A lightweight deadlock analysis for programs with threads and reentrant locks

Cosimo Laneve

A lightweight deadlock analysis for programs with threads and reentrant locks

Cosimo Laneve

Dept. of Computer Science and Engineering, University of Bologna – INRIA Focus
cosimo.laneve@unibo.it

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). The technique is intended to be an effective tool for the deadlock analysis of programming languages by defining ad-hoc extraction processes.

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.). Most of these languages feature thread creations and guarantee exclusive access to objects by means of synchronizations. In this model, 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

```
buildTable(x,y;n) = (newObject z)( if (n=0) then sync(y){ sync(x){ 0 } }
                                    else (newThread sync(x){ sync(z){ 0 } })
                                         buildTable(z,y;n-1)
                                  )
```

where `newObject z` creates a new object (the class is omitted), (`newThread` P) Q creates a new thread whose body is P (the class is again omitted) and running it in parallel with the continuation Q, and `sync(x){` P `}` is the operation locking the object `x` and performing P. This method creates a table of $n+1$ threads – the *philosophers* – each one sharing an object – the *fork* – with the close one. 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 release 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 `buildTable(x,x;n)`, 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(x){ sync(y){ 0 } }` a deadlock may occur because philosophers' strategies are all symmetric. It is worth to notice that *`buildTable`*`(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 analysers [4, 7, 1, 13, 19, 20] use finite approximate models representing the dependencies between object names. The corresponding algorithms usually return false positives with input *`buildTable`*`(x,x;n)` because they are not powerful enough to manage structures that are not statically bounded.

In [10, 14] we solved this problem for value-passing CCS [16] and pi-calculus [17]. In that case, the technique used two formal models: Petri Nets and *deadLock Analysis Model* – lams, which are basic recursive models that collect dependencies and features recursion and dynamic name creation. In the pi-calculus analyser, 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. In particular, the corresponding algorithm required a tool for verifying the reachability of Petri Nets that model channels' behaviours (which has exponential computational complexity with respect to the size of the net [12]) and a tool for analysing 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 *`buildTable`* is [1]

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

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 [10, 14]; it has been necessary for modelling 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 it means that t is acquiring x *twice*. Therefore, $\texttt{buildTable}(t,x,x)$ manifests a circularity in the model of [10, 14] and it does not in the extended model. A problematic lam 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 [10, 14]. 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

[1] Actually, the lam function associated to *`buildTable`* by the type system in Section 4 has an additional name that records the last name synchronized by the thread t.

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 object-oriented calculus featuring recursion, threads and synchronizations that guarantee exclusive access to objects. (The method `buildTable` is written in our calculus.) The syntax, semantics, and examples of the object-oriented calculus are in Section 3. In Section 4 we define a type system that associates lams to processes. Using the type system, for example, the lam function `buildTable` can be extracted from the method `buildTable`. As a byproduct of the type system and the lams, our technique can detect deadlocks of programs like `buildTable`. For space constraints, the proof of soundness of the type system is omitted: is is reported in the full paper [2]. We discuss a few weaknesses of the techniques in Section 5 and we point to related works and deliver some concluding remark in Section 6.

Overall, the technicalities (the algorithm for lams, the syntax and semantics of the calculus, the typing rules, and the type safety) illustrate many interesting features of a deadlock analyser for a full object-oriented language, while remaining pleasingly compact. In fact, this paper also aims at presenting a handy tool for studying the consequences of extensions and variations of the constructs defined here.

2 Lams and the algorithm for detecting circularities

This section extends the theory developed in [10, 14] to cover thread reentrancy. In particular, the new definitions are those of transitive closure and Definition 3. Theorem 1 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 $(x, y)_t \in {\mathtt{R}'_j}^+$ with $t \in \mathscr{A}$;
  3. if $(x, y)_\bullet \in \mathtt{R}_i$ then either $(x, y)_\bullet \in {\mathtt{R}'_j}^+$ or $(x, y)_t \in {\mathtt{R}'_j}^+$ with $t \in \mathscr{A}$.

[2] Available at `http://cs.unibo.it/~laneve/papers/FM2018-full.pdf`.

– $\{\mathtt{R}_1, \cdots, \mathtt{R}_m\} \& \{\mathtt{R}'_1, \cdots, \mathtt{R}'_n\} \stackrel{def}{=} \{\mathtt{R}_i \cup \mathtt{R}'_j \mid 1 \leqslant i \leqslant m \text{ and } 1 \leqslant j \leqslant n\}$.

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 out 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 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* [14], noted ℓ, whose syntax is

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

The term $\mathtt{0}$ 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 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 $\big(\mathscr{L}, \ell\big)$, 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 $\ell_f = (\nu\ \overline{z})\ \ell'_\mathtt{f}$ where $\ell'_\mathtt{f}$ has no ν-binder. Similarly for ℓ. The function `buildTable` in the Introduction is an example of a lam function.

The semantics of lams is very simple: it amounts to unfolding function invocations. 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. The operational semantics of a program $\big(\mathscr{L}, (\nu\ \overline{x})\ \ell\big)$ is a transition system where *states* are lams, the *transition relation* is the least one satisfying the rule

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

and the initial state is the lam 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 any 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

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

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

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

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

Definition 2. *A lam ℓ* has a circularity *if $I^{\perp}(\ell)$ has a circularity. A lam program $\big(\mathscr{L}, \ell\big)$* has a circularity *if there is $\ell \longrightarrow^* \ell'$ and ℓ' has a circularity.*

For example the above lam ℓ has a circularity because

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

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 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

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

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 `proj`:

- first of all, notice that `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 1. The flattening functions of `buildTable` are

$$\begin{aligned} I^{(0)}_{\mathscr{L}}(\texttt{buildTable}) &= \{\varnothing\} \\ I^{(1)}_{\mathscr{L}}(\texttt{buildTable}) &= \{\ \{(y,x)_t\}\ \} \end{aligned}$$

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

$$\begin{aligned} I^{(0)}_{\mathscr{L}}(\texttt{g}) &= \{\varnothing\} \\ I^{(1)}_{\mathscr{L}}(\texttt{g}) &= \{\ \{(x,y)_\bullet\}\ \} \\ I^{(2)}_{\mathscr{L}}(\texttt{g}) &= \{\ \{(x,y)_\bullet, (y,z)_\bullet, (x,z)_\checkmark\}\ \} \end{aligned}$$

Proposition 1. *Let* $\texttt{f}(\overline{x}) = (\nu\ \overline{z})\ \ell_{\texttt{f}} \in \mathscr{L}$.

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

Proof. (1) follows by definition. As regards (2), we observe that $I(\ell)$ is monotonic on I: for every $\texttt{f}$, $I(\texttt{f}) \Subset I'(\texttt{f})$ implies $I(\ell) \Subset I'(\ell)$, which can be demonstrated by a standard structural induction on ℓ. Then, an induction on k gives $I^{(k)}_{\mathscr{L}}(\texttt{f}) \Subset I^{(k+1)}_{\mathscr{L}}(\texttt{f})$. □

Since, for every k, $I^{(k)}_{\mathscr{L}}(\texttt{f}_i)$ 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 1, $I^{(1)}_{\mathscr{L}}$ is the fixpoint of `buildTable` and $I^{(2)}_{\mathscr{L}}$ is the fixpoint of `g` .

Proposition 2. *Let* `L` *be a lam context,* ℓ *be a lam, and* $I(\cdot)$ *be a flattening. Then we have:*

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

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

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

Lemma 1. *Let* $\big(\{\mathtt{f}_1(\overline{x}_1) = (\nu\ \overline{z}_1)\ \ell_1, \cdots, \mathtt{f}_n(\overline{x}_n) = (\nu\ \overline{z}_n)\ \ell_n\},\ \ell\big)$ *be a lam program 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(1), there exists $\mathtt{R}_j \in I^{(k)}(\ell_{i_1}\{\overline{z}'_j/\overline{z}_{i_j}\}\{\overline{u}_j/\overline{x}_{i_j}\})$ with $1 \leqslant j \leqslant m$ such that $I^{(k)}(\mathtt{L}[\mathtt{R}_1]\cdots[\mathtt{R}_m])$ 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(2), we have that

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

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

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

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 [10, 14], Lemma 1 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 $k \geqslant 1$, $I^{(k)}_{\mathscr{L}}(\mathtt{h}) = \{\,\{(x,y)_\bullet, (y,z)_\bullet\}\,\}$. Notice that $I^{(k)}_{\mathscr{L}}(\mathtt{h}(x,y,x)) = \{\,\{(x,y)_\bullet, (y,x)_\bullet\}\,\}$, which has a circularity – see Definition 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 1. *Let* $\big(\mathscr{L},\ell\big)$ *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 1(2) $I^{(0)}_{\mathscr{L}}(\ell') \Subset I_{\mathscr{L}}(\ell')$. Therefore $I_{\mathscr{L}}(\ell')$ has also a circularity. Then, by Lemma 1, 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 [9].

3 The language and its semantics

In the rest of the paper, we use lams to define an analysis technique for a simple programming model of concurrent object-oriented languages (the basic operations of thread creation and synchronization used in `Java` and `C#` may be easily recognized). In this section we first define the model, give a description of how deadlock may be identified, and discuss few examples. The next section defines the type system associating 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 A, B, $\cdots$. A *program* is a pair $\big(\mathscr{D}, P\big)$, where $\mathscr{D}$ is a *finite set* of *method name definitions* $A(\overline{x};\overline{y}) = P_A$, with $\overline{x};\overline{y}$ and P_A respectively being the *formal parameters* and the *body* of A, and P is the *main process*.

The syntax of processes P and expressions e is defined below

$$\begin{array}{rcl} P & ::= & \mathtt{0} \;\mid\; (\nu\ x)\ P \;\mid\; (\nu\ P)\ P \;\mid\; \mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ P \;\mid\; A(\overline{x};\overline{y}) \\ & \mid & \mathit{sync}(x)\{\ P\ \}.\ P \\ e & ::= & x \;\mid\; v \;\mid\; e\ \mathtt{op}\ e \end{array}$$

A process can be the inert process $\mathtt{0}$, or a restriction $(\nu\ x)\ P$ that behaves like P except that the external environment cannot access to the object x, or the spawn $(\nu\ Q)\ P$ of a new thread Q by a process P, or a conditional $\mathtt{if}\ e\ \mathtt{then}\ P\ \mathtt{else}\ Q$ 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 $A(\overline{x};\overline{y})$ of the process corresponding to A. In the invocation, a semicolon separates the arguments that are objects from those that are integers. The last process is $\mathit{sync}(x)\{\ P\ \}.\ Q$ that executes P with exclusive access to x and then performs Q. An expression e can be a name x, an integer value v, or a generic binary operation on integers $v\ \mathtt{op}\ v'$, where $\mathtt{op}$ ranges over a set including the usual operators like $+$, $\leqslant$, 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 a constant expression e ($[\![e]\!]$ is undefined when the integer expression e contains integer names). Let also $[\![x]\!] = x$ when x is a non-integer name. We always shorten $\mathit{sync}(x)\{\ P\ \}.\ \mathtt{0}$ into $\mathit{sync}(x)\{\ P\ \}$.

In order to define the operational semantics, we use terms $\mathbb{P} ::= P \mid P \overset{x}{\bullet} \mathbb{P}$ that are called *threads*. The term $P \overset{x}{\bullet} \mathbb{P}$ corresponds to a thread that is performing P in a critical section for x; when P terminates, the lock of x must be released (if $\mathbb{P}$

does not contain $\overset{x}{\bullet}$) and the continuation $\mathbb{P}$ may start. The thread $\mathbb{P}$ is *reentrant* on x when $\overset{x}{\bullet}$ occurs at least twice in $\mathbb{P}$.

States, ranged over by $\mathscr{T}$, are multisets of threads, written $\mathbb{P}_1 \mid \cdots \mid \mathbb{P}_n$ and sometime shortened into $\prod_{i\in 1..n} \mathbb{P}_i$. We write $x \in \mathbb{P}$ if $\mathbb{P}$ contains $\overset{x}{\bullet}$; we write $x \in \mathscr{T}$ if there is $\mathbb{P} \in \mathscr{T}$ such that $x \in \mathbb{P}$.

Definition 4. *The* structural equivalence $\equiv$ *on threads is the least congruence containing alpha-conversion of bound names, commutativity and associativity of* $\mid$ *with identity* $\mathsf{0}$*, closed under the rule:*

$$((\nu\ x)\ \mathbb{P}) \mid \mathscr{T} \ \equiv\ (\nu\ x)\ (\mathbb{P} \mid \mathscr{T}) \qquad x \notin \mathit{var}(\mathscr{T})\ .$$

The operational semantics *of a program* $\big(\mathscr{D}, P\big)$ *is a transition system where the initial state is* P*, and the* transition relation $\longrightarrow_{\mathscr{D}}$ *is the least one closed under the rules (the notation* $P\,[\,\overset{x}{\bullet}\,\mathbb{P}\,]$ *stands for either* P *or* $P \overset{x}{\bullet} \mathbb{P}$*):*

$$\text{(Zero)}\quad \mathsf{0} \overset{x}{\bullet} \mathbb{P} \mid \mathscr{T} \longrightarrow_{\mathscr{D}} \mathbb{P} \mid \mathscr{T}$$

$$\text{(NewO)}\quad \frac{z \text{ fresh}}{(\nu\ x)\ \mathbb{P} \mid \mathscr{T} \longrightarrow_{\mathscr{D}} \mathbb{P}\{^{z}/_{x}\} \mid \mathscr{T}} \qquad \text{(NewT)}\quad (\nu\ P)\ \mathbb{P} \mid \mathscr{T} \longrightarrow_{\mathscr{D}} P \mid \mathbb{P} \mid \mathscr{T}$$

$$\text{(IfT)}\quad \frac{[\![e]\!] \neq 0}{\texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ P'\,[\,\overset{x}{\bullet}\,\mathbb{P}\,] \mid \mathscr{T} \longrightarrow_{\mathscr{D}} P\,[\,\overset{x}{\bullet}\,\mathbb{P}\,] \mid \mathscr{T}} \qquad \text{(IfF)}\quad \frac{[\![e]\!] = 0}{\texttt{if}\ e\ \texttt{then}\ P\ \texttt{else}\ P'\,[\,\overset{x}{\bullet}\,\mathbb{P}\,] \mid \mathscr{T} \longrightarrow_{\mathscr{D}} P'\,[\,\overset{x}{\bullet}\,\mathbb{P}\,] \mid \mathscr{T}}$$

$$\text{(Call)}\quad \frac{[\![\overline{e}]\!] = \overline{v} \quad A(\overline{y};\overline{z}) = P \in \mathscr{D}}{A(\overline{u};\overline{e})\,[\,\overset{x}{\bullet}\,\mathbb{P}\,] \mid \mathscr{T} \longrightarrow_{\mathscr{D}} P\{^{\overline{u};\overline{v}}/_{\overline{y},\overline{z}}\}\,[\,\overset{x}{\bullet}\,\mathbb{P}\,] \mid \mathscr{T}}$$

$$\text{(Sync)}\quad \frac{x \notin \mathscr{T}}{\mathit{sync}(x)\{\ P\ \}.\ \mathbb{P} \mid \mathscr{T} \longrightarrow_{\mathscr{D}} P \overset{x}{\bullet} \mathbb{P} \mid \mathscr{T}} \qquad \text{(Cong)}\quad \frac{\mathscr{T}_1' \longrightarrow_{\mathscr{D}} \mathscr{T}_2' \qquad \mathscr{T}_1 \equiv (\nu\ x)\ \mathscr{T}_1' \quad (\nu\ x)\ \mathscr{T}_2' \equiv \mathscr{T}_2}{\mathscr{T}_1 \longrightarrow_{\mathscr{D}} \mathscr{T}_2}$$

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

Definition 5 (deadlock-freedom). *A program* $\big(\mathscr{D}, P\big)$ *is* deadlock-free *if the following condition holds:*

whenever $P \longrightarrow^* \mathscr{T}$ *and* $\mathscr{T} \equiv (\nu\ x_1)\ \cdots (\nu\ x_n)\ (\mathit{sync}(x)\{\ P'\ \}.\ \mathbb{P} \mid \mathscr{T}')$
then there exists $\mathscr{T}''$ *such that* $\mathscr{T} \longrightarrow \mathscr{T}''$*.*

Example 2. We select three processes and discuss their behaviours, highlighting whether they deadlock or not:

– $(\nu\ sync(x)\{\ sync(y)\{\ \mathtt{0}\ \}\ \})\ sync(x)\{\ sync(y)\{\ \mathtt{0}\ \}\ \}$. 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 $(\nu\ sync(y)\{\ sync(x)\{\ \mathtt{0}\ \}\ \})\ sync(x)\{\ sync(y)\{\ \mathtt{0}\ \}\ \}$ spawns a thread acquiring the locks in reverse order. This is a computation giving a deadlock:

$$\begin{aligned}(\nu\ sync(y)\{\ sync(x)\{\ \mathtt{0}\ \}\ \})\ sync(x)\{\ sync(y)\{\ \mathtt{0}\ \}\ \}\\ &\longrightarrow\quad sync(y)\{\ sync(x)\{\ \mathtt{0}\ \}\ \}\mid sync(x)\{\ sync(y)\{\ \mathtt{0}\ \}\ \}\\ &\longrightarrow\quad sync(x)\{\ \mathtt{0}\ \}\overset{y}{\bullet}\mathtt{0}\mid sync(x)\{\ sync(y)\{\ \mathtt{0}\ \}\ \}\\ &\longrightarrow\quad sync(x)\{\ \mathtt{0}\ \}\overset{y}{\bullet}\mathtt{0}\mid sync(y)\{\ \mathtt{0}\ \}\overset{x}{\bullet}\mathtt{0}\end{aligned}$$

– The following method

$$\begin{aligned}A(x,y;n) = {}&\mathtt{if}\ (n=0)\ \mathtt{then}\ (\nu\ sync(y)\{\ sync(x)\{\ \mathtt{0}\ \}\ \})\ sync(y)\{\ \mathtt{0}\ \}\\ &\mathtt{else}\ sync(x)\{\ A(x,y;n-1)\ \}\end{aligned}$$

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

4 Static semantics

Environments, ranged over by Γ, contain the types of objects, e.g. $x : \mathtt{C}$ (we assume objects have all the same class $\mathtt{C}$), the type of integer variables e.g. $x : \mathtt{int}$, and the types of process names, e.g. $A : [\overline{C}; \overline{\mathtt{int}}]$. Types $\mathtt{C}$ and $\mathtt{int}$ are ranged over by $\mathtt{T}$. Let $dom(\Gamma)$ be the domain of Γ and let

– $\Gamma, x{:}\mathtt{T}$, when $x \notin dom(\Gamma)$

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

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

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

We also use sequences σ of (object) names that record the nesting of synchronizations. Let $(x_1 \cdots x_n)^t \stackrel{def}{=} \&_{i\in 1..n-1}(x_i, x_{i+1})_t$.

The static semantics has two 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 $\Gamma; \sigma$.

Processes:

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

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

$$\text{(T-Par)}\quad \frac{\Gamma;\sigma \vdash_t P : \ell \quad \Gamma, t'{:}\mathtt{C}, z'{:}\mathtt{C}; z' \vdash_{t'} P' : \ell'}{\Gamma;\sigma \vdash_t (\nu\ P')\ P : \ell \& (\nu\ t', z')\ \ell'} \qquad \text{(T-Call)}\quad \frac{\Gamma(A) = [\overline{\mathtt{C}}; \overline{\mathtt{int}}] \quad |\overline{u}| = |\overline{\mathtt{C}}| \quad \Gamma \vdash \overline{e} : \overline{\mathtt{int}}}{\Gamma;\sigma \cdot x \vdash_t A(\overline{u}; \overline{e}) : \mathtt{f}_A(t, x, \overline{u}) \& (\sigma \cdot x)^t}$$

Expressions:

$$\text{(T-Int)}\quad \Gamma \vdash n : \mathtt{int} \qquad \text{(T-Var)}\quad \Gamma, x : \mathtt{T} \vdash x : \mathtt{T} \qquad \text{(T-Op)}\quad \frac{\Gamma \vdash e : \mathtt{int} \quad \Gamma \vdash e' : \mathtt{int}}{\Gamma \vdash e\ \mathtt{op}\ e' : \mathtt{int}} \qquad \text{(T-Seq)}\quad \frac{(\Gamma \vdash e_i : \mathtt{T}_i)^{i \in 1..n}}{\Gamma \vdash e_1, \ldots, e_n : \mathtt{T}_1, \ldots, \mathtt{T}_n}$$

Programs:

$$\text{(T-Prog)}\quad \frac{\begin{array}{c}\mathscr{D} = \bigcup_{i \in 1..n} \{A_i(\overline{x}_i; \overline{y}_i) = P_i\} \quad \Gamma = (A_i : [\overline{\mathtt{C}}; \overline{\mathtt{int}}])^{i \in 1..n} \\ (\Gamma, \overline{x}_i{:}\overline{\mathtt{C}}, \overline{y}_i{:}\overline{\mathtt{int}}, t_i{:}\mathtt{C}, z_i{:}\mathtt{C}; z_i \vdash_{t_i} P_i : \ell_i)^{i \in 1..n} \qquad \Gamma, t{:}\mathtt{C}, z{:}\mathtt{C}; z \vdash_t P : \ell \\ \mathscr{L} = \bigcup_{i \in 1..n} \{\mathtt{f}_{A_i}(t_i, z_i, \overline{x}_i) = \ell_i\}\end{array}}{\Gamma \vdash (\mathscr{D}, P) : (\mathscr{L}, \ell)}$$

Fig. 1. The type system (we assume a function name $\mathtt{f}_A$ for every process name A)

The type system is defined in Figure 1. A few key rules are discussed. Rule (T-Zero) types the process $\mathtt{0}$ 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 – *c.f.* notation $(\sigma)^t$. Rule (T-Sync) types the critical section P with a sequence of locks extended with x. The corresponding lam is in disjunction with the lam of the continuation P' because, in P' the lock on x has been released. Rule (T-Par) types a parallel composition of processes by collecting the lams of the components. Rule (T-Call) types a process name invocation in terms of a (lam) function invocation and constrains the sequences of object names in the two invocations to have equal lengths ($|\overline{u}| = |\overline{\mathtt{C}}|$) and the types of expressions to match with the types in the process 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.

Example 3. Let us show the typing of the method ***buildTable*** in the Introduction (the keywords `newThread` and `newObject` are replaced by ν). Let

$$\begin{array}{l} \Gamma = \texttt{buildTable}{:}[\texttt{C},\texttt{C},\texttt{C},\texttt{C};\texttt{int}], x{:}\texttt{C}, y{:}\texttt{C}, n{:}\texttt{int}, t{:}\texttt{C}, u{:}\texttt{C} \\ P = sync(y)\{sync(x)\{\texttt{0}\}\} \\ Q = (\nu\ sync(x)\{sync(z)\{\texttt{0}\}\})\ \texttt{buildTable}(t,u,z,y;n-1) \end{array}$$

Then

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

where $\ell_2 = (\nu\ s,v)\ \ell_2' + \ell_2''$ and $(*)$ are the two proof trees

$$\dfrac{\cdots}{\Gamma, z:\texttt{C}, s:\texttt{C}, v:\texttt{C}; v \vdash_t sync(x)\{sync(z)\{\texttt{0}\}\} : \ell_2'}$$

and

$$\dfrac{\cdots}{\Gamma, z:\texttt{C}; u \vdash_t \texttt{buildTable}(t,u,z,y;n-1) : \ell_2''}$$

(the reader may complete them). After completing the proof tree, one obtains the lam function

$$\begin{array}{rl} \texttt{buildTable}(t,u,x,y) = & (\nu\, z,s,v)(\ (u,y)_t\&(y,x)_t \\ & \quad + (v,x)_s\&(x,z)_s\ \&\ \texttt{buildTable}(t,u,z,y)\ ) \end{array}$$

which has an additional argument with respect to the one in the Introduction.

The following theorem states the soundness of our type system.

Theorem 2. *Let $\Gamma \vdash (\mathscr{D}, P) : (\mathscr{L}, \ell)$. If $(\mathscr{L}, \ell)$ has no circularity then $(\mathscr{D}, P)$ is deadlock-free.*

Example 4. Let us verify whether the process ***buildTable***(x,x,n) is deadlock-free. The lam function associated by the type system is detailed in Example 3. The interpretation function $I_{\mathscr{L}}(\texttt{buildTable})$ is computed as follows:

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

Since $I^{(2)}_{\mathscr{L}} = I_{\mathscr{L}}$, we are reduced to compute $I^{(2)}_{\mathscr{L}}(\texttt{buildTable}(t,u,x,x))$. 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 ***buildTable***(x, x, n) is deadlock-free.

It is interesting to verify whether the process ***buildTableD***(x, x, n) is deadlock-free, where ***buildTableD*** is the method having philosophers with symmetric strategies:

```
buildTableD(x,y;n) = (ν z)( if (n=0) then sync(x){ sync(y){ 0 } }
                            else (ν sync(x){ sync(z){ 0 } })
                                 buildTableD(z,y;n-1)
                          )
```

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

5 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

$$(\nu\ sync(x)\{\ sync(z)\{\ sync(y)\{\ 0\ \}\ \}\ \})\ \ sync(x)\{\ sync(y)\{\ sync(z)\{\ 0\ \}\ \}\ \}\ .$$

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 [9] 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 $A(x, y, n)$ in Example 2(3). This process never deadlocks when $n \leqslant 0$. However, since our technique drops integer values, it always return a circularity (this is a correct result when $n > 0$ and it is 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.

6 Related works and Conclusions

In this paper 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 defined an algorithm for verifying the absence of circularities in lams and we have applied this model to a simple concurrent object-oriented calculus. This work is intended to serve as a core system for studying the consequences of extensions and variations.

The lam model has been introduced and studied for detecting deadlocks of an object-oriented language with futures (and no lock and lock reentrancy) [11], 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` bytecode [9]. The paper [15], reports an initial assessment of `JaDA` with respect to other tools (it also contains a (very) informal description of the algorithm). As we discussed in the Introduction, the model has been also applied to process calculi [10, 14].

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, 18], 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`, 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 [11].

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. For this reason we are currently analysing Petri Nets techniques that complement our extended lam model with *happen-before* informations, in the same way as we did for process calculi.

References

1. Martin Abadi, Cormac Flanagan, and Stephen N. Freund. Types for safe locking: Static race detection for Java. *ACM Trans. Prog. Lang. Syst.*, 28:207–255, 2006.
2. Robert Atkey and Donald Sannella. Threadsafe: Static analysis for Java concurrency. *Electronic Communications of the ECEASST*, 72, 2015.
3. Saddek Bensalem and Klaus Havelund. Dynamic deadlock analysis of multi-threaded programs. In *in Hardware and Software Verification and Testing*, volume 3875 of *LNCS*, pages 208–223. Springer, 2005.
4. Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe program.: preventing data races and deadlocks. In *OOPSLA*, pages 211–230. ACM, 2002.
5. B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 2002.
6. Mahdi Eslamimehr and Jens Palsberg. Sherlock: scalable deadlock detection for concurrent programs. In *Proc. 22nd International Symposium on Foundations of Software Engineering (FSE-22)*, pages 353–365. ACM, 2014.
7. Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *PLDI*, pages 338–349. ACM, 2003.
8. Antonio Flores-Montoya, Elvira Albert, and Samir Genaim. May-happen-in-parallel based deadlock analysis for concurrent objects. In *FORTE/FMOODS 2013*, volume 7892 of *LNCS*, pages 273–288. Springer, 2013.
9. Abel Garcia and Cosimo Laneve. `JaDA` – the Java Deadlock Analyser. In *Behavioural Types: from Theories to Tools*, pages 169–192. River Publishers, 2017.
10. Elena Giachino, Naoki Kobayashi, and Cosimo Laneve. Deadlock analysis of unbounded process networks. In *CONCUR 2014 - Concurrency Theory - 25th International Conference*, volume 8704 of *Lecture Notes in Computer Science*, pages 63–77. Springer, 2014.
11. Elena Giachino, Cosimo Laneve, and Michael Lienhardt. A framework for deadlock detection in core ABS. *Software and System Modeling*, 15(4):1013–1048, 2016.
12. Neil D. Jones, Lawrence H. Landweber, and Y. Edmund Lien. Complexity of some problems in petri nets. *Theoretical Computer Science*, 4(3):277 – 299, 1977.
13. Naoki Kobayashi. A new type system for deadlock-free processes. In *CONCUR*, volume 4137 of *LNCS*, pages 233–247. Springer, 2006.
14. Naoki Kobayashi and Cosimo Laneve. Deadlock analysis of unbounded process networks. *Inf. Comput.*, 252:48–70, 2017.
15. Cosimo Laneve and Abel Garcia. Deadlock detection of Java Bytecode. accepted at LOPSTR 2017 pre-proceedings, 2017. URL: `http://arxiv.org/abs/1709.04152`.
16. Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer, 1980.
17. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, ii. *Inf. and Comput.*, 100:41–77, 1992.
18. Mayur Naik, Chang-Seo Park, Koushik Sen, and David Gay. Effective static deadlock detection. In *Proc. 31st International Conference on Software Engineering (ICSE 2009)*, pages 386–396. ACM, 2009.
19. Kohei Suenaga. Type-based deadlock-freedom verification for non-block-structured lock primitives and mutable references. In *APLAS*, volume 5356 of *LNCS*, pages 155–170. Springer, 2008.
20. Vasco Thudichum Vasconcelos, Francisco Martins, and Tiago Cogumbreiro. Type inference for deadlock detection in a multithreaded polymorphic typed assembly language. In *PLACES*, volume 17 of *EPTCS*, pages 95–109, 2009.