verify whether the process $x.\mathtt{setTable}(x,x,n)$ is deadlock-free. The lam function associated by the type system is detailed in Example 4.1. The interpretation function $I_{\mathscr{L}}(\mathtt{setTable})$ is computed as follows:

$$\begin{array}{l}I^{(0)}_{\mathscr{L}}(\mathtt{setTable})\ =\ \{\varnothing\}\\ I^{(1)}_{\mathscr{L}}(\mathtt{setTable})\ =\ \{\,\{(u,y)_t,(y,x)_t,(u,x)_t\}\,\}\\ I^{(2)}_{\mathscr{L}}(\mathtt{setTable})\ =\ \{\,\{(u,y)_t,(y,x)_t,(u,x)_t\},\ \{(u,y)_t\}\,\}\,.\end{array}$$

Since $I^{(2)}_{\mathscr{L}}=I_{\mathscr{L}}$, we are reduced to compute $I^{(2)}_{\mathscr{L}}(\mathtt{setTable}(t,u,x,x,x))$. That is

$$\{\,\{(u,y)_t,(y,x)_t,(u,x)_t\},\ \{(u,y)_t\}\,\}\{^x/_y\}\ =\ \{\,\{(u,x)_t,(x,x)_t\},\ \{(u,x)_t\}\,\}$$

which has no circularity, therefore the process $x.\mathtt{setTable}(x,x,n)$ is deadlock-free.

It is interesting to verify whether the process $x.\mathtt{setTableD}(x,x,n)$ is deadlock-free, where $\mathtt{setTableD}$ is the method having philosophers with symmetric strategies:

```
setTableD(C x,C y,int n) = (new C z)(  if (n=0) then sync(x){ sync(y){ 0 } }
                                       else fork { sync(x){ sync(z){ 0 } } }
                                       z.setTableD(z,y,n-1)
                                    )
```

In this case $I_{\mathscr{L}}(\mathtt{setTableD}) = \{\{(u,x)_t, (x,y)_t, (u,y)_t\},\ \{(x,y)_{\checkmark}, (u,y)_t\}\}$. It is easy to verify that $I_{\mathscr{L}}(\mathtt{setTableD}(t,u,x,x,x))$ has a circularity, therefore the process $x.\mathtt{setTableD}(x,x,n)$ may have (and actually has) a deadlock.

To conclude, we observe that the deadlock freedom property guaranteed by Theorem 4.2 is stronger than the one defined in Definition 3.2 that admits a subset of processes to deadlock provided that some other process progresses. In fact, as usual [16,17], our type system guarantees that no subset of processes deadlocks.

5. The extension of inheritance

The basic language in Section 3 does not admit derivations of classes from other classes. As a consequence, when a method is invoked, it is possible to uniquely locate the method definition. This feature is expressed in the type system in Fig. 1 by the strict correspondence between types of arguments and types of formal parameters (e.g. the constraint $\Gamma \vdash \overline{u} : \overline{\mathtt{C}}$ in the type rule (T-Call)). On the contrary, object-oriented programming languages admit subclasses, thereby allowing redefinitions of methods in the subclasses. In this context, locating a method definition is performed at run-time through a mechanism called dynamic dispatch that uses the actual value of the called object.

For example, let $\mathtt{C}$ be a class with a method $\mathtt{foo}$ that is overridden in the subclasses $\mathtt{C}_1$ and $\mathtt{C}_2$. Then, consider a method with arguments x, y, z of types $\mathtt{C}$, $\mathtt{C}$, and $\mathtt{int}$, respectively. If this snippet

$$\mathtt{if}\ z\ \mathtt{then}\ x.\mathtt{foo}()\ \mathtt{else}\ y.\mathtt{foo}()$$

is part of its body and the method is invoked with objects of types $\mathtt{C}_1$ and $\mathtt{C}_2$, the method $\mathtt{foo}$ of $\mathtt{C}_1$ or of $\mathtt{C}_2$ will be performed according to the value of z. Clearly, dispatching an invocation to a definition rather than another may change the overall result of the deadlock analysis.

In order to study the impact of inheritance on the deadlock analysis, we extend the language in Section 3 as follows. Let a *program* be a tuple

$$\left(D_1\ \mathscr{D}_1, \cdots, D_n\ \mathscr{D}_n, P\right)$$

where every $\mathscr{D}_i$ is as in Section 3 and D is a term of the following grammar

$$D\ ::=\ \mathtt{C}\ \mid\ \mathtt{C}\ \mathtt{extends}\ \mathtt{C}$$

This extension carries a subtyping relation $<:$ that is reflexive, transitive and contains $\mathtt{C} <: \mathtt{C}'$ if $\mathtt{C}\ \mathtt{extends}\ \mathtt{C}'$ is in the program. Processes are extended with a more flexible object creation operation:

$$P\ ::=\ \cdots\ \mid\ (\mathtt{new}\ \mathtt{C}_1, \cdots, \mathtt{C}_m\ x)\ P$$

meaning that the type of x may be one of $\mathtt{C}_1, \cdots, \mathtt{C}_m$ (it is non-deterministic) and it is intended that there is $\mathtt{C}$ such that $\mathtt{C}_i <: \mathtt{C}$ for every $\mathtt{C}_i$. This apparently exotic operation corresponds to the pattern

```
C x ;
if e1 x = new C1() ;
else if e2 x = new C2() ;
...
else x = new Cm() ;
```

that we cannot use because our basic language does not feature assignments (see Section 6).

The static semantics of the foregoing extensions uses (finite) sets of object names $\{\mathtt{C}_1, \cdots, \mathtt{C}_m\}$ such that there is a $\mathtt{C}$ with $\mathtt{C}_i <: \mathtt{C}$ for every $\mathtt{C}_i$. Let $\mathscr{C}, \mathscr{C}', \cdots$ range over these sets. Let also environments Γ bind object names to sets $\mathscr{C}$. The typing rules become ($+_{0\le i\le n}\ \ell_i$ is an abbreviation for $\ell_1 + \cdots + \ell_n$):

(T-Subt)
$$\frac{\mathtt{C}\ [\mathtt{extends}\ \mathtt{C}']\{\cdots, \mathtt{m}(\overline{\mathtt{C}''}\ \overline{x}, \overline{\mathtt{int}}\ \overline{y}) = P, \cdots\} \quad \textit{is in the program} \qquad \mathtt{E} <: \mathtt{C} \qquad \overline{\mathtt{E}'} <: \overline{\mathtt{C}''}}{\Gamma \vdash \mathtt{E.m} : [\overline{\mathtt{E}'}, \overline{\mathtt{int}}]}$$

(T-New-Inh)
$$\frac{\Gamma, x{:}\mathscr{C}; \sigma \vdash_t P : \ell}{\Gamma \vdash_t (\mathtt{new}\ \mathscr{C}\ x)\ P : (\nu\ x)\ \ell}$$

(T-Call-Inh)
$$\frac{\Gamma(z) = \mathscr{C} \quad \Gamma \vdash \overline{u} : \overline{\mathscr{C}'} \qquad \left(\Gamma \vdash \mathtt{C.m} : [\overline{\mathtt{C}}, \overline{\mathtt{int}}]\right)^{\mathtt{C} \in \mathscr{C}, \overline{\mathtt{C}} \in \overline{\mathscr{C}}} \qquad \Gamma \vdash \overline{e} : \overline{\mathtt{int}}}{\Gamma; \sigma \cdot x \vdash_t z.\mathtt{m}(\overline{u}, \overline{e}) : \left(+_{\mathtt{C} \in \mathscr{C}, \overline{\mathtt{C}} \in \overline{\mathscr{C}}}\ \mathtt{f}_{\mathtt{C.m}[\overline{\mathtt{C}}]}(t, x, z, \overline{u})\right) \& (\sigma \cdot x)^t}$$

In the rule (T-Subt), the text in $[\cdots]$ is optional: the rule applies both to class and to subclass definitions. Our analyzer associates a lam function $f_{\mathtt{C.m}[\overline{\mathtt{C}'}]}$ to every method $\mathtt{m}$ such that $\Gamma \vdash \mathtt{C.m} : [\overline{\mathtt{C}'}, \overline{\mathtt{int}}]$. As in the system of Fig. 1, the body ℓ of this function is defined by the judgment (it is assumed that $\mathtt{m}(\overline{\mathtt{C}'}\ \overline{x}, \overline{\mathtt{int}}\ \overline{y}) = P$ belongs to a class $\mathtt{C}$):

$$\Gamma, \overline{x} : \overline{\{\mathtt{C}'\}}, \overline{y} : \overline{\mathtt{int}}, t : \{\mathtt{Thread}\}, u : \{\mathtt{Obj}\}; u \vdash_t P : \ell$$

(T-Call-Inh) is the critical rule. Since statically we do not know what method will be actually executed at runtime, we return the disjunctions of the lams of every possible method, in conjunction with the dependencies $(\sigma \cdot x)^t$ created by the caller. We illustrate the point by discussing an example.

Example 5.1. Consider the following program:

```
C1 { m(C x, C y) = sync(x){ sync(y){ 0 } } } ;
C2 extends C1 { m(C x, Cy) = sync(y){ sync(x){ 0 } } } ;
new C x, C y, {C1,C2} u. fork{ u.m(x,y) } sync(x){ sync(y){ 0 } }
```

In this case, the type of u may be either `C1` or `C2`, therefore the method `m` that will be actually invoked will be either `C1.m` or `C2.m`. Therefore we may have a deadlock because the main thread locks x and then y, while the spawned thread `C2.m` locks y and then x. There is no deadlock if the type of u is `C1`.

By rule (T-Call-Inh) and (T-Par) we derive the following lam for main:

$$\ell = \quad ((u, x)_t \& (x, y)_t) \quad \& \quad (\nu\ t')\ (((u, x)_{t'} \& (x, y)_{t'}) + ((u, y)_{t'} \& (y, x)_{t'}))$$

It is easy to verify that $I^{\perp}(\ell)$ has a circularity.

The foregoing management of inheritance is generic. In several cases, it is possible to have a more specific solution using explicit typings. In particular, whenever the run-time types of objects are known, it is possible to attach the right type to method invocations and apply concrete type reasoning. `JaDA` uses this technique starting the analysis from the main program.

6. Other extensions

In order to deliver initial assessments of our technique, we have extended the language in Sections 3 and 5 to cover many features of `Java`. The resulting tool, called `JaDA` [10,19], has been prototyped. Overall, the attitude we had in the development of `JaDA` was to stick to solutions that can be easily integrated with the type system of the core language (and possibly simple to implement). In many places, we might have taken more refined solutions (see, for instance, the following paragraph about recursive types), with the danger of further delaying the release of the tool because of issues that are orthogonal to our technique.

In this section we overview two relevant extensions – assignments and recursive data types – and discuss our solutions in `JaDA`; the details of other extensions (e.g. static members, exceptions, and arrays) can be found in Garcia's PhD thesis [9]. The comparison of `JaDA` with other deadlock analysis tools is reported in Section 7.

Assignments. In the language of Sections 3 and 5 it is not possible to store and retrieve values. These operations are usually achieved by admitting objects with *fields* and the operations of read and write of fields. E.g. the syntax of processes and expressions is extended as follows

$$\begin{array}{rcll} P & ::= & \cdots & \mid \quad \mathtt{f} = e.\ P \\ e & ::= & \cdots & \mid \quad x.\mathtt{m}(\overline{e}) \end{array}$$

(method invocations may return values).

Assignments impact the type system in a significative way because field values change during the execution and these changes may be unpredictable because of concurrency. In addition, since fields may store other objects and may be method's arguments, it is possible to create aliases. Overall, tracing the state changes becomes quite hard. The type system of JaDA overcomes these issues by

- *checking data races* of parallel threads (for example, one thread reading an object and the other one writing on the object). In particular, the environment Γ binds fields to annotations in $\{-, \mathtt{r}, \mathtt{w}\}$, with $-$ meaning no access, $\mathtt{r}$ meaning read access, $\mathtt{w}$ meaning write (and possible read) access. The set $\{-, \mathtt{r}, \mathtt{w}\}$ is a total order ($- < \mathtt{r} < \mathtt{w}$) and we use the operation of least upper-bound $\sqcup$.
- *analyzing aliases* created by copying mechanisms. This is implemented by decoupling the connection between object names and values in the environment. E.g. we introduce *object pointers*, noted a, b, $\cdots$ (that include the pointer $\top$) and we let the environment to bind object names to object pointers and object pointer to pairs $([\mathtt{f}_1^{\mathtt{h}_1} : a_1, \cdots, \mathtt{f}_m^{\mathtt{h}_m} : a_m], \mathscr{C})$, where $\mathtt{h}_i$ are in $\{-, \mathtt{r}, \mathtt{w}\}$ and a_i may also be $\top$. The pointer $\top$ represents a record with fields that are tagged $\mathtt{w}$ and with values $\top$.

In order to trace data races, we use a companion type system $\Gamma \vdash^e P : \Gamma'$ that stores in Γ' the type of accesses performed by P to fields of objects in Γ (we assume that accesses in Γ are all set to $-$; when this is not the case, we use the function $\Gamma\uparrow^-$ that rewrites accesses into $-$). The *merge operation* on environments, noted $\Gamma \| \Gamma'$, returns an environment where fields have the least upper-bounds of accesses in Γ and Γ' while values are set to $\top$ when accesses are conflictual (e.g. either $\mathtt{r}$-$\mathtt{w}$ or $\mathtt{w}$-$\mathtt{w}$). For instance, let

$$\Gamma = a \mapsto ([\mathtt{f}^{\mathtt{r}} : b, \mathtt{g}^{-} : b'], \mathscr{C}) \qquad \Gamma' = a \mapsto ([\mathtt{f}^{\mathtt{w}} : c, \mathtt{g}^{\mathtt{w}} : d], \mathscr{C})$$

then $\Gamma \| \Gamma' = a \mapsto ([\mathtt{f}^{\mathtt{w}} : \top, \mathtt{g}^{\mathtt{w}} : d], \mathscr{C})$, that is

1. the field $\mathtt{f}$ has access $\mathtt{w}$ because this is the least upper bound of the two accesses; additionally the resulting value of $\mathtt{f}$ is $\top$ because we do not know whether the read access in Γ will read b or c;
2. the field $\mathtt{g}$ has access $\mathtt{w}$ because Γ does not accesses to it while Γ' writes it; additionally the resulting value of $\mathtt{g}$ is d because this is the value written by Γ'.

Updates are recorded by means of judgments that also return environments. Namely, we use $\Gamma; \sigma \vdash_t P : \ell, \Gamma'$, where Γ' is the environment Γ updated by P. We also use the operation $\Gamma[\Gamma']$ that returns the environment Γ updated according to what is prescribed in Γ'. The typing rule for the spawn of new threads is

(T-Par-Eff)

$$\frac{\Gamma\uparrow^- \vdash^e P : \Gamma_P \qquad \Gamma\uparrow^- \vdash^e P' : \Gamma_{P'} \qquad \Gamma[\Gamma_P \| \Gamma_{P'}]; \sigma \vdash_t P : \ell,\ \Gamma' \qquad \Gamma[\Gamma_P \| \Gamma_{P'}], u' : \mathtt{Obj}; u' \vdash_{t'} P' : \ell',\ \Gamma''}{\Gamma \vdash_t \mathtt{fork}\{P'\}\ P : \ell \,\&\, (\nu\, t', u')\ \ell',\ \Gamma' \| \Gamma''}$$

We notice that P and P' are verified in an environment that already record the possible data races (a priori, P might start after P' or conversely). We also notice that the returned environment in the conclusion is the merge of the returned environments in the premises.

The presence of assignments also impacts on lams. First of all, in order to model object values, arguments of lam functions may also be records. This extension is not problematic with respect to Theorem 2.8 because our records are finite trees with fixed structures (this is also the case when object have recursive types, see below). In addition, the special name $\top$ influences the definition of $\mathtt{R}^+$ in Section 2. In fact, this name, being a place holder for possible object names, may be replaced by every name when transitive closure is computed, thereby overapproximating the set $\mathtt{R}^+$ with more dependencies (and circularities). For example, $(\top, a)_t \& (a, b)_{t'}$ is a circularity (because $\top$ may be replaced by b), while $(\top, a)_t \& (a, b)_t$ is not a circularity (because, in case, this may be a reentrant thread).

Recursive types. In JaDA, the analysis of objects with recursive types is over-approximated by using finite representations. In particular, a record of a *recursive type* is built by unfolding the type (exactly) up to those nodes containing a name of a class already present in the tree. Nodes inside the tree have values that are object names, nodes in the leaves whose type

Table 1
Comparison with different deadlock detection tools. The inner cells show the number of deadlocks detected by each tool. The output labeled "(*)" are related to modified versions of the original programs: see the text.

Benchmarks	LOC, #Threads	Deadlock	Static		Hybrid	Dynamic	Commercial
			JaDA[tm]	Chord[tm]	GoodLock[tm]	Sherlock[tm]	ThreadSafe[tm]
Sor	1274, 5	yes	1 [135s]	1 [210s]	7 [4s]	1 [39s]	4 [435s]
RayTracer(*)	1292, 5	no	0 [155s]	0 [223s]	8 [2s]	2 [30s]	0 [502s]
MolDyn (*)	1351, 5	no	0 [110s]	0 [191s]	6 [5s]	1 [49s]	0 [423s]
MonteCarlo (*)	3619, 4	no	0 [231s]	0 [342s]	23 [5s]	2 [102s]	0 [821s]
BuildNetworkN	40, N+1	yes	3 [8s]	0 [50s]			0 [50s]
PhilosophersN	60, N+1	yes	3 [12s]	0 [51s]			0 [51s]
ThreadArraysN	23, N+1	yes	1 [6s]	1 [40s]			1 [40s]
ThreadArraysJoinsN	37, N+1	yes	1 [6s]	1 [41s]			0 [41s]

is not recursive are marked `int` (a primitive type value); leaves of a recursive type have values that are names subscribed by a `$`; the name corresponds to the nodes of the classes that are already present in the tree.

For instance, if `C` is a class whose type is $[\texttt{val} : \texttt{Thread}, \texttt{next} : \texttt{C}]$ (a list of threads) then, in correspondence of a `new C` instruction, we produce an environment $r_{\$} \mapsto [\texttt{val} : (a[\,], \texttt{Thread}), \texttt{next} : r_{\$}]$. Lists like the foregoing one are therefore abstracted by a single node. The analyzer replaces names indexed by `$` with $\top$ and uses the same approximation described in previous paragraph. In particular, if the types of $r_{\$}$ and of $r'_{\$}$ are the same, a term like $(r_{\$}, a)_t \& (b, r'_{\$})_{t'}$ is a circularity because it corresponds to $(\top, a)_t \& (b, \top)_{t'}$.

7. JaDA and its assessment

Since `JaDA` covers many features of `Java`, it has been possible to deliver an initial assessment of it with respect to existing deadlock analysis tools. In particular, we have considered tools using different techniques: `Chord` for static analysis [23], `Sherlock` for dynamic analysis [6], and `GoodLock` for hybrid analysis [3]. We have also considered a commercial tool, `ThreadSafe`[3] [2]. Out of these tools, we were able to install and effectively test only two of them – `Chord` and `ThreadSafe` – because the other twos have been discontinued; the results corresponding to `GoodLock` and `Sherlock` come from [6]. We also had problems in testing `Chord` with some of the examples in the benchmarks, perhaps due to some misconfigurations, that we were not able to solve because `Chord` has been also discontinued.

The analyzed programs exhibit a variety of sharing patterns. The source of all benchmarks in Table 1 is available either at [6,23] or in the `JaDA-deadlocks` repository.[4] Since the current release of `JaDA` does not completely cover `Java`, in order to gain preliminary experience, we modified the Java libraries and the multithreaded server programs of RayTracer, MolDyn and MonteCarlo (labeled with "(*)" in the Table 1) and implemented them in our system. This required little programming overhead; in particular, we removed volatile variables, avoided the use of `Runnable` interfaces for creating threads, and reduced the invocations of native methods involved in I/O operations. For every program, we give the lines of code (LOC), the number #Threads of threads explicitly created (in the second and third block this number depends on the argument N). We also state whether the program under examination has a deadlock or not and the time in seconds (tm) each tool took to perform the analysis. The times for `GoodLock` and `Sherlock` were taken from the literature [6].

Here are our remarks. The first block of programs belongs to a well known group used as benchmarks for several `Java` analysis tools; the second block corresponds to examples designed to test `JaDA` against complex deadlock scenarios. First of all, at least for the benchmark codes in Table 1, `JaDA` is the unique tool that never returns false negatives (in general, `JaDA` may return false positives). `Chord` and `ThreadSafe` are unsound because they return false negatives (see the second block). The execution time of the tools are similar (`JaDA` appears more efficient), except for `GoodLock` and `Sherlock`, which appear however much less precise (they return a lot of false positives). As regards the second block, we observe that `JaDA` returns few deadlocks, which do not depend from N. This is because our analysis is symbolic and does not consider numeric values (most of the deadlock are considered "to be similar").

We retain that the results of Table 1 are encouraging and solicit further programming efforts for covering `Java` features that have not been considered so far, as well as more elaborate comparisons. This is left for future work.

It is also worth to observe that `JaDA` actually addresses the `Java` bytecode (we compile `Java` programs into bytecode and analyze it). This means that `JaDA` also analyzes every programming language that is compiled into `Java` bytecode, such as `Scala` [24]. As regards `Scala`, to the best of our knowledge, `JaDA` is the first static deadlock analysis tool.

[3] http://www.contemplateltd.com/threadsafe.
[4] https://github.com/abelunibo/Java-Deadlocks.

8. Remarks about the analysis technique

The deadlock analysis technique presented in this paper is lightweight because it is compact, intelligible and theoretically manageable. The technique is also very powerful because we can successfully verify processes like `buildTable` and its variant where every philosopher has a symmetric strategy. However, there are processes for which our technique of collecting dependencies is too rough (and we get false positives).

One example is

$$\texttt{fork}\{sync(x)\{\, sync(z)\{\, sync(y)\{\, 0\,\}\,\}\,\}\}\;\; sync(x)\{\, sync(y)\{\, sync(z)\{\, 0\,\}\,\}\,\}\,.$$

This process has two threads: the first one locks x and then z and y in order; the second one locks x and then grabs y and z in order. Since the two threads initially compete on the object x, they will be executed *in sequence* and no deadlock will ever occur. However, if we compute the dependencies, we obtain

$$(x \cdot z \cdot y)^t \,\&\, (x \cdot y \cdot z)^s$$

that is equal to $((x,z)_t \& (z,y)_t) \,\&\, ((x,y)_s \& (y,z)_s)$ where the reader may easily recognize the circularity $(z,y)_t \& (y,z)_s$. This inaccuracy follows by the fact that our technique does not record the dependencies between threads and their state (of locks) when the spawns occur: this is the price we pay to simplicity. In [10] we overcome this issue by associating line codes to symbolic names and dependencies. Then, when a circularity is found, we can exhibit an abstract witness computation that, at least in the simple cases as the above one, can be used to manually verify whether the circularity is a false positive or not.

Another problematic process is $\texttt{m(C}\ x, \texttt{C}\ y, \texttt{int}\ n\texttt{)}$ in Example 3.3. This process never deadlocks when $n \leq 0$. However, since our technique drops integer values, it always return a circularity (this is a correct result when $n > 0$ and it is a false positive otherwise). To cope with these cases, it suffices to complement our analysis with standard techniques of data-flow analysis and abstract evaluation of expressions.

9. Related works and conclusions

We have defined a simple technique for detecting deadlocks in object-oriented programs. This technique uses an extension of the lam model in order to cope with reentrant locks, a standard feature of object-oriented programs. We have specified an algorithm for verifying the absence of circularities in extended lams and we have applied this model to a simple concurrent object-oriented calculus that is intended to be a minimal concurrent subset of `Java`. We demonstrate the effectivity of our analysis by applying it to `Java`. In particular, we discuss a prototype verifier, called `JaDA`, and deliver initial assessments that seem quite encouraging.

The lam model has been introduced and studied in [12] for detecting deadlocks of an object-oriented language with futures (and no lock and lock reentrancy) [13], but the extension discussed in this paper is new as well as the algorithm for the circularity of lams. We have prototyped this algorithm in `JaDA`, where we use it for the deadlock analysis of `Java` [10]. As we discussed in the Introduction, the model has been also applied to process calculi [11,17].

Several techniques have been developed for the deadlock detection of concurrent object-oriented languages. The technique [2] uses a data-flow analysis that constructs an execution flow graph and searches for cycles within this graph. Some heuristics are used to remove likely false positives. No alias analysis to resolve object identity across method calls is attempted. This analysis is performed in [6,23], which can detect reentrance on restricted cases, such as when lock expressions concern local variables (the reentrance of formal parameters, as in `buildTable(x,x,0)` is not detected). The technique in [3] and its refinement [6] use a theory that is based on monitors. Therefore the technique is a runtime technique that tags each segment of the program reached by the execution flow and specifies the exact order of lock acquisitions. Thereafter, these segments are analyzed for detecting potential deadlocks that might occur because of different scheduler choices (than the current one). This kind of technique is partial because one might overlook sensible patterns of methods' arguments (*cf.* `buildTable`, for instance). A powerful static techniques that is based on abstract interpretation is `SACO` [8]. `SACO` has been developed for `ABS` [14], an object-oriented language with a concurrent model different from `Java`. A comparison between `SACO` and a tool using a technique similar to the one in this paper can be found in [13].

Our future work includes the analysis of concurrent features of object-oriented calculi that have not been studied yet. A relevant one is thread coordination, which is usually expressed by the methods `wait` and `notify` (and `notifyAll`). These methods modify the scheduling of processes: the thread executing `wait`(x) is suspended, and the corresponding lock on x is released; the thread executing `notify`(x) wakes up one thread suspended on x, which will attempt again to grab x. A simple deadlock in programs with `wait` and `notify` is when the `wait` operation is either mismatched or *happens-after* the matching notification. Initial steps toward the deadlock analysis of coordination primitives can be found in [20], where we design a behavioral type system for coordination primitives. This system resorts to simple object protocols – called *usages* – to determine whether objects are used reliably, so as to guarantee deadlock freedom. In turn, this reliability is reduced to a reachability problem in a Petri net that, in the case of coordination primitives, have also inhibitor arcs. While the technique in [20] seems complementary to our extended lam model, the actual integration must still be studied.

A research direction that is close to the above one is the analysis of concurrent libraries, such as `java.util.concurrent`. This issue has several cases, we discuss the most relevant ones. One case is when a library function can be implemented in `Java` (bytecode) using features that are already covered [9] or that can be covered by our analysis (*e.g.* coordination primitives). In this case, we derive the corresponding lam either in an automatic way or in a semi-automatic way, by providing the missing judgments, and use *explicit typing* in every place where the function is used. A second case is when a function is written in native code. In this case, we compute all the dependencies by parsing the code manually and type the function with the resulting lam (and again use explicit typing). Another case is when the function uses synchronization primitives that rely on conditional expressions, such as conditional await. In this case, a reasonable solution is integrating our technique with some axiomatic formalism using pre- and post-conditions. These axiomatic formalisms are also very powerful for tackling assignments and pointers (see Section 6), *cf.* separation logic [26]. The study of this integration and the possible definition of an hybrid solution for the deadlock analysis of `Java`-like languages is matter of future research.

Further work will be also necessary for the development of `JaDA`: while the results of Table 1 are encouraging, they also solicit further programming efforts for covering `Java` features that have not been considered so far. This should allow us to address the benchmarks labeled (*) without any programming overhead and possibly other benchmarks as in [23].

Acknowledgements

We thank Abel Garcia and Elena Giachino for the fruitful discussions and useful comments. In particular, Abel has been the main developer of `JaDA`. We also thank the anonymous referees for reading the paper carefully and for their many insightful comments and suggestions.

Appendix A. Proof of Theorem 4.2

We first need to extend the typing to states. Therefore we extend the type system in Fig. 1 with the following two rules:

$$\frac{\Gamma, t : \mathtt{Thread}; \sigma \vdash_t P : \ell}{\Gamma \vdash P \bullet \sigma : (\nu\, t)\, \ell}\ (\textsc{T-Thread}) \qquad \frac{\left(\Gamma \vdash P_i \bullet \sigma_i : \ell_i\right)^{i \in 1..n}}{\Gamma \vdash \prod_{i\in 1..n} P_i \bullet \sigma_i : \&_{i\in 1..n}\ell_i}\ (\textsc{T-State})$$

Theorem 4.2 is a consequence of the following statements. The first two are obtained by a standard induction on the proof-tree of the judgment in the premise.

Proposition A.1. *If* $\Gamma; x \vdash_t P : \ell$ *and names in* σ *belong to* $dom(\Gamma)$ *then there is* ℓ' *such that* $\Gamma; \sigma \cdot x \vdash_t P : \ell'$ *and* $I_{\mathscr{L}}(\ell') \Subset I_{\mathscr{L}}(\ell \& (\sigma \cdot x)^t)$.

Let $\Gamma\{^{\overline{u}}/_{\overline{x}}\}$ be defined as follows:

$$(\Gamma\{^{u_1,\cdots,u_n}/_{x_1,\cdots,x_n}\})(z) = \begin{cases} \Gamma(x_i) & \text{if } z = u_i,\ 1 \le i \le n \\ \Gamma(z) & \text{if } z \notin u_1, \cdots, u_n \text{ and } z \notin x_1, \cdots, x_n \end{cases}$$

In the above definition, the topmost alternative applies before the other one. Therefore, for example, $\Gamma\{^{x}/_{x}\} = \Gamma$ and $\Gamma\{^{u,v}/_{v,u}\}$ simply swaps the mappings of u and v, e.g. $\Gamma\{^{u,v}/_{v,u}\}(u) = \Gamma(v)$ and $\Gamma\{^{u,v}/_{v,u}\}(v) = \Gamma(u)$.

Lemma A.2 *(Substitution lemma). Let* $\Gamma; \sigma \vdash_t P : \ell$ *and* $t' = t\{^{\overline{u}}/_{\overline{x}}\}$. *Then* $\Gamma\{^{\overline{u}}/_{\overline{x}}\}; \sigma\{^{\overline{u}}/_{\overline{x}}\} \vdash_{t'} P\{^{\overline{u}}/_{\overline{x}}\} : \ell\{^{\overline{u}}/_{\overline{x}}\}$. *Similarly for* $\Gamma \vdash e : \mathtt{T}$.

Lemma A.3 *(Type preservation). Let* $\Gamma \vdash \left(C_1\ \mathscr{D}_1, \cdots, C_n\ \mathscr{D}_n, P\right) : \left(\mathscr{L}, \ell\right)$ *and* $P \bullet \varepsilon \longrightarrow^* \mathbb{P}'$. *Then there exist* ℓ' *and* Γ' *such that* $\Gamma + \Gamma' \vdash \mathbb{P}' : \ell'$ *and* $I_{\mathscr{L}}(\ell') \Subset I_{\mathscr{L}}(\ell)$.

Proof. This follows by induction on the derivation of $P \bullet \varepsilon \longrightarrow^* \mathbb{P}'$, with case analysis on the last rule used, let it be $\mathbb{P} \longrightarrow \mathbb{P}'$. The only non-trivial cases are (Sync) and (Call).

- Case (Sync): In this case, we have

$$\mathbb{P} = \mathtt{sync}(x)\{\ P'\ \} \bullet \sigma \mid \mathbb{P}'' \qquad x \notin \mathbb{P}'' \qquad \mathbb{P}' = P' \bullet \sigma \cdot x \mid \mathbb{P}''$$

$$\Gamma + \Gamma'' \vdash \mathbb{P} : \ell''$$

where $\mathbb{P}'' = \prod_{i\in 1..n} P_i \bullet \sigma_i$.

By the condition $\Gamma + \Gamma'' \vdash \mathbb{P} : \ell''$ and rules (T-State) and (T-Thread) we have

$$\Gamma + \Gamma'', t_0 : \mathtt{Thread}; \sigma \vdash_{t_0} \mathtt{sync}(x)\{\ P'\ \} : \ell_0 \tag{A.1}$$

$$\left(\Gamma + \Gamma'' \vdash P_i \bullet \sigma_i : \ell_i\right)^{i\in 1..n} \tag{A.2}$$

$$\ell'' = (\nu\ t_0)\ \ell_0 \,\&\, (\&_{i\in 1..n}\ell_i) \tag{A.3}$$

By using (T-Sync), from (A.1) we derive

$$\Gamma + \Gamma'', t_0 : \mathtt{Thread}; \sigma \cdot x \vdash_{t_0} P' : \ell_0$$

By (T-Thread) applied to the foregoing judgment we derive

$$\Gamma + \Gamma'' \vdash P' \bullet \sigma \cdot x : (\nu\ t_0)\ \ell_0$$

Then, by applying (T-State) to this last judgment and (A.2), we have the required result (with $\ell' = \ell$).

- Case (Call): We assume $\mathtt{C}\ \mathscr{D}$ is in the program and $\mathtt{m}(\overline{\mathtt{C}'}\ \overline{z}, \overline{\mathtt{int}}\ \overline{y}) = P_{\mathtt{C.m}}$ is in $\mathscr{D}$. We have

$$\mathbb{P} = x.\mathtt{m}(\overline{u}, \overline{e}) \bullet \sigma \mid \mathbb{P}'' \qquad [\![\overline{e}]\!] = \overline{v} \qquad \mathbb{P}' = P_{\mathtt{C.m}}\{^{x,\overline{u},\overline{v}}/_{this,\overline{y},\overline{z}}\} \bullet \sigma \mid \mathbb{P}''$$

$$\Gamma + \Gamma'' \vdash \mathbb{P} : \ell''$$

where $\mathbb{P}'' = \prod_{i\in 1..n} P_i \bullet \sigma_i$. By the condition $\Gamma + \Gamma'' \vdash \mathscr{T} : \ell''$ and rules (T-State) and (T-Thread) we have

$$\Gamma + \Gamma'', t_0 : \mathtt{Thread}; \sigma \vdash_{t_0} x.\mathtt{m}(\overline{u}, \overline{e}) : \ell_0 \tag{A.4}$$

$$\left(\Gamma + \Gamma'' \vdash P_i \bullet \sigma_i : \ell_i\right)^{i\in 1..n}$$

$$\ell'' = (\nu\ t_0)\ \ell_0 \,\&\, (\&_{i\in 1..n}\ell_i)$$

Let $\sigma = \sigma' \cdot x'$. By (T-Call) applied to (A.4) we have that $\ell_0 = \mathtt{f}_{\mathtt{C.m}}(t_0, x', x, \overline{u})\&(\sigma)^{t_0}$. By the hypothesis $\Gamma \vdash (\mathtt{C}_1\ \mathscr{D}_1, \cdots, \mathtt{C}_n\ \mathscr{D}_n, P) : (\mathscr{L}, \ell)$ and (T-Prog) we derive

$$\Gamma, this : \mathtt{C}, \overline{y} : \overline{\mathtt{C}'}, \overline{z}{:}\overline{\mathtt{int}}, t' : \mathtt{Thread}, z' : \mathtt{Thread}; z' \vdash_{t'} P_{\mathtt{C.m}} : \ell_{\mathtt{C.m}} \tag{A.5}$$

$$\mathtt{f}_{\mathtt{C.m}}(t', z', this, \overline{y}) = \ell_{\mathtt{C.m}} \in \mathscr{L}$$

By applying the Substitution Lemma A.2 to (A.5) we have

$$\Gamma''' = \left(\Gamma, this : \mathtt{C}, \overline{y} : \overline{\mathtt{C}'}, \overline{z} : \overline{\mathtt{int}}, t' : \mathtt{Thread}, x' : \mathtt{Thread}\right)\{^{x,\overline{u},t_0,x'}/_{this,\overline{y},t',z'}\}$$

$$\Gamma'''; x' \vdash_{t_0} P_{\mathtt{C.m}}\{^{x,\overline{u},\overline{v}}/_{this,\overline{y},\overline{z}}\} : \ell_{\mathtt{C.m}}\{^{x,\overline{u},t_0,x'}/_{this,\overline{y},t',z'}\}$$

Then we notice that $\Gamma''' \subseteq \Gamma + \Gamma'', t_0 : \mathtt{Thread}$. Therefore, by (T-Thread) and (T-State) we derive

$$\Gamma + \Gamma'', t_0 : \mathtt{Thread}; x' \vdash P_{\mathtt{C.m}}\{^{x,\overline{u},\overline{v}}/_{this,\overline{y},\overline{z}}\} : \ell_{\mathtt{C.m}}\{^{x,\overline{u},t_0,x'}/_{this,\overline{y},t',z'}\}$$

and, by Proposition A.1,

$$\Gamma + \Gamma'', t_0 : \mathtt{Thread}; \sigma' \cdot x' \vdash P_{\mathtt{C.m}}\{^{x,\overline{u},\overline{v}}/_{this,\overline{y},\overline{z}}\} : \ell'_0$$

$$I_{\mathscr{L}}(\ell'_0) \Subset I_{\mathscr{L}}(\ell_{\mathtt{C.m}}\{^{\overline{u},t_0,x}/_{\overline{y},t',z'}\}\&(\sigma' \cdot x')^{t_0}) \tag{A.6}$$

(we recall that $\sigma = \sigma' \cdot x'$). By applying again (T-Thread) and (T-State) we derive

$$\Gamma + \Gamma'' \vdash \mathbb{P}' : \ell'$$

where $\ell' = ((\nu\ t_0)\ \ell'_0)\,\&\,(\&_{i\in 1..n}\ell_i)$. By definition of $I_{\mathscr{L}}$ and (A.6), we conclude $I_{\mathscr{L}}(\ell') \Subset I_{\mathscr{L}}(\ell)$. □

Lemma A.4 *(Correctness). Let* $\Gamma \vdash (\mathtt{C}_1\ \mathscr{D}_1, \cdots, \mathtt{C}_n\ \mathscr{D}_n, P) : (\mathscr{L}, \ell)$ *such that* $I_{\mathscr{L}}(\ell)$ *has no circularity. If*

$$P \bullet \varepsilon \longrightarrow^* \mathbb{P} \quad \textit{and} \quad \mathbb{P} = \mathtt{sync}(x_0)\{\ P_0\ \} \bullet \sigma_0 \mid \mathbb{P}'$$

then there exists $\mathbb{P}''$ *such that* $\mathbb{P} \longrightarrow \mathbb{P}''$.

Proof. If $\mathbb{P}'$ contains object creation or forks or conditionals or method invocations at the top level, then the required property immediately follows. Similarly if it contains threads like $0 \bullet \sigma$. Thus, we can assume that

$$\mathbb{P}' = \prod_{i \in 1..m} \mathtt{sync}(x_i)\{\ P_i\ \} \bullet \sigma_i\ .$$

We notice that $m \geq 1$, otherwise rule (Sync) is possible. We also assume that, for every j, there is $i_j \neq j$ such that $x_j \in \sigma_{i_j}$, otherwise (Sync) is also possible. We demonstrate a contradiction: $I_{\mathscr{L}}(\ell)$ must have a circularity.

By Lemma A.3, there are Γ' and ℓ' such that $\Gamma + \Gamma' \vdash \mathbb{P} : \ell'$ and $I_{\mathscr{L}}(\ell')$ has no circularity. Because of the form of $\mathbb{P}$, by applying (T-State) and (T-Thread), in order, we have

$$\left(\Gamma + \Gamma', t_i : \mathtt{Thread}; \sigma_i \vdash_{t_i} \mathtt{sync}(x_i)\{\ P_i\ \} : \ell_i\right)^{i \in 0..m} \qquad \ell' = \&_{i \in 0..m}(\nu\ t_i)\ \ell_i$$

Consider the thread $\mathtt{sync}(x_0)\{\ P_0\ \} \bullet \sigma_0$ and let $x_0 \in \sigma_1$. For simplicity, assume that $x_1 \in \sigma_0$.

By (T-Sync) and Proposition A.1, we have $I_{\mathscr{L}}((\sigma_0 \cdot x_0)^{t_0}) \Subset I_{\mathscr{L}}(\ell_0)$ and $I_{\mathscr{L}}((\sigma_1 \cdot x_1)^{t_1}) \Subset I_{\mathscr{L}}(\ell_1)$. Since, according to the assumptions, $(x_1, x_0)_{t_0} \in I_{\mathscr{L}}((\sigma_0 \cdot x_0)^{t_0})$ and $(x_0, x_1)_{t_1} \in I_{\mathscr{L}}((\sigma_1 \cdot x_1)^{t_1})$, by definition of ℓ' and of $I_{\mathscr{L}}$, we derive $\{(x_1, x_0)_{t_0}, (x_0, x_1)_{t_1}\} \Subset I_{\mathscr{L}}(\ell')$ (without loss of generality, we assuming t_0 and t_1 are replaced by identical names). Therefore $I_{\mathscr{L}}(\ell')$ contains a circularity and, by Theorem 2.8, $I_{\mathscr{L}}(\ell)$ must also have a circularity. This contradicts the hypothesis. The general case, when the chain $x_0 \in \sigma_1$, $x_1 \in \sigma_2$, $\cdots$, $x_h \in \sigma_0$, is similar. □

References

[1] Martin Abadi, Cormac Flanagan, Stephen N. Freund, Types for safe locking: static race detection for Java, ACM Trans. Program. Lang. Syst. 28 (2006) 207–255.
[2] Robert Atkey, Donald Sannella, Threadsafe: static analysis for Java concurrency, Electron. Commun. ECEASST 72 (2015).
[3] Saddek Bensalem, Klaus Havelund, Dynamic deadlock analysis of multi-threaded programs, in: Hardware and Software Verification and Testing, in: LNCS, vol. 3875, Springer, 2005, pp. 208–223.
[4] Chandrasekhar Boyapati, Robert Lee, Martin Rinard, Ownership types for safe programming: preventing data races and deadlocks, in: OOPSLA, ACM, 2002, pp. 211–230.
[5] B.A. Davey, H.A. Priestley, Introduction to Lattices and Order, Cambridge University Press, 2002.
[6] Mahdi Eslamimehr, Jens Palsberg, Sherlock: scalable deadlock detection for concurrent programs, in: Proc. 22nd International Symposium on Foundations of Software Engineering (FSE-22), ACM, 2014, pp. 353–365.
[7] Cormac Flanagan, Shaz Qadeer, A type and effect system for atomicity, in: PLDI, ACM, 2003, pp. 338–349.
[8] Antonio Flores-Montoya, Elvira Albert, Samir Genaim, May-happen-in-parallel based deadlock analysis for concurrent objects, in: FORTE/FMOODS 2013, in: LNCS, vol. 7892, Springer, 2013, pp. 273–288.
[9] Abel Garcia, Static Analysis of Concurrent Programs Based on Behavioral Type Systems, PhD thesis, School in Computer Science and Engineering, 2017. Available at JaDA.cs.unibo.it.
[10] Abel Garcia, Cosimo Laneve, JaDA – the Java deadlock analyser, in: Behavioural Types: From Theories to Tools, River Publishers, 2017, pp. 169–192.
[11] Elena Giachino, Naoki Kobayashi, Cosimo Laneve, Deadlock analysis of unbounded process networks, in: CONCUR 2014 – Concurrency Theory – 25th International Conference, in: Lecture Notes in Computer Science, vol. 8704, Springer, 2014, pp. 63–77.
[12] Elena Giachino, Cosimo Laneve, Deadlock detection in linear recursive programs, in: Proceedings of SFM-14:ESM, in: LNCS, vol. 8483, Springer-Verlag, 2014, pp. 26–64.
[13] Elena Giachino, Cosimo Laneve, Michael Lienhardt, A framework for deadlock detection in core ABS, Softw. Syst. Model. 15 (4) (2016) 1013–1048.
[14] Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, Martin Steffen, ABS: a core language for abstract behavioral specification, in: FMCO, in: LNCS, vol. 6957, Springer-Verlag, 2011, pp. 142–164.
[15] Neil D. Jones, Lawrence H. Landweber, Y. Edmund Lien, Complexity of some problems in Petri nets, Theor. Comput. Sci. 4 (3) (1977) 277–299.
[16] Naoki Kobayashi, A new type system for deadlock-free processes, in: CONCUR, in: LNCS, vol. 4137, Springer, 2006, pp. 233–247.
[17] Naoki Kobayashi, Cosimo Laneve, Deadlock analysis of unbounded process networks, Inf. Comput. 252 (2017) 48–70.
[18] Cosimo Laneve, A lightweight deadlock analysis for programs with threads and reentrant locks, in: Proc. of 22nd Int. Symposium on Formal Methods (FM 2018), in: Lecture Notes in Computer Science, vol. 10951, Springer, 2018, pp. 608–624.
[19] Cosimo Laneve, Abel Garcia, Deadlock detection of Java bytecode, in: Proc. of 27th Int. Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2017), in: Lecture Notes in Computer Science, vol. 10855, Springer, 2017, pp. 37–53.
[20] Cosimo Laneve, Luca Padovani, Deadlock analysis of wait-notify coordination, cs.unibo.it/~laneve/papers/DAoWNC.pdf, January 2019.
[21] Robin Milner, A Calculus of Communicating Systems, LNCS, vol. 92, Springer, 1980.
[22] Robin Milner, Joachim Parrow, David Walker, A calculus of mobile processes, II, Inf. Comput. 100 (1992) 41–77.
[23] Mayur Naik, Chang-Seo Park, Koushik Sen, David Gay, Effective static deadlock detection, in: Proc. 31st International Conference on Software Engineering (ICSE 2009), ACM, 2009, pp. 386–396.
[24] Martin Odersky, et al., An Overview of the Scala Programming Language, Technical Report IC/2004/64, EPFL, Lausanne, Switzerland, 2004.
[25] Wolfgang Reisig, Petri Nets: An Introduction, Springer-Verlag, Berlin, Heidelberg, 1985.
[26] John C. Reynolds, Separation logic: a logic for shared mutable data structures, in: Proc. of LICS 2002, IEEE Computer Society, 2002, pp. 55–74.
[27] Kohei Suenaga, Type-based deadlock-freedom verification for non-block-structured lock primitives and mutable references, in: APLAS, in: LNCS, vol. 5356, Springer, 2008, pp. 155–170.
[28] Vasco Thudichum Vasconcelos, Francisco Martins, Tiago Cogumbreiro, Type inference for deadlock detection in a multithreaded polymorphic typed assembly language, in: PLACES, in: EPTCS, vol. 17, 2009, pp. 95–109.