Deadlock Detection in Linear Recursive Programs

Elena Giachino, Cosimo Laneve

Deadlock detection in linear recursive programs*

Elena Giachino and Cosimo Laneve

Dept. of Computer Science and Egineering, Università di Bologna – INRIA FOCUS
{giachino,laneve}@cs.unibo.it

Abstract. Deadlock detection in recursive programs that admit dynamic resource creation is extremely complex and solutions either give imprecise answers or do not scale.
We define an algorithm for detecting deadlocks of *linear recursive programs* of a basic model. The theory that underpins the algorithm is a generalization of the theory of permutations of names to so-called *mutations*, which transform tuples by introducing duplicates and fresh names. Our algorithm realizes the back-end of deadlock analyzers for object-oriented programming languages, once the association programs/basic-model-programs has been defined as front-end.

1 Introduction

Modern systems are designed to support a high degree of parallelism by ensuring that as many system components as possible are operating concurrently. Deadlock represents an insidious and recurring threat when such systems also exhibit a high degree of resource and data sharing. In these systems, deadlocks arise as a consequence of exclusive resource access and circular wait for accessing resources. A standard example is when two processes are exclusively holding a different resource and are requesting access to the resource held by the other. In other words, the correct termination of each of the two process activities *depends* on the termination of the other. Since there is a *circular dependency*, termination is not possible.

The techniques for detecting deadlocks build graphs of dependencies (x, y) between resources, meaning that the release of a resource referenced by x depends on the release of the resource referenced by y. The absence of cycles in the graphs entails deadlock freedom. The difficulties arise in the presence of infinite (mutual) recursion: consider, for instance, systems that create an unbounded number of processes such as server applications. In such systems, process interaction becomes complex and either hard to predict or hard to be detected during testing and, even when possible, it can be difficult to reproduce deadlocks and find their causes. In these cases, the existing deadlock detection tools, in order to ensure termination, typically lean on finite models that are extracted from the dependency graphs.

* Partly funded by the EU project FP7-610582 ENVISAGE: Engineering Virtualized Services.

The most powerful deadlock analyzer we are aware of is TyPiCal, a tool developed for pi-calculus by Kobayashi [20, 18, 16, 19]. This tool uses a clever technique for deriving inter-channel dependency information and is able to deal with several recursive behaviors and the creation of new channels without using any pre-defined order of channel names. Nevertheless, since TyPiCal is based on an inference system, there are recursive behaviors that escape its accuracy. For instance, it returns false positives when recursion is mixed up with *delegation*. To illustrate the issue we consider the following deadlock-free pi-calculus factorial program

```
*factorial?(n,(r,s)).
  if n=0 then r?m. s!m else new t in
                        (r?m. t!(m*n)) | factorial!(n-1,(t,s))
```

In this code, `factorial` returns the value (on the channel `s`) by *delegating* this task to the recursive invocation, if any. In particular, the initial invocation of `factorial`, which is `r!1 | factorial!(n,(r,s))`, performs a synchronization between `r!1` and the input `r?m` in the continuation of `factorial?(n,(r,s))`. In turn, this may delegate the computation of the factorial to a subsequent synchronization on a new channel `t`. TyPiCal signals a deadlock on the two inputs `r?m` because it fails in connecting the output `t!(m*n)` with them.

The technique we develop in this paper allows us to demonstrate the deadlock freedom of programs like the one above.

To ease program reasoning, our technique relies on an abstraction process that extracts the dependency constraints in programs

- by dropping primitive data types and values;
- by highlighting dependencies between pi-calculus actions;
- by overapproximating statement behaviors, namely collecting the dependencies and the invocations in the two branches of the conditional (the set union operation is modeled by $\&$).

This abstraction process is currently performed by a formal inference system that does not target pi-calculus, but it is defined for a Java-like programming language, called `ABS` [17], see Section 6. Here, pi-calculus has been considered for expository purposes. The `ABS` program corresponding to the pi-calculus `factorial` may be downloaded from [15]; readers that are familiar with Java may find the code in the Appendix A. As a consequence of the abstraction operation we get the function

$$\texttt{factorial}(r,s) = (r,s)\&(r,t)\&\texttt{factorial}(t,s)$$

where (r,s) shows the dependency between the actions `r?m` and `s!m` and (r,t) the one between `r?m` and `t!(m*n)`. The semantics of the abstract `factorial` is defined operationally by unfolding the recursive invocations. In particular, the unfolding of $\texttt{factorial}(r,s)$ yields the sequence of abstract states (free names in the definition of `factorial` are replaced by fresh names in the unfoldings)

$$
\begin{aligned}
\texttt{factorial}(r,s) &\longrightarrow (r,s)\&(r,t)\&\texttt{factorial}(t,s)\\
&\longrightarrow (r,s)\&(r,t)\&(t,s)\&(t,u)\&\texttt{factorial}(u,s)\\
&\longrightarrow (r,s)\&(r,t)\&(t,s)\&(t,u)\&(u,s)\&(u,v)\\
&\quad\ \&\texttt{factorial}(v,s)\\
&\longrightarrow \quad \cdots
\end{aligned}
$$

We demonstrate that the abstract `factorial` (and, therefore, the foregoing pi-calculus code) never manifests a circularity by using a *model checking* technique. This despite the fact that the model of `factorial` has infinite states. In particular, we are able to decide the deadlock freedom by analyzing finitely many states – precisely three – of `factorial`.

Our solution. We introduce a basic recursive model, called *lam programs* – lam is an acronym for *deadLock Analysis Model* – that are collections of function definitions and a main term to evaluate. For example,

$$\big(\ \texttt{factorial}(r,s) = (r,s)\&(r,t)\&\texttt{factorial}(t,s)\ ,\texttt{factorial}(r,s)\ \big)$$

defines `factorial` and the main term $\texttt{factorial}(r,s)$. Because lam programs feature recursion and dynamic name creation – *e.g.* the free name t in the definition of `factorial` – the model is not finite state (see Section 3).

In this work we address the

Question 1. Is it decidable whether the computations of a lam program will ever produce a circularity?

and the main contribution is the positive answer when programs are *linear recursive*.

To begin the description of our solution, we notice that, if lam programs are non-recursive then detecting circularities is as simple as unfolding the invocations in the main term. In general, as in case of `factorial`, the unfolding may not terminate. Nevertheless, the following two conditions may ease our answer:

(i) the functions in the program are *linear recursive*, that is (mutual) recursions have at most one recursive invocation – such as `factorial`;
(ii) function invocations do not show duplicate arguments and function definitions do not have free names.

When (i) and (ii) hold, as in the program

$$\big(\texttt{f}(x,y,z) = (x,y)\&\texttt{f}(y,z,x),\ \texttt{f}(u,v,w)\big)\ ,$$

recursive functions may be considered as *permutations of names* – technically we define a notion of *associated (per)mutation* – and the corresponding theory [8] guarantees that, by repeatedly applying a same permutation to a tuple of names, at some point, one obtains the initial tuple. This point, which is known as the *order* of the permutation, allows one to define the following algorithm for Question 1:

1. compute the order of the permutation associated to the function in the lam and
2. correspondingly unfold the term to evaluate.

For example, the permutation of `f` has order 3. Therefore, it is possible to stop the evaluation of `f` after the third unfolding (at the state $(u, v)\&(v, w)\&(w, u)$ & $\mathtt{f}(u, v, w)$) because every dependency pair produced afterwards will belong to the relation $(u, v)\&(v, w)\&(w, u)$.

When the constraint (ii) is dropped, as in `factorial`, the answer to Question 1 is not simple anymore. However, the above analogy with permutations has been a source of inspiration for us.

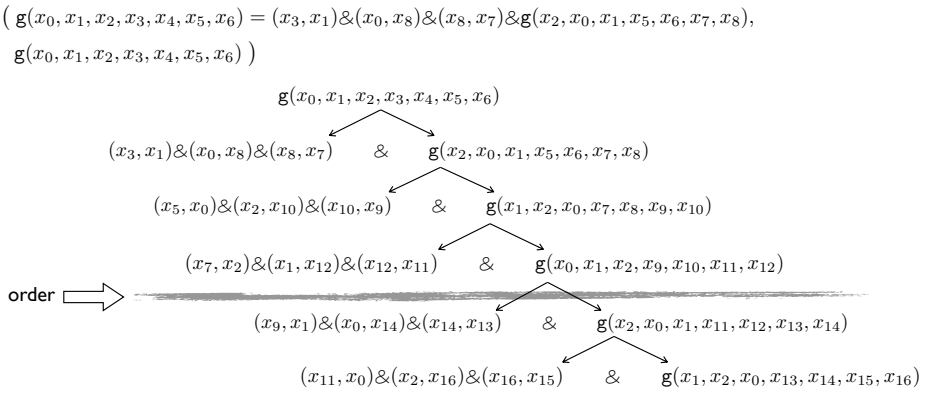

Fig. 1. A lam program and its unfolding

Consider the main term $\mathtt{factorial}(r, s)$. Its evaluation will never display $\mathtt{factorial}(r, s)$ twice, as well as any other invocation in the states, because the first argument of the recursive invocation is free. Nevertheless, we notice that, from the second state – namely $(r, s)\&(r, t)\&\mathtt{factorial}(t, s)$ – onwards, the invocations of `factorial` are not identical, but *may be identified by a map* that

- associates names created in the last evaluation step to past names,
- is the identity on other names.

The definition of this map, called *flashback*, requires that the transformation associated to a lam function, called *mutation*, also records the name creation. In fact, the theory of mutations allows us to map $\mathtt{factorial}(t, s)$ back to $\mathtt{factorial}(r, s)$ by recording that t has been created after r, *e.g.* $r<t$.

We generalize the result about permutation orders (Section 2):

> *by repeatedly applying a same mutation to a tuple of names, at some point we obtain a tuple that is identical, up-to a flashback, to a tuple in the past.*

As for permutations, this point is the *order* of the mutation, which (we prove) it is possible to compute in similar ways.

However, unfolding a function as many times as the order of the associated mutation may not be sufficient for displaying circularities. This is unsurprising because the arguments about mutations and flashbacks focus on function invocations and do not account for dependencies. In the case of lams where (i) and (ii) hold, these arguments were sufficient because permutations reproduce *the same* dependencies of past invocations. In the case of mutations, this is not true anymore as displayed by the function g in Figure 1. This function has order 3 and the first three unfoldings of $\mathtt{g}(x_0, x_1, x_2, x_3, x_4, x_5, x_6)$ are those above the horizontal line. While there is a flashback from $\mathtt{g}(x_0, x_1, x_2, x_9, x_{10}, x_{11}, x_{12})$ to $\mathtt{g}(x_0, x_1, x_2, x_3, x_4, x_5, x_6)$, the pairs produced up-to the third unfolding

$$(x_3, x_1)\&(x_0, x_8)\&(x_8, x_7)\&(x_5, x_0)\&(x_2, x_{10})\&(x_{10}, x_9)$$
$$\&(x_7, x_2)\&(x_1, x_{12})\&(x_{12}, x_{11})$$

do not manifest any circularity. Yet, two additional unfoldings (displayed below the horizontal line of Figure 1), show the circularity

$$(x_0, x_8)\&(x_8, x_7)\&(x_7, x_2)\&(x_2, x_{10})\&(x_{10}, x_9)$$
$$\&(x_9, x_1)\&(x_1, x_{12})\&(x_{12}, x_{11})\&(x_{11}, x_0)\ .$$

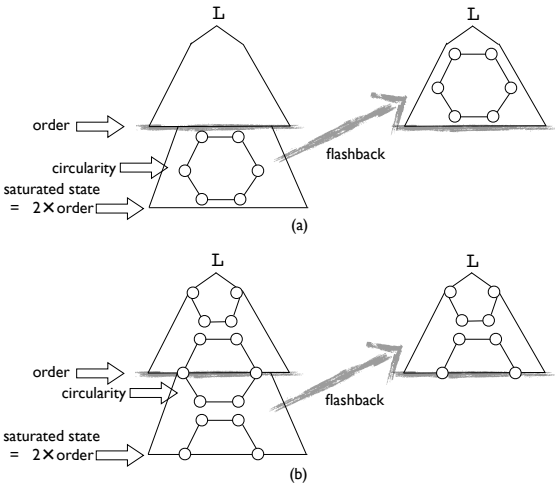

Fig. 2. Flashbacks of circularities

In Section 4 we prove that a sufficient condition for deciding whether a lam program as in Figure 1 will ever produce a circularity is to unfold the function g *up-to two times* the order of the associated mutation – this state will be called *saturated*. If no circularity is manifested in the saturated state then the lam is "circularity-free". This supplement of evaluation is due to the existence of two alternative ways for creating circularities. A first way is when the circularity is given by the dependencies produced by the unfoldings from the order to the saturated state. Then, our theory guarantees that the circularity is also present in the unfolding of g till the order – see Figure 2.a. A second way is when the

dependencies of the circularity are produced by (1) the unfolding till the order *and* by (2) the unfolding from the order till the saturated state – these are the so-called *crossover circularities* – see Figure 2.b. Our theory allows us to map dependencies of the evaluation (2) to those of the evaluation (1) and the flashback may break the circularity – in this case, the evaluation till the saturated state is necessary to collect enough informations. Other ways for creating circularities are excluded. The intuition behind this fact is that the behavior of the function (the dependencies) repeats itself following the same pattern every order-wise unfolding. Thus it is not possible to reproduce a circularity that crosses more than one order without having already a shorter one. The algorithm for detecting circularities in linear recursive lam programs is detailed in Section 5, together with a discussion about its computational cost.

We have prototyped our algorithm [15]. In particular, the prototype (1) uses a (standard but not straightforward) *inference system* that we developed for deriving behavioral types with dependency informations out of `ABS` programs [13] and (2) has an add-on translationg these behavioral types into lams. We have been able to verify an industrial case study developed by SDL Fredhoppper – more than 2600 lines of code – in 31 seconds. Details about our prototype and a comparison with other deadlock analysis tools can be found in Section 6. There is no space in this contribution to discuss the inference system: the interested readers are referred to [13].

2 Generalizing permutations: mutations and flashbacks

Natural numbers are ranged over by a, b, i, j, m, n, ..., possibly indexed. Let V be an infinite set of names, ranged over by $x, y, z, \cdots$. We will use partial order relations on names – relations that are reflexive, antisymmetric, and transitive –, ranged over by $\mathbb{V}, \mathbb{V}', \mathbb{I}, \cdots$. Let $x \in \mathbb{V}$ if, for some y, either $(x, y) \in \mathbb{V}$ or $(y, x) \in \mathbb{V}$. Let also $var(\mathbb{V}) = \{x \mid x \in \mathbb{V}\}$. For notational convenience, we write $\widetilde{x}$ when we refer to a list of names $x_1, \ldots, x_n$.

Let $\mathbb{V} \oplus \widetilde{x}{<}\widetilde{z}$, with $\widetilde{x} \in \mathbb{V}$ and $\widetilde{z} \notin \mathbb{V}$, be the least partial order containing the set $\mathbb{V} \cup \{(y, z) \mid x \in \widetilde{x}$ and $(x, y) \in \mathbb{V}$ and $z \in \widetilde{z}\}$. That is, $\widetilde{z}$ become *maximal names* in $\mathbb{V} \oplus \widetilde{x}{<}\widetilde{z}$. For example,

- $\{(x, x)\} \oplus x{<}z = \{(x, x), (x, z), (z, z)\}$;
- if $\mathbb{V} = \{(x, y), (x', y')\}$ (the reflexive pairs are omitted) then $\mathbb{V} \oplus y{<}z$ is the reflexive and transitive closure of $\{(x, y), (x', y'), (y, z)\}$;
- if $\mathbb{V} = \{(x, y), (x, y')\}$ (the reflexive pairs are omitted) then $\mathbb{V} \oplus x{<}z$ is the reflexive and transitive closure of $\{(x, y), (x, y'), (y, z), (y', z)\}$.

Let $x \leqslant y \in \mathbb{V}$ be $(x, y) \in \mathbb{V}$.

Definition 1. *A* mutation *of a tuple of names, denoted* $[\![a_1, \cdots, a_n]\!]$ *where* $1 \leqslant a_1, \cdots, a_n \leqslant 2 \times n$, *transforms a pair* $\langle \mathbb{V}, (x_1, \cdots, x_n) \rangle$ *into* $\langle \mathbb{V}', (x'_1, \cdots, x'_n) \rangle$ *as follows. Let* $\{b_1, \cdots, b_k\} = \{a_1, \cdots, a_n\} \backslash \{1, 2, \cdots, n\}$ *and let* $z_{b_1}, \cdots, z_{b_k}$ *be* k *pairwise different fresh names. [That is names not occurring either in* $x_1, \cdots, x_n$ *or in* $\mathbb{V}$*.] Then*

– *if* $1 \leqslant a_i \leqslant n$ *then* $x'_i = x_{a_i}$*;*
– *if* $a_i > n$ *then* $x'_i = z_{a_i}$*;*
– $\mathbb{V}' = \mathbb{V} \oplus x_1, \cdots, x_n{<}z_{i_1}, \cdots, z_{i_k}$.

The mutation $[\![a_1, \cdots, a_n]\!]$ *of* $\langle \mathbb{V}, (x_1, \cdots, x_n) \rangle$ *is written* $\langle \mathbb{V}, (x_1, \cdots, x_n) \rangle \xrightarrow{[\![a_1, \cdots, a_n]\!]} \langle \mathbb{V}', (x'_1, \cdots, x'_n) \rangle$ *and the label* $[\![a_1, \cdots, a_n]\!]$ *is omitted when the mutation is clear from the context. Given a mutation* $\mu = [\![a_1, \cdots, a_n]\!]$*, we define the application of* μ *to an index* i*,* $1 \leqslant i \leqslant n$*, as* $\mu(i) = a_i$.

Permutations are mutations $[\![a_1, \cdots, a_n]\!]$ where the elements are pairwise different and belong to the set $\{1, 2, \cdots, n\}$ (e.g. $[\![2, 3, 5, 4, 1]\!]$). In this case the partial order $\mathbb{V}$ never changes and therefore it is useless. Actually, our terminology and statements below are inspired by the corresponding ones for permutations. A mutation differs from a permutation because it can exhibit repeated elements, or even new elements (identified by $n + 1 \leqslant a_i \leqslant 2 \times n$, for some a_i). For example, by successively applying the mutation $[\![2, 3, 6, 1, 1]\!]$ to $\langle \mathbb{V}, (x_1, x_2, x_3, x_4, x_5) \rangle$, with $\mathbb{V} = \{(x_1, x_1), \cdots, (x_5, x_5)\}$ and $\widetilde{x} = x_1, x_2, x_3, x_4, x_5$, we obtain

$$\begin{array}{lll} \langle \mathbb{V}, (x_1, x_2, x_3, x_4, x_5) \rangle & \longrightarrow & \langle \mathbb{V}_1, (x_2, x_3, y_1, x_1, x_1) \rangle \\ & \longrightarrow & \langle \mathbb{V}_2, (x_3, y_1, y_2, x_2, x_2) \rangle \\ & \longrightarrow & \langle \mathbb{V}_3, (y_1, y_2, y_3, x_3, x_3) \rangle \\ & \longrightarrow & \langle \mathbb{V}_4, (y_2, y_3, y_4, y_1, y_1) \rangle \\ & \longrightarrow & \cdots \end{array}$$

where $\mathbb{V}_1 = \mathbb{V} \oplus \widetilde{x}{<}y_1$ and, for $i \geqslant 1$, $\mathbb{V}_{i+1} = \mathbb{V}_i \oplus y_i{<}y_{i+1}$. In this example, 6 identifies a new name to be added at each application of the mutation. The new name created at each step is a maximal one for the partial order.

We observe that, by definition, $[\![2, 3, 6, 1, 1]\!]$ and $[\![2, 3, 7, 1, 1]\!]$ define a same transformation of names. That is, the choice of the natural between 6 and 10 is irrelevant in the definition of the mutation. Similarly for the mutations $[\![2, 3, 6, 1, 6]\!]$ and $[\![2, 3, 7, 1, 7]\!]$.

Definition 2. *Let* $[\![a_1, \cdots, a_n]\!] \approx [\![a'_1, \cdots, a'_n]\!]$ *if there exists a bijective function* f *from* $[n + 1..2 \times n]$ *to* $[n + 1..2 \times n]$ *such that:*

1. $1 \leqslant a_i \leqslant n$ *implies* $a'_i = a_i$*;*
2. $n + 1 \leqslant a_i \leqslant 2 \times n$ *implies* $a'_i = f(a_i)$.

We notice that $[\![2, 3, 6, 1, 1]\!] \approx [\![2, 3, 7, 1, 1]\!]$ and $[\![2, 3, 6, 1, 6]\!] \approx [\![2, 3, 7, 1, 7]\!]$. However $[\![2, 3, 6, 1, 6]\!] \not\approx [\![2, 3, 6, 1, 7]\!]$; in fact these two mutations define different transformations of names.

Definition 3. *Given a partial order* $\mathbb{V}$*, a* $\mathbb{V}$-flashback *is an injective renaming* ρ *on names such that* $\rho(x) \leqslant x \in \mathbb{V}$.

In the above sequence of mutations of $(x_1, x_2, x_3, x_4, x_5)$ there is a $\mathbb{V}_4$-flashback from $(y_2, y_3, y_4, y_1, y_1)$ to $(x_2, x_3, y_1, x_1, x_1)$. In the following, flashbacks will be also applied to tuples: $\rho(x_1, \cdots, x_n) \stackrel{def}{=} (\rho(x_1), \cdots, \rho(x_n))$.

In case of mutations that are permutations, a flashback is the identity renaming and the following statement is folklore. Let μ be a mutation. We write μ^m for the application of μ m times, namely $\langle \mathbb{V}, (x_1, \cdots, x_n) \rangle \xrightarrow{\mu^m} \langle \mathbb{V}', (y_1, \cdots, y_n) \rangle$ abbreviates $\underbrace{\langle \mathbb{V}, (x_1, \cdots, x_n) \rangle \xrightarrow{\mu} \cdots \xrightarrow{\mu} \langle \mathbb{V}', (y_1, \cdots, y_n) \rangle}_{m \text{ times}}$.

Proposition 1. *Let* $\mu = [\![a_1, \cdots, a_n]\!]$ *and*

$$\begin{aligned} \langle \mathbb{V}, (x_1, \cdots, x_n) \rangle &\xrightarrow{\mu} \langle \mathbb{V}', (x'_1, \cdots, x'_n) \rangle \\ &\xrightarrow{\mu^m} \langle \mathbb{V}'', (y_1, \cdots, y_n) \rangle \\ &\xrightarrow{\mu} \langle \mathbb{V}''', (y'_1, \cdots, y'_n) \rangle \end{aligned}$$

If there is a $\mathbb{V}''$*-flashback* ρ *such that* $\rho(y_1, \cdots, y_n) = (x_1, \cdots, x_n)$ *then there is a* $\mathbb{V}'''$*-flashback from* $(y'_1, \cdots, y'_n)$ *to* $(x'_1, \cdots, x'_n)$*.*

Proof. Let ρ' be the relation $y'_i \mapsto x'_i$, for every i. Then

1) ρ' is a mapping: $y'_i = y'_j$ implies $x'_i = x'_j$. In fact, $y'_i = y'_j$ means that either (i) $1 \leqslant a_i, a_j \leqslant n$ or (ii) $a_i, a_j > n$. In subcase (i) $y_{a_i} = y_{a_j}$, by definition of mutation. Therefore $\rho(y_{a_i}) = \rho(y_{a_j})$ that in turn implies $x_{a_i} = x_{a_j}$. From this last equality we obtain $x'_i = x'_j$. In subcase (ii), $a_i = a_j$ and the implication follows by the fact that $[\![a_1, \cdots, a_n]\!]$ is a mutation.

2) ρ' is injective: $x'_i = x'_j$ implies $y'_i = y'_j$. If $x'_i \in \{x_1, \cdots, x_n\}$ then $1 \leqslant a_i, a_j \leqslant n$. Therefore, by the definition of mutation, $x_{a_i} = x_{a_j}$ and, because ρ is a flashback, $y_{a_i} = y_{a_j}$. By this last equation $y'_i = y'_j$. If $x'_i \notin \{x_1, \cdots, x_n\}$ then $a_i > n$ and $a_i = a_j$. Therefore $y'_i = y'_j$ by definition of mutation.

3) ρ' is a flashback: $x'_i \neq y'_i$ implies $x'_i \leqslant y'_i \in \mathbb{V}'''$. If $1 \leqslant a_i \leqslant n$ then $y'_i = y_{a_i}$ and $x'_i = x_{a_i}$. Therefore $y_{a_i} \neq x_{a_i}$ and we conclude by the hypothesis about ρ that $\rho'(y_{a_i})$ satisfies the constraint in the definition of flashback. If $a_i > n$ then $x_1, \cdots, x_n \leqslant x'_i \in \mathbb{V}'$. Since $\rho(y_i) = x_i$, by the hypothesis about ρ, $x_i \leqslant y_i \in \mathbb{V}''$. Therefore, by definition of mutation, $x'_i \leqslant y_i \in \mathbb{V}''$. We derive $x'_i \leqslant y'_i \in \mathbb{V}'''$ by transitivity because $\mathbb{V}'' \subseteq \mathbb{V}'''$ and $y_i \leqslant y'_i \in \mathbb{V}'''$.

The following Theorem 1 generalizes the property that every permutation has an *order*, which is the number of applications that return the initial tuple. In the theory of permutations, the order is the least common multiple, in short *lcm*, of the lengths of the cycles of the permutation. This result is clearly false for mutations because of the presence of duplications and of fresh names. The generalization that holds in our setting uses flashbacks instead of identities. We begin by extending the notion of cycle.

Definition 4 (Cycles and sinks). *Let* $\mu = [\![a_1, \cdots, a_n]\!]$ *be a mutation and let* $1 \leqslant a_{i_1}, \cdots, a_{i_\ell} \leqslant n$ *be pairwise different naturals. Then:*

i. the term $(a_{i_1} \cdots a_{i_\ell})$ *is a* cycle *of* μ *whenever* $\mu(a_{i_j}) = a_{i_{j+1}}$*, with* $1 \leqslant j \leqslant \ell - 1$*, and* $\mu(a_{i_\ell}) = a_{i_1}$ *(i.e.,* $(a_{i_1} \cdots a_{i_\ell})$ *is the ordinary permutation cycle);*
ii. the term $[a_{i_1} \cdots a_{i_{\ell-1}}]_{a_{i_\ell}}$ *is a* bound sink *of* μ *whenever* $a_{i_1} \notin \{a_1, \cdots, a_n\}$*,* $\mu(a_{i_j}) = a_{i_{j+1}}$*, with* $1 \leqslant j \leqslant \ell - 1$*, and* a_{i_ℓ} *belongs to a cycle;*

iii. the term $[a_{i_1} \cdots a_{i_\ell}]_a$*, with* $n < a \leqslant 2 \times n$*, is a* free sink *of* μ *whenever* $a_{i_1} \notin \{a_1, \cdots, a_n\}$ *and* $\mu(a_{i_j}) = a_{i_{j+1}}$*, with* $1 \leqslant j \leqslant \ell - 1$ *and* $\mu(a_{i_\ell}) = a$*.*

The length of a cycle *is the number of elements in the cycle; the* length of a sink *is the number of the elements in the square brackets.*

For example the mutation $(\!|\, 5, 4, 8, 8, 3, 5, 8, 3, 3 \,|\!)$ has cycle $(3, 8)$ and has bound sinks $[1, 5]_3$, $[6, 5]_3$, $[9]_3$, $[2, 4]_8$, and $[7]_8$. The mutation $(\!|\, 6, 3, 1, 8, 7, 1, 8 \,|\!)$ has cycle $(1, 6)$, has bound sink $[2, 3]_1$ and free sinks $[4]_8$ and $[5, 7]_8$.

Cycles and sinks are an alternative description of a mutation. For instance $(3, 8)$ means that the mutation moves the element in position 8 to the element in position 3 and the one in position 3 to the position 8; the free sink $[5, 7]_8$ means that the element in position 7 goes to the position 5, whilst a fresh name goes in position 7.

Theorem 1. *Let* μ *be a mutation,* ℓ *be the lcm of the length of its cycles,* ℓ' *and* ℓ'' *be the lengths of its longest bound sink and free sink, respectively. Let also* $k \stackrel{def}{=} \mathtt{max}\{\ell + \ell',\ \ell''\}$*. Then there exists* $0 \leqslant h < k$ *such that* $\langle \mathbb{V}, (x_1, \cdots, x_n) \rangle \xrightarrow{\mu^h} \langle \mathbb{V}', (y_1, \cdots, y_n) \rangle \xrightarrow{\mu^{k-h}} \langle \mathbb{V}'', (z_1, \cdots, z_n) \rangle$ *and* $\rho(z_1, \cdots, z_n) = (y_1, \cdots, y_n)$*, for some* $\mathbb{V}''$*-flashback* ρ*. The value* k *is called* order of μ *and denoted by* $\mathfrak{o}_\mu$*.*

Proof. Let $\mu = (\!|\, a_1, \cdots, a_n \,|\!)$ be a mutation, and let $A = \{1, 2, \ldots, n\} \backslash \{a_1, \cdots, a_n\}$.

If $A = \varnothing$, then μ is a permutation; hence, by the theory of permutations, the theorem is immediately proved taking ρ as the identity and $h = 0$.

If $A \neq \varnothing$ then let $a \in A$. By definition, a must be the first element of (i) a bound sink or (ii) a free sink of μ. We write either $a \in A_{(i)}$ or $a \in A_{(ii)}$ if a is the first element of a bound or free sink, respectively.

In subcase (i), let ℓ'_a be the length of the bound sink with subscript a' and $\ell_{a'}$ be the length of the cycle of a'. We observe that in $\langle \mathbb{V}, (x_1, \cdots, x_n) \rangle \xrightarrow{\mu^{\ell'_a}} \langle \mathbb{U}, (x'_1, \cdots, x'_n) \rangle \xrightarrow{\mu^{\ell_{a'}}} \langle \mathbb{W}, (x''_1, \cdots, x''_n) \rangle$ we have $x'_{a'} = x''_{a'}$.

In subcase (ii), let ℓ''_a be the length of the free sink. We observe that in $\langle \mathbb{V}, (x_1, \cdots, x_n) \rangle \xrightarrow{\mu^{\ell''_a}} \langle \mathbb{U}, (x'_1, \cdots, x'_n) \rangle$ we have $x_a \leqslant x'_a \in \mathbb{U}$, by definition of mutation.

Let ℓ, ℓ' and ℓ'' as defined in the theorem. Then, if $\ell + \ell' \geqslant \ell''$ we have that $\langle \mathbb{V}, (x_1, \cdots, x_n) \rangle \xrightarrow{\mu^{\ell'}} \langle \mathbb{V}', (y_1, \cdots, y_n) \rangle \xrightarrow{\mu^{\ell}} \langle \mathbb{V}'', (z_1, \cdots, z_n) \rangle$ and $\rho(z_1, \cdots, z_n) = (y_1, \cdots, y_n)$, where $\rho = [z_1 \mapsto y_1, \cdots, z_n \mapsto y_n]$ is a $\mathbb{V}''$-flashback. If $\ell + \ell' < \ell''$ then $\langle \mathbb{V}, (x_1, \cdots, x_n) \rangle \xrightarrow{\mu^{\ell'' - \ell}} \langle \mathbb{V}', (y_1, \cdots, y_n) \rangle \xrightarrow{\mu^{\ell}} \langle \mathbb{V}'', (z_1, \cdots, z_n) \rangle$ and $\rho(z_1, \cdots, z_n) = (y_1, \cdots, y_n)$, where $\rho = [z_1 \mapsto y_1, \cdots, z_n \mapsto y_n]$ is a $\mathbb{V}''$-flashback.

For example, $\mu = (\!|\, 6, 3, 1, 8, 7, 1, 8 \,|\!)$, has a cycle $(1, 6)$, bound sink $[2, 3]_1$ and free sinks $[4]_8$ and $[5, 7]_8$. Therefore $\ell = 2$, $\ell' = 2$ and $\ell'' = 2$. In this case, the values k and h of Theorem 1 are 4 and 2, respectively. In fact, if we

apply the mutation μ four times to the pair $\langle \mathbb{V}, (x_1, x_2, x_3, x_4, x_5, x_6, x_7) \rangle$, where $\mathbb{V} = \{(x_i, x_i) \,|\, 1 \leqslant i \leqslant 7\}$ we obtain

$$\begin{aligned} \langle \mathbb{V}, (x_1, x_2, x_3, x_4, x_5, x_6, x_7) \rangle &\xrightarrow{\mu} \langle \mathbb{V}_1, (x_6, x_3, x_1, y_1, x_7, x_1, y_1) \rangle \\ &\xrightarrow{\mu} \langle \mathbb{V}_2, (x_1, x_1, x_6, y_2, y_1, x_6, y_2) \rangle \\ &\xrightarrow{\mu} \langle \mathbb{V}_3, (x_6, x_6, x_1, y_3, y_2, x_1, y_3) \rangle \\ &\xrightarrow{\mu} \langle \mathbb{V}_4, (x_1, x_1, x_6, y_4, y_3, x_6, y_4) \rangle \end{aligned}$$

where $\mathbb{V}_1 = \mathbb{V} \oplus x_1, x_2, x_3, x_4, x_5, x_6, x_7 < y_1$ and, for $i \geqslant 1$, $\mathbb{V}_{i+1} = \mathbb{V}_i \oplus y_{i-1} < y_i$. We notice that there is a $\mathbb{V}_4$-flashback ρ from $(x_1, x_1, x_6, y_4, y_3, x_6, y_4)$ (produced by μ^4) to $(x_1, x_1, x_6, y_2, y_1, x_6, y_2)$ (produced by μ^2).

3 The language of lams

We use an infinite set of *function names*, ranged over $\mathtt{f}$, $\mathtt{f}'$, $\mathtt{g}$, $\mathtt{g}'$,..., which is disjoint from the set V of Section 2. A *lam program* is a tuple $\big(\mathtt{f}_1(\widetilde{x_1}) = \mathtt{L}_1, \cdots, \mathtt{f}_\ell(\widetilde{x}_\ell) = \mathtt{L}_\ell, \mathtt{L}\big)$ where $\mathtt{f}_i(\widetilde{x}_i) = \mathtt{L}_i$ are *function definitions* and $\mathtt{L}$ is the *main lam*. The syntax of $\mathtt{L}_i$ and $\mathtt{L}$ is

$$\mathtt{L} \quad ::= \quad \mathtt{0} \;\mid\; (x, y) \;\mid\; \mathtt{f}(\widetilde{x}) \;\mid\; \mathtt{L}\&\mathtt{L} \;\mid\; \mathtt{L} + \mathtt{L}$$

Whenever parentheses are omitted, the operation "&" has precedence over " + ". We will shorten $\mathtt{L}_1 \& \cdots \& \mathtt{L}_n$ into $\&_{i \in 1..n} \mathtt{L}_i$. Moreover, we use $\mathtt{T}$ to range over lams that do not contain function invocations.

Let $var(\mathtt{L})$ be the set of names in $\mathtt{L}$. In a function definition $\mathtt{f}(\widetilde{x}) = \mathtt{L}$, $\widetilde{x}$ are the *formal parameters* and the occurrences of names $x \in \widetilde{x}$ in $\mathtt{L}$ are *bound*; the names $var(\mathtt{L}) \backslash \widetilde{x}$ are *free*.

In the syntax of $\mathtt{L}$, the operations "&" and " + " are associative, commutative with $\mathtt{0}$ being the identity. Additionally the following axioms hold ($\mathtt{T}$ does not contain function invocations)

$$\mathtt{T}\&\mathtt{T} = \mathtt{T} \qquad \mathtt{T} + \mathtt{T} = \mathtt{T} \qquad \mathtt{T}\&(\mathtt{L}' + \mathtt{L}'') = \mathtt{T}\&\mathtt{L}' + \mathtt{T}\&\mathtt{L}''$$

and, in the rest of the paper, we will never distinguish equal lams. For instance, $\mathtt{f}(\widetilde{u}) + (x, y)$ and $(x, y) + \mathtt{f}(\widetilde{u})$ will be always identified. These axioms permit to rewrite a lam without function invocations as a *collection* (operation +) *of relations* (elements of a relation are gathered by the operation &).

Proposition 2. *For every* $\mathtt{T}$*, there exist* $\mathtt{T}_1, \cdots, \mathtt{T}_n$ *that are dependencies composed with &, such that* $\mathtt{T} = \mathtt{T}_1 + \cdots + \mathtt{T}_n$.

Remark 1. Lams are intended to be abstract models of programs that highlight the resource dependencies in the reachable states. The lam $\mathtt{T}_1 + \cdots + \mathtt{T}_n$ of Proposition 2 models a program whose possibly infinite set of states $\{\mathtt{S}_1, \mathtt{S}_2, \cdots\}$ is such that the resource dependencies in $\mathtt{S}_i$ are a subset of those in some $\mathtt{T}_{j_i}$, with $1 \leqslant j_i \leqslant n$. With this meaning, generic lams $\mathtt{L}_1 + \cdots + \mathtt{L}_m$ are abstractions of transition systems (a standard model of programming languages), where transitions are ignored and states record the resource dependencies and the function invocations.

Remark 2. The above axioms, such as $\mathtt{T}\&(\mathtt{L}' + \mathtt{L}'') = \mathtt{T}\&\mathtt{L}' + \mathtt{T}\&\mathtt{L}''$ are restricted to terms $\mathtt{T}$ that do not contain function invocations. In fact, $\mathtt{f}(\widetilde{u})\&((x,y) + (y,z)) \neq (\mathtt{f}(\widetilde{u})\&(x,y)) + (\mathtt{f}(\widetilde{u})\&(y,z))$ because the two terms have a different number of occurrences of invocations of $\mathtt{f}$, and this is crucial for linear recursion – see Definition 6.

In the paper, we always assume lam programs $\big(\mathtt{f}_1(\widetilde{x_1}) = \mathtt{L}_1, \cdots, \mathtt{f}_\ell(\widetilde{x_\ell}) = \mathtt{L}_\ell, \mathtt{L}\big)$ to be *well-defined*, namely *(1)* all function names occurring in $\mathtt{L}_i$ and $\mathtt{L}$ are defined; *(2)* the arity of function invocations matches that of the corresponding function definition.

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

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

As usual $\mathfrak{L}[\mathtt{L}]$ is the lam where the hole of $\mathfrak{L}[\ ]$ is replaced by $\mathtt{L}$. The operational semantics of a program $\big(\mathtt{f}_1(\widetilde{x_1}) = \mathtt{L}_1, \cdots, \mathtt{f}_\ell(\widetilde{x_\ell}) = \mathtt{L}_\ell, \mathtt{L}_{\ell+1}\big)$ is a transition system whose *states* are pairs $\big\langle \mathbb{V},\ \mathtt{L} \big\rangle$ and the *transition relation* is the least one satisfying the rule:

$$\frac{\begin{array}{l}\mathtt{f}(\widetilde{x}) = \mathtt{L} \qquad var(\mathtt{L}) \backslash \widetilde{x} = \widetilde{z} \qquad \widetilde{w} \text{ are fresh} \\ \qquad\qquad\quad \mathtt{L}[\widetilde{w}/\widetilde{z}][\widetilde{u}/\widetilde{x}] = \mathtt{L}'\end{array}}{\big\langle \mathbb{V},\ \mathfrak{L}[\mathtt{f}(\widetilde{u})] \big\rangle \longrightarrow \big\langle \mathbb{V} \oplus \widetilde{u}{<}\widetilde{w},\ \mathfrak{L}[\mathtt{L}'] \big\rangle}\ (\textsc{Red})$$

By (RED), a lam $\mathtt{L}$ is evaluated by successively replacing function invocations with the corresponding lam instances. Name creation is handled with a mechanism similar to that of mutations. For example, if $\mathtt{f}(x) = (x,y)\&\mathtt{f}(y)$ and $\mathtt{f}(u)$ occurs in the main lam, then $\mathtt{f}(u)$ is replaced by $(u,v)\&\mathtt{f}(v)$, where v is a *fresh maximal name* in some partial order. The initial state of a program with main lam $\mathtt{L}$ is $\big\langle \mathbb{I}_\mathtt{L},\ \mathtt{L} \big\rangle$, where $\mathbb{I}_\mathtt{L} \stackrel{def}{=} \{(x,x) \mid x \in var(\mathtt{L})\}$.

To illustrate the semantics of the language of lams we discuss three examples:

1. $\big(\mathtt{f}(x,y,z) = (x,y)\&\mathtt{g}(y,z) + (y,z),\ \mathtt{g}(u,v) = (u,v) + (v,u),\ \mathtt{f}(x,y,z)\big)$ and $\mathbb{I} = \{(x,x),(y,y),(z,z)\}$. Then
   $$\begin{array}{l}\big\langle \mathbb{I},\ \mathtt{f}(x,y,z) \big\rangle \longrightarrow \big\langle \mathbb{I},\ (x,y)\&\mathtt{g}(y,z) + (y,z) \big\rangle \\ \qquad\qquad\quad \longrightarrow \big\langle \mathbb{I},\ (x,y)\&(y,z) + \ (x,y)\&(z,y) + (y,z) \big\rangle\end{array}$$
   The lam in the final state *does not contain function invocations*. This is because the above program is not recursive. Additionally, the evaluation of $\mathtt{f}(x,y,z)$ *has not created names*. This is because names in the bodies of $\mathtt{f}(x,y,z)$ and $\mathtt{g}(u,v)$ are bound.
2. $\big(\mathtt{f}'(x) = (x,y)\&\mathtt{f}'(y)\ ,\ \mathtt{f}'(x)\big)$ and $\mathbb{V}_0 = \{(x_0,x_0)\}$. Then
   $$\begin{array}{l}\big\langle \mathbb{V}_0,\ \mathtt{f}'(x_0) \big\rangle \\ \longrightarrow \big\langle \mathbb{V}_1,\ (x_0,x_1)\&\mathtt{f}'(x_1) \big\rangle \\ \longrightarrow \big\langle \mathbb{V}_2,\ (x_0,x_1)\&(x_1,x_2)\&\mathtt{f}'(x_2) \big\rangle \\ \longrightarrow^n \big\langle \mathbb{V}_{n+2},\ (x_0,x_1)\&\cdots\&(x_{n+1},x_{n+2})\&\mathtt{f}'(x_{n+2}) \big\rangle\end{array}$$

where $\mathbb{V}_{i+1} = \mathbb{V}_i \oplus x_i{<}x_{i+1}$. In this case, the states grow in the number of dependencies as the evaluation progresses. This growth *is due to the presence of a free name* in the definition of $\mathtt{f}'$ that, as said, corresponds to generating a fresh name at every recursive invocation.

3. $\big(\mathtt{f}''(x) = (x,x') + (x,x')\&\mathtt{f}''(x'),\ \mathtt{f}''(x_0)\big)$ and $\mathbb{V}_0 = \{(x_0,x_0)\}$. Then
$\big\langle \mathbb{V}_0,\ \mathtt{f}''(x_0)\big\rangle$
$\longrightarrow \big\langle \mathbb{V}_1, (x_0,x_1) + (x_0,x_1)\&\mathtt{f}''(x_1)\big\rangle$
$\longrightarrow \big\langle \mathbb{V}_2, (x_0,x_1) + (x_0,x_1)\&(x_1,x_2) + (x_0,x_1)\&(x_1,x_2)\&\mathtt{f}''(x_2)\big\rangle$
$\longrightarrow^n \big\langle \mathbb{V}_{n+2}, (x_0,x_1) + \cdots + (x_0,x_1)\&\cdots\&(x_{n+1},x_{n+2})\&\mathtt{f}''(x_{n+2})\big\rangle$
where $\mathbb{V}_{i+1}$ are as before. In this case, the states grow in the number of " + "-terms, which become larger and larger as the evaluation progresses.

The semantics of the language of lams is nondeterministic because of the choice of the invocation to evaluate. However, lams enjoy a diamond property *up-to bijective renaming of (fresh) names.*

Proposition 3. *Let $\imath$ be a bijective renaming and $\imath(\mathbb{V}) = \{(\imath(x),\imath(y)) \mid (x,y) \in \mathbb{V}\}$. Let also $\big\langle \mathbb{V},\ \mathtt{L}\big\rangle \longrightarrow \big\langle \mathbb{V}',\ \mathtt{L}'\big\rangle$ and $\big\langle \imath(\mathbb{V}),\ \mathtt{L}[^{\imath(\widetilde{x})}/_{\widetilde{x}}]\big\rangle \longrightarrow \big\langle \mathbb{V}'',\ \mathtt{L}''\big\rangle$, where $\widetilde{x} = var(\mathbb{V})$. Then*

(i) either there exists a bijective renaming $\imath'$ such that

$$\big\langle \mathbb{V}'',\ \mathtt{L}''\big\rangle = \big\langle \imath(\mathbb{V}'),\ \mathtt{L}'[^{\imath(\widetilde{x'})}/_{\widetilde{x'}}]\big\rangle,$$

where $\widetilde{x}' = var(\mathbb{V}')$,

(ii) or there exist $\mathtt{L}'''$ and a bijective renaming $\imath'$ such that $\big\langle \mathbb{V}',\ \mathtt{L}'\big\rangle \longrightarrow \big\langle \mathbb{V}''',\ \mathtt{L}'''\big\rangle$ and $\big\langle \mathbb{V}'',\ \mathtt{L}''\big\rangle \longrightarrow \big\langle \imath'(\mathbb{V}'''),\ \mathtt{L}'''[^{\imath'(\widetilde{z})}/_{\widetilde{z}}]\big\rangle$, where $\widetilde{z} = var(\mathbb{V}''')$.

The informative operational semantics. In order to detect the circularity-freedom, our technique computes a lam till every function therein has been adequately unfolded (up-to twice the order of the associated mutation). This is formalized by switching to an "informative" operational semantics where basic terms (dependencies and function invocations) are labelled by so-called *histories.*

Let a *history*, ranged over by $\alpha, \beta, \cdots$, be a sequence of function names $\mathtt{f}_{i_1}\mathtt{f}_{i_2}\cdots\mathtt{f}_{i_n}$. We write $\mathtt{f} \in \alpha$ if $\mathtt{f}$ occurs in α. We also write α^n for $\underbrace{\alpha\cdots\alpha}_{n \text{ times}}$. Let $\alpha \preceq \beta$ if there is α' such that $\alpha\alpha' = \beta$. The symbol ε denotes the empty history.

The informative operational semantics is a transition system whose states are tuples $\big\langle \mathbb{V},\ ^{\mathfrak{h}}\mathbb{F},\ \mathbb{L}\big\rangle$ where $^{\mathfrak{h}}\mathbb{F}$ is a set of function invocations with histories and $\mathbb{L}$, called *informative lam*, is a term as $\mathtt{L}$, except that pairs and function invocations are indexed by histories, i.e. $^{\alpha}(x,y)$ and $^{\alpha}\mathtt{f}(\widetilde{u})$, respectively.

Let

$$addh(\alpha,\mathtt{L}) \stackrel{def}{=} \begin{cases} ^{\alpha}(x,y) & \text{if } \mathtt{L} = (x,y) \\ ^{\alpha}\mathtt{f}(\widetilde{x}) & \text{if } \mathtt{L} = \mathtt{f}(\widetilde{x}) \\ addh(\alpha,\mathtt{L}')\&addh(\alpha,\mathtt{L}'') & \text{if } \mathtt{L} = \mathtt{L}'\&\mathtt{L}'' \\ addh(\alpha,\mathtt{L}') + addh(\alpha,\mathtt{L}'') & \text{if } \mathtt{L} = \mathtt{L}' + \mathtt{L}'' \end{cases}$$

For example $addh(\mathtt{fl}, (x_4, x_2)\&\mathtt{f}(x_2, x_3, x_4, x_5)) = {}^{\mathtt{fl}}(x_4, x_2)\&\,{}^{\mathtt{fl}}\mathtt{f}(x_2, x_3, x_4, x_5)$. Let also ${}^{\flat}\mathfrak{L}[\ ]$ be a lam context with histories (dependency pairs and function invocations are labelled by histories, the definition is similar to $\mathfrak{L}[\ ]$).

The informative transition relation is the least one such that

(RED+)

$$\frac{\mathtt{f}(\widetilde{x}) = \mathtt{L} \qquad var(\mathtt{L})\backslash\widetilde{x} = \widetilde{z} \qquad \widetilde{w} \text{ are fresh} \quad \mathtt{L}[\widetilde{w}/\widetilde{z}][\widetilde{u}/\widetilde{x}] = \mathtt{L}'}{\big\langle \mathbb{V},\, {}^{\flat}\mathbb{F},\, {}^{\flat}\mathfrak{L}[{}^{\alpha}\mathtt{f}(\widetilde{u})]\big\rangle \quad\longrightarrow\quad \big\langle \mathbb{V} \oplus \widetilde{u}{<}\widetilde{w},\, {}^{\flat}\mathbb{F} \cup \{{}^{\alpha}\mathtt{f}(\widetilde{u})\},\, {}^{\flat}\mathfrak{L}[addh(\alpha\mathtt{f}, \mathtt{L}')]\big\rangle}$$

When $\big\langle \mathbb{V},\, {}^{\flat}\mathbb{F},\, \mathbb{L}\big\rangle \longrightarrow \big\langle \mathbb{V}',\, {}^{\flat}\mathbb{F}',\, \mathbb{L}'\big\rangle$ by applying (RED+) to ${}^{\alpha}\mathtt{f}(\widetilde{u})$, we say that the term ${}^{\alpha}\mathtt{f}(\widetilde{u})$ *is evaluated in the reduction*. The initial informative state of a program with main lam $\mathtt{L}$ is $\big\langle \mathbb{I}_{\mathtt{L}},\, \varnothing,\, addh(\varepsilon, \mathtt{L})\big\rangle$.

For example, the `flh`-program

$$\begin{array}{l} \big(\ \mathtt{f}(x,y,z,u) \quad = (x,z)\&\mathtt{l}(u,y,z)\ , \\ \quad \mathtt{l}(x,y,z) \quad = (x,y)\&\mathtt{f}(y,z,x,u)\ , \\ \quad \mathtt{h}(x,y,z,u) \quad = (z,x)\&\mathtt{h}(x,y,z,u)\&\mathtt{f}(x,y,z,u)\ , \\ \quad \mathtt{h}(x_1,x_2,x_3,x_4)\ \big) \end{array}$$

has an (informative) evaluation

$$\begin{array}{l} \big\langle \mathbb{I}_{\mathtt{L}},\, \varnothing,\, {}^{\varepsilon}\mathtt{h}(x_1,x_2,x_3,x_4)\big\rangle \\ \longrightarrow \big\langle \mathbb{I}_{\mathtt{L}},\, {}^{\flat}\mathbb{F},\, \mathbb{L}\,\&\,{}^{\mathtt{h}}\mathtt{f}(x_1,x_2,x_3,x_4)\big\rangle \\ \longrightarrow \big\langle \mathbb{I}_{\mathtt{L}},\, {}^{\flat}\mathbb{F}_1,\, \mathbb{L}\,\&\,{}^{\mathtt{hf}}(x_1,x_3)\,\&\,{}^{\mathtt{hf}}\mathtt{l}(x_4,x_2,x_3)\big\rangle \\ \longrightarrow \big\langle \mathbb{I}_{\mathtt{L}} \oplus x_4{<}x_5,\, {}^{\flat}\mathbb{F}_2,\, \mathbb{L}'\,\&\,{}^{\mathtt{hfl}}(x_4,x_2)\,\&\,{}^{\mathtt{hfl}}\mathtt{f}(x_2,x_3,x_4,x_5)\big\rangle, \end{array}$$

where

$$\begin{array}{ll} \mathbb{L} & = {}^{\mathtt{h}}(x_3,x_1)\&{}^{\mathtt{h}}\mathtt{h}(x_1,x_2,x_3,x_4) \\ \mathbb{L}' & = \mathbb{L}\&{}^{\mathtt{hf}}(x_1,x_3) \\ {}^{\flat}\mathbb{F} & = \{{}^{\varepsilon}\mathtt{h}(x_1,x_2,x_3,x_4)\} \\ {}^{\flat}\mathbb{F}_1 & = {}^{\flat}\mathbb{F} \cup \{{}^{\mathtt{h}}\mathtt{f}(x_1,x_2,x_3,x_4)\} \\ {}^{\flat}\mathbb{F}_2 & = {}^{\flat}\mathbb{F}_1 \cup \{{}^{\mathtt{hf}}\mathtt{l}(x_4,x_2,x_3)\}. \end{array}$$

There is a strict correspondence between the non-informative and informative semantics that is crucial for the correctness of our algorithm in Section 5. Let $[\![\cdot]\!]$ be an *eraser map* that takes an informative lam and removes the histories. The formal definition is omitted because it is straightforward.

Proposition 4. *1. If* $\big\langle \mathbb{V}, {}^{\flat}\mathbb{F}, \mathbb{L}\big\rangle \longrightarrow \big\langle \mathbb{V}', {}^{\flat}\mathbb{F}', \mathbb{L}'\big\rangle$ *then* $\big\langle \mathbb{V}, [\![\mathbb{L}]\!]\big\rangle \longrightarrow \big\langle \mathbb{V}', [\![\mathbb{L}']\!]\big\rangle$*;*
2. If $\big\langle \mathbb{V}, [\![\mathbb{L}]\!]\big\rangle \longrightarrow \big\langle \mathbb{V}', \mathtt{L}'\big\rangle$ *then there are* ${}^{\flat}\mathbb{F}$*,* ${}^{\flat}\mathbb{F}'$*,* $\mathbb{L}'$ *such that* $[\![\mathbb{L}']\!] = \mathtt{L}'$ *and* $\big\langle \mathbb{V}, {}^{\flat}\mathbb{F}, \mathbb{L}\big\rangle \longrightarrow \big\langle \mathbb{V}', {}^{\flat}\mathbb{F}', \mathbb{L}'\big\rangle$*.*

Circularities. Lams record sets of relations on names. The following function $\flat(\cdot)$, called *flattening*, makes explicit these relations

$$\begin{array}{c} \flat(\mathtt{0}) = \mathtt{0}, \qquad \flat((x,y)) = (x,y), \qquad \flat(\mathtt{f}(\widetilde{x})) = \mathtt{0}, \\ \flat(\mathtt{L}\&\mathtt{L}') = \flat(\mathtt{L})\&\flat(\mathtt{L}'), \qquad \flat(\mathtt{L} + \mathtt{L}') = \flat(\mathtt{L}) + \flat(\mathtt{L}'). \end{array}$$

For example, if $\mathtt{L} = \mathtt{f}(x,y,z) + (x,y)\&\mathtt{g}(y,z)\&\mathtt{f}(u,y,z) + \mathtt{g}(u,v)\&(u,v) + (v,u)$ then $\flat(\mathtt{L}) = (x,y) + (u,v) + (v,u)$. That is, there are three relations in $\mathtt{L}$: $\{(x,y)\}$ and $\{(u,v)\}$ and $\{(v,u)\}$. By Proposition 2, $\flat(\mathtt{L})$ returns, up-to the

lam axioms, sequences of (pairwise different) &-compositions of dependencies. The operation $\flat(\cdot)$ may be extended to informative lams $\mathbb{L}$ in the obvious way: $\flat({}^{\alpha}(x,y)) = {}^{\alpha}(x,y)$ and $\flat({}^{\alpha}\mathtt{f}(\widetilde{x})) = \mathtt{0}$.

Definition 5. *A lam* $\mathtt{L}$ has a circularity *if*

$$\flat(\mathtt{L}) = (x_1,x_2)\&(x_2,x_3)\&\cdots\&(x_m,x_1)\&\mathtt{T}' + \mathtt{T}''$$

for some $x_1,\cdots,x_m$. *A state* $\langle \forall,\ \mathtt{L}\rangle$ *has a circularity if* $\mathtt{L}$ *has a circularity. Similarly for an informative lam* $\mathbb{L}$.

The final state of the `fgh`-program computation has a circularity; another function displaying a circularity is `g` in Section 1. None of the states in the examples 1, 2, 3 at the beginning of this section has a *circularity*.

4 Linear recursive lams and saturated states

This section develops the theory that underpins the algorithm of Section 5. In order to lightening the section, the technical details have been moved in Appendix B.

We restrict our arguments to (mutually) recursive lam programs. In fact, circularity analysis in non-recursive programs is trivial: it is sufficient to evaluate all the invocations till the final state and verify the presence of circularities therein. A further restriction allows us to simplify the arguments without loosing in generality (*cf.* the definition of saturation): we assume that every function is (mutually) recursive. We may reduce to this case by expanding function invocation of non-(mutually) recursive functions (and removing their definitions).

Linear recursive functions and mutations. Our decision algorithm relies on interpreting recursive functions as mutations. This interpretation is not always possible: the recursive functions that have an associated mutation are the linear recursive ones, as defined below.

The technique for dealing with the general case is briefly discussed in Section 8 and is detailed in Appendix C.

Definition 6. *Let* $\big(\mathtt{f}_1(\widetilde{x_1}) = \mathtt{L}_1,\cdots,\mathtt{f}_\ell(\widetilde{x_\ell}) = \mathtt{L}_\ell,\mathtt{L}\big)$ *be a lam program. A sequence* $\mathtt{f}_{i_0}\mathtt{f}_{i_1}\cdots\mathtt{f}_{i_k}$ *is called a* recursive history of $\mathtt{f}_{i_0}$ *if (a) the function names are pairwise different and (b) for every* $0 \leqslant j \leqslant k$, $\mathtt{L}_{i_j}$ *contains one invocation of* $\mathtt{f}_{i_{j+1\%k}}$ *(the operation % is the remainder of the division).*

The lam program is linear recursive *if (a) every function name has a unique recursive history and (b) if* $\mathtt{f}_{i_0}\mathtt{f}_{i_1}\cdots\mathtt{f}_{i_k}$ *is a recursive history then, for every* $0 \leqslant j \leqslant k$, $\mathtt{L}_{i_j}$ *contains* exactly *one invocation of* $\mathtt{f}_{i_{j+1\%k}}$.

For example, the program

$$\big(\ \mathtt{f}_1(x,y) = (x,y)\&\mathtt{f}_1(y,z)\&\mathtt{f}_2(y) + \mathtt{f}_2(z)\ ,\mathtt{f}_2(y) = (y,z)\&\mathtt{f}_2(z)\ ,\mathtt{L}\quad\big)$$

is linear recursive. On the contrary

$$\big(\mathtt{f}(x) = (x,y)\&\mathtt{g}(x)\ ,\ \mathtt{g}(x) = (x,y)\&\mathtt{f}(x) + \mathtt{g}(y)\ ,\ \mathtt{L}\ \big)$$

is not linear recursive because $\mathtt{g}$ has two recursive histories, namely $\mathtt{g}$ and $\mathtt{gf}$.

Linearity allows us to associate a *unique* mutation to every function name. To compute this mutation, let $\mathtt{H}$ range over sequences of function invocations. We use the following two rules:

$$\frac{\mathtt{f}_i\alpha \models \varepsilon \quad \mathtt{f}_i(\widetilde{x}_i) = \mathtt{L}_i}{\alpha \models \mathtt{f}_i(\widetilde{x}_i)} \qquad \frac{\begin{array}{c}\mathtt{f}_j\alpha \models \mathtt{H}\mathtt{f}_i(\widetilde{x}) \quad \mathtt{f}_i(\widetilde{x}_i) = \mathtt{L}_i \\ var(\mathtt{L}_i)\backslash\widetilde{x}_i = \widetilde{z} \qquad \widetilde{w} \text{ are fresh} \\ \mathfrak{L}[\mathtt{f}_j(\widetilde{y})] = \mathtt{L}_i[\widetilde{w}/\widetilde{z}][\widetilde{x}/\widetilde{x}_i]\end{array}}{\alpha \models \mathtt{H}\mathtt{f}_i(\widetilde{x})\mathtt{f}_j(\widetilde{y})}$$

Let $\varepsilon \models \mathtt{f}(x_1, \cdots, x_n)\cdots\mathtt{f}(x'_1, \cdots, x'_n)$ be the final judgment of the proof tree with leaf $\mathtt{f}\alpha\mathtt{f} \models \varepsilon$, where $\mathtt{f}\alpha$ is the recursive history of $\mathtt{f}$. Let also $x'_1, \cdots, x'_n \backslash x_1, \cdots, x_n = z_1, \cdots, z_k$. Then the *mutation of* $\mathtt{f}$, written $\mu_{\mathtt{f}} = \langle\!\langle\, a_1, \cdots, a_n \,\rangle\!\rangle$ is defined by

$$a_i = \begin{cases} j & \text{if } x'_i = x_j \\ \\ n+j & \text{if } x'_i = z_j \end{cases}$$

Let $\mathfrak{o}_{\mathtt{f}}$, called *order of the function* $\mathtt{f}$, be the order of $\mu_{\mathtt{f}}$. For example, in the $\mathtt{flh}$-program, the recursive history of $\mathtt{f}$ is $\mathtt{fl}$ and, applying the algorithm above to $\mathtt{flf} \models \varepsilon$, we get $\varepsilon \models \mathtt{f}(x,y,z,u)\mathtt{l}(u,y,z)\mathtt{f}(y,z,u,v)$. The mutation of $\mathtt{f}$ is $\langle\!\langle\, 2,3,4,5 \,\rangle\!\rangle$ and $\mathfrak{o}_{\mathtt{f}} = 4$. Analogously we can compute $\mathfrak{o}_{\mathtt{l}} = 3$ and $\mathfrak{o}_{\mathtt{h}} = 1$.

Saturation. In the remaining part of the section we assume a fixed linear recursive program $\big(\mathtt{f}_1(\widetilde{x_1}) = \mathtt{L}_1, \cdots, \mathtt{f}_\ell(\widetilde{x_\ell}) = \mathtt{L}_\ell, \mathtt{L}\big)$ and let $\mathfrak{o}_{\mathtt{f}_1}, \cdots, \mathfrak{o}_{\mathtt{f}_\ell}$ be the orders of the corresponding functions.

Definition 7. *A history α is*

$\mathtt{f}$-*complete*
: *if $\alpha = \beta^{\mathfrak{o}_{\mathtt{f}}}$, where β is the recursive history of $\mathtt{f}$. We say that α is* complete *when it is $\mathtt{f}$-complete, for some $\mathtt{f}$.*

$\mathtt{f}$-*saturating*
: *if $\alpha = \beta_1 \cdots \beta_{n-1}\alpha_n^2$, where $\beta_i \preceq (\alpha_i)^2$, with α_i complete, and α_n $\mathtt{f}$-complete. We say that α is* saturating *when it is $\mathtt{f}$-saturating, for some $\mathtt{f}$.*

In the $\mathtt{flh}$-program, $\mathfrak{o}_{\mathtt{f}} = 4$, $\mathfrak{o}_{\mathtt{l}} = 3$, and $\mathfrak{o}_{\mathtt{h}} = 1$, and the recursive histories of $\mathtt{f}$, $\mathtt{l}$ and $\mathtt{h}$ are equal to $\mathtt{fl}$, to $\mathtt{lf}$ and to $\mathtt{h}$, respectively. Then $\alpha = (\mathtt{fl})^4$ is the $\mathtt{f}$-complete history and $\mathtt{h}^2(\mathtt{fl})^8$ and $\mathtt{h}(\mathtt{fl})^8$ are $\mathtt{f}$-saturating.

The following proposition is an important consequence of the theory of mutations (Theorem 1) and the semantics of lams (and their axioms). In particular, it states that, if a function invocation $\mathtt{f}_0(\widetilde{u_0})$ is unfolded up to the order of $\mathtt{f}_0$ then (i) the last invocation $\mathtt{f}_0(\widetilde{v})$ may be mapped back to a previous invocation by a flashback and (ii) the same flashback also maps back dependencies created by the unfolding of $\mathtt{f}_0(\widetilde{v})$.

Proposition 5. *Let $\beta = \mathtt{f}_0\mathtt{f}_1\cdots\mathtt{f}_n$ be $\mathtt{f}_0$-complete and let*

$$\big\langle \mathbb{V},\, {}^{\natural}\mathbb{F},\, {}^{\natural}\mathfrak{L}_0[{}^{\alpha}\mathtt{f}_0(\widetilde{u_0})] \big\rangle \longrightarrow^{n+1} \big\langle \mathbb{V}',\, {}^{\natural}\mathbb{F}',\, {}^{\natural}\mathfrak{L}_0[{}^{\natural}\mathfrak{L}_1[\cdots {}^{\natural}\mathfrak{L}_n[{}^{\alpha\mathtt{f}_0\cdots\mathtt{f}_n}\mathtt{f}_0(\widetilde{u_{n+1}})]\cdots]] \big\rangle$$

where ${}^{\flat}\mathbb{F}' = {}^{\flat}\mathbb{F} \cup \{{}^{\alpha}\mathtt{f}_0(\widetilde{u_0}), {}^{\alpha\mathtt{f}_0}\mathtt{f}_1(\widetilde{u_1}), \cdots, {}^{\alpha\mathtt{f}_0\cdots\mathtt{f}_{n-1}}\mathtt{f}_n(\widetilde{u_n})\}$ *and* $\mathtt{f}_i(\widetilde{u_i}) = \mathtt{L}'_i$ *and* $addh(\alpha\mathtt{f}_0\cdots\mathtt{f}_i, \mathtt{L}'_i) = {}^{\flat}\mathfrak{L}_i[{}^{\alpha\mathtt{f}_0\cdots\mathtt{f}_i}\mathtt{f}_{i+1}(\widetilde{u_{i+1}})]$ *(unfolding of the functions in the complete history of* $\mathtt{f}_0$*). Then there is a* ${}^{\alpha\mathtt{f}_0\cdots\mathtt{f}_{h-1}}\mathtt{f}_h(\widetilde{u_h}) \in {}^{\flat}\mathbb{F}'$ *and a* $\mathbb{V}'$*-flashback* ρ *such that*

1. $\mathtt{f}_0(\rho(\widetilde{u_{n+1}})) = \mathtt{f}_h(\widetilde{u_h})$ *(hence* $\mathtt{f}_0 = \mathtt{f}_h$*);*
2. *let* $\mathtt{f}_0(\widetilde{u_{n+1}}) = \mathtt{L}$ *and* $\flat(\mathtt{L}) = \mathtt{T}_1 + \cdots + \mathtt{T}_k$ *and* $\flat({}^{\flat}\mathfrak{L}_0[{}^{\flat}\mathfrak{L}_1[\cdots{}^{\flat}\mathfrak{L}_n[{}^{\alpha\beta}\mathtt{f}_0(\widetilde{u_{n+1}})]\cdots]]) = {}^{\flat}\mathtt{T}'_1 + \cdots + {}^{\flat}\mathtt{T}'_{k'}$*. Then, for every* $1 \leqslant i \leqslant k$*, there exists* $1 \leqslant j \leqslant k'$ *such that* ${}^{\flat}\mathtt{T}'_j = addh(\alpha\mathtt{f}_0\cdots\mathtt{f}_{h-1}, \mathtt{T}_i)\&{}^{\flat}\mathtt{T}''_j$*, for some* ${}^{\flat}\mathtt{T}''_j$*.*

The notion of $\mathtt{f}$-saturating will be used to define a "saturated" state, i.e., a state where the evaluation of programs may safely (as regards circularities) stop.

Definition 8. *An informative lam* $\langle \mathbb{V}, {}^{\flat}\mathbb{F}, \mathbb{L} \rangle$ *is* saturated *when, for every* ${}^{\flat}\mathfrak{L}[\,]$ *and* $\mathtt{f}(\widetilde{u})$ *such that* $\mathbb{L} = {}^{\flat}\mathfrak{L}[{}^{\alpha}\mathtt{f}(\widetilde{u})]$*,* α *has a saturating prefix.*

It is easy to check that the following informative lam generated by the computation of the `flh`-program is saturated:

$$\langle \mathbb{V}_7, {}^{\flat}\mathbb{F}, {}^{\mathtt{h}^2}\mathtt{h}(x_1,x_2,x_3,x_4) \;\&\; \&_{0\leqslant i\leqslant 8}{}^{\mathtt{hf}(\mathtt{lf})^i}(x_{i+1},x_{i+3}) \;\&\; \&_{0\leqslant i\leqslant 8}{}^{\mathtt{h}(\mathtt{fl})^i}(x_{i+3},x_{i+1}) \;\&\; {}^{\mathtt{h}(\mathtt{fl})^8}\mathtt{f}(x_9,x_{10},x_{11},x_{12})\rangle,$$

where $\mathbb{V}_{i+1} = \mathbb{V}_i \oplus x_{i+4}{<}x_{i+5}$, and

$$\begin{aligned}{}^{\flat}\mathbb{F} = {} & \{{}^{\varepsilon}\mathtt{h}(x_1,x_2,x_3,x_4), {}^{\mathtt{h}}\mathtt{h}(x_1,x_2,x_3,x_4)\} \\ & \cup \{{}^{\mathtt{h}(\mathtt{fl})^i}\mathtt{f}(x_{i+1},x_{i+2},x_{i+3},x_{i+4}) \mid 0 \leqslant i \leqslant 7\} \\ & \cup \{{}^{\mathtt{hf}(\mathtt{lf})^i}\mathtt{l}(x_{i+4},x_{i+2},x_{i+3}) \mid 0 \leqslant i \leqslant 7\}.\end{aligned}$$

Every preliminary statement is in place for our key theorem that details the mapping of circularities created by transitions of saturated states to past circularities.

Theorem 2. *Let* $\langle \mathbb{I}_{\mathtt{L}}, \varnothing, addh(\varepsilon, \mathtt{L}) \rangle \longrightarrow^* \langle \mathbb{V}, {}^{\flat}\mathbb{F}, \mathbb{L} \rangle$ *and* $\langle \mathbb{V}, {}^{\flat}\mathbb{F}, \mathbb{L} \rangle$ *be a saturated state. If* $\langle \mathbb{V}, {}^{\flat}\mathbb{F}, \mathbb{L} \rangle \longrightarrow \langle \mathbb{V}', {}^{\flat}\mathbb{F}', \mathbb{L}' \rangle$ *then*

1. $\langle \mathbb{V}', {}^{\flat}\mathbb{F}', \mathbb{L}' \rangle$ *is saturated;*
2. *if* $\mathbb{L}'$ *has a circularity then* $\mathbb{L}$ *has already a circularity.*

Proof. (*Sketch*) Item 1. directly follows from Proposition 5. However, this proposition is not sufficient to guarantee that circularities created in saturated states are mapped back to past ones. In particular, the interesting case is the one of *crossover circularities*, as discussed in Section 1. Therefore, let

$${}^{\alpha_1}(x_1,x_2), \cdots, {}^{\alpha_{h-1}}(x_{h-1},x_h), {}^{\alpha_h}(x_h,x_{h+1}), \cdots, {}^{\alpha_n}(x_n,x_1)$$

be a circularity in $\mathbb{L}'$ such that ${}^{\alpha_h}(x_h,x_{h+1}), \cdots, {}^{\alpha_n}(x_n,x_1)$ were already present in $\mathbb{L}$. Proposition 5 guarantees the existence of a flashback ρ that maps ${}^{\alpha_1}(x_1,x_2)\&\cdots\&{}^{\alpha_{h-1}}(x_{h-1},x_h)$ to ${}^{\alpha_1}(\rho(x_1),\rho(x_2))\&\cdots\&{}^{\alpha_{h-1}}(\rho(x_{h-1}),\rho(x_h))$. However, it is possible that

$$^{\alpha_1}(\rho(x_1),\rho(x_2))\&\cdots\&^{\alpha_{h-1}}(\rho(x_{h-1}),\rho(x_h))\&^{\alpha_h}(x_h,x_{h+1})\&\cdots\&^{\alpha_n}(x_n,x_1)$$

is no more a circularity because, for example, $\rho(x_h) \neq x_h$ (assume that $\rho(x_1) = x_1$). Let us discuss this issue. The hypothesis of saturation guarantees that transitions produce histories $\alpha^2\beta$, where α is complete. Additionally, $\alpha_1, \cdots, \alpha_{h-1}$ must be equal because they have been created by $\langle \mathbb{V}, {}^{\mathfrak{h}}\mathbb{F}, \mathbb{L}\rangle \longrightarrow \langle \mathbb{V}', {}^{\mathfrak{h}}\mathbb{F}', \mathbb{L}'\rangle$. For simplicity, let $\beta = \mathtt{f}$ and $\alpha = \mathtt{f}\alpha'$. Therefore, by Proposition 5, ρ maps $^{\alpha^2\mathtt{f}}(x_1,x_2)\&\cdots\&\,^{\alpha^2\mathtt{f}}(x_{h-1},x_h)$ to $^{\alpha\mathtt{f}}(\rho(x_1),\rho(x_2))\&\cdots\&^{\alpha\mathtt{f}}(\rho((x_{h-1}),\rho(x_h))$ and, $\rho(x_h) \neq x_h$ when x_h is created by the computation evaluating functions in α'.

To overcome this problem, it is possible to demonstrate using a statement similar to (but stronger than) Proposition 5 that ρ maps $^{\alpha_h}(x_h,x_{h+1})\ \&\ \cdots\ \&^{\alpha_n}(x_n,x_1)$ to $^{[\alpha_h]}(\rho(x_h),\rho(x_{h+1}))\&\cdots\&^{[\alpha_n]}(\rho(x_n),\rho(x_1))$ where $[\alpha_i]$ are "kernels" of α_i where every γ^k in α_i, with γ a complete history and $k \geqslant 2$, is replaced by γ. The proof terminates by demonstrating that the term

$$\begin{aligned}&^{\alpha\mathtt{f}}(\rho(x_1),\rho(x_2))\&\cdots\&^{\alpha\mathtt{f}}(\rho((x_{h-1}),\rho(x_h))\\&\quad\&^{[\alpha_h]}(\rho(x_h),\rho(x_{h+1}))\&\cdots\&^{[\alpha_n]}(\rho(x_n),\rho(x_1))\end{aligned}$$

is in $\mathbb{L}$ (and it is a circularity).

5 The decision algorithm for detecting circularities in linear recursive lams

The algorithm for deciding the circularity-freedom problem in linear recursive lam programs takes as input a lam program $\big(\mathtt{f}_1(\widetilde{x_1}) = \mathtt{L}_1, \cdots, \mathtt{f}_\ell(\widetilde{x_\ell}) = \mathtt{L}_\ell, \mathtt{L}\big)$ and performs the following steps:

Step 1: *find recursive histories*. By parsing the lam program we create a graph where nodes are function names and, for every invocation of `g` in the body of `f`, there is an edge from `f` to `g`. Then a standard depth first search associates to every node its recursive histories (the paths starting and ending at that node, if any). The lam program is linear recursive if every node has at most one associated recursive history.

Step 2: *computation of the orders*. Given the recursive history α associated to a function `f`, we compute the corresponding mutation by running $\alpha \models \varepsilon$ (see Section 4). A straightforward parse of the mutation returns the set of cycles and sinks and, therefore, gives the order $\mathfrak{o}_\mathtt{f}$.

Step 3: *evaluation process*. The main lam is unfolded till the the saturated state. That is, every function invocation $\mathtt{f}(\widetilde{x})$ in the main lam is evaluated up-to twice the order of the corresponding mutation. The function invocation of `f` in the saturated state is erased and the process is repeated on every other function invocation (which, therefore, does not belong to the recursive history of `f`), till no function invocation is present in the state. At this stage we use the lam axioms that yield a term $\mathtt{T}_1 + \cdots + \mathtt{T}_n$.

Step 4: *detection of circularities*. Every $\mathtt{T}_i$ in $\mathtt{T}_1 + \cdots + \mathtt{T}_n$ may be represented as a graph where nodes are names and edges correspond to dependency pairs. To

detect whether $\mathtt{T}_i$ contains a circular dependency, we run Tarjan algorithm [31] for connected components of graphs and we stop the algorithm when a circularity is found.

Every preliminary notion is in place for stating our main result; we also make few remarks about the correctness of the algorithm and its computational cost.

Theorem 3. *The problem of the circularity-freedom of a lam program is decidable when the program is linear recursive.*

The algorithm consists of the four steps described above. The critical step, as far as correctness is concerned, is the third one, which follows by Theorem 2 and by the diamond property in Proposition 3 (whatever other computation may be completed in such a way the final state is equal up-to a bijection to a saturated state).

As regards the computational complexity Steps 1 and 2 are linear with respect to the size of the lam program and Step 4 is linear with respect to the size of the term $\mathtt{T}_1 + \cdots + \mathtt{T}_n$. Step 3 evaluates the program till the saturated state. Let

$\mathfrak{o}_{max}$ be the largest order of a function;
m_{max} be the maximal number of function invocations in a body, apart the one in the recursive history.

Without loss of generality, we assume that recursive histories have length 1 and that the main lam consists of m_{max} invocations of the same function. Then an upper bound to the length of the evaluation till the saturated state is

$$(2 \times \mathfrak{o}_{max} \times m_{max}) + (2 \times \mathfrak{o}_{max} \times m_{max})^2 + \cdots + (2 \times \mathfrak{o}_{max} \times m_{max})^\ell$$

Let k_{max} be the maximal number of dependency pairs in a body. Then the size of the saturated state is $O(k_{max} \times (\mathfrak{o}_{max} \times m_{max})^\ell)$, which is also the computational complexity of our algorithm.

6 Assessments

The algorithm defined in Section 5 has been prototyped [15]. As anticipated in Section 1, our analysis has been applied to a concurrent object-oriented language called `ABS` [17], which is a Java-like language with futures and an asynchronous concurrency model (`ASP` [6] is another language in the same family).

The prototype is part of a bigger framework for the deadlock analysis of `ABS` programs called `DF4ABS` (Deadlock Framework for `ABS`). It is a modular framework which includes two different approaches for analysing lams: `DF4ABS/model-check` (which is the one described in the current paper) and `DF4ABS/fixpoint` (which is the one described in [13, 14]).

The technique underpinning the `DF4ABS/fixpoint` tool derives the dependency graph(s) of lam programs by means of a standard fixpoint analysis. To

circumvent the issue of the infinite generation of new names, the fixpoint is computed on models with a limited capacity of name creation. This introduces over-approximations that in turn display false positives (for example, `DF4ABS/fixpoint` returns a false positive for the lam of `factorial`). In the present work, this limitation of finite models is overcome (for linear recursive programs) by recognizing patterns of recursive behaviors, so that it is possible to reduce the analysis to a finite portion of computation without losing precision in the detection of deadlocks.

The derivation of lams from `ABS` programs is defined by an *inference system* [13, 14]. The inference system extracts behavioral types from `ABS` programs and feeds them to the analyzer. These types display the resource dependencies and the method invocations while discarding irrelevant (for the deadlock analysis) details. There are two relevant differences between inferred types and lams: (i) methods' arguments have a record structure and (ii) behavioral types have the union operator (for modeling the if-then-else statement). To bridge this gap and have some initial assessments, we perform a basic *automatic* transformation of types into lams.

We tested our prototype on a number of medium-size programs written for benchmarking purposes by `ABS` programmers and on an industrial case study based on the Fredhopper Access Server (FAS) developed by SDL Fredhopper [9]. This Access Server provides search and merchandising IT services to e-Commerce companies. The (leftmost three columns of the) following table reports the experiments: for every program we display the number of lines, whether the analysis has reported a deadlock (`D`) or not (✓), the time in seconds required for the analysis. Concerning time, we only report the time of the analysis (and not the one taken by the inference) when they run on a QuadCore 2.4GHz and Gentoo (Kernel 3.4.9):

program	lines	`DF4ABS/model-check` result	time	`DF4ABS/fixpoint` result	time	`DECO` result	time
`PingPong`	61	✓	0.311	✓	0.046	✓	1.30
`MultiPingPong`	88	`D`	0.209	`D`	0.109	`D`	1.43
`BoundedBuffer`	103	✓	0.126	✓	0.353	✓	1.26
`PeerToPeer`	185	✓	0.320	✓	6.070	✓	1.63
`FAS Module`	2645	✓	31.88	✓	39.78	✓	4.38

The rightmost column of the above table reports the results of another tool that have also been developed for the deadlock analysis of `ABS` programs: `DECO` [11]. The technique in [11] integrates a point-to analysis with an analysis returning (an over-approximation of) program points that may be running in parallel. As for other model checking techniques, the authors use a finite amount of (abstract) object names to ensure termination of programs with object creations underneath iteration or recursion. For example, `DECO` (as well as `DF4ABS/fixpoint`) signals a deadlock in programs containing methods whose

lam is [1] $\mathtt{m}(x,y) = (y,x)\&\mathtt{m}(z,x)$ that our technique correctly recognizes as deadlock-free.

As highlighted by the above table, the three tools return the same results as regards deadlock analysis, but are different as regards performance. In particular `DF4ABS/model-check` and `DF4ABS/fixpoint` are comparable on small/mid-size programs, `DECO` appears less performant (except for `PeerToPeer`, where `DF4ABS/fixpoint` is quite slow because of the number of dependencies produced by the fixpoint algorithm). On the `FAS module`, `DF4ABS/model-check` and `DF4ABS/fixpoint` are again comparable – their computational complexity is exponential – `DECO` is more performant because its worst case complexity is cubic in the dimension of the input. As we discuss above, this gain in performance is payed by `DECO` in a loss of precision.

Our final remark is about the proportion between linear recursive functions and nonlinear ones in programs. This is hard to assess and our answer is perhaps not enough adequate. We have parsed the three case-studies developed in the European project HATS [9]. The case studies are the FAS module, a Trading System (TS) modeling a supermarket handling sales, and a Virtual Office of the Future (VOF) where office workers are enabled to perform their office tasks seamlessly independent of their current location. FAS has 2645 code-lines, TS has 1238 code-lines, and VOF has 429 code-lines. In none of them we found a nonlinear recursion, TS and VOF have respectively 2 and 3 linear recursions (there are recursions in functions on data-type values that have nothing to do with locks and control). This substantiates the usefulness of our technique in these programs; the analysis of a wider range of programs is matter of future work.

7 Related works

The solutions in the literature for deadlock detection in infinite state programs either give imprecise answers or do not scale when, for instance, programs also admit dynamic resource creation. Two basic techniques are used: type-checking and model-checking.

Type-based deadlock analysis has been extensively studied both for process calculi [19, 30, 32] and for object-oriented programs [3, 10, 1]. In Section 1 we have thoroughly discussed our position with respect to Kobayashi's works; therefore we omit here any additional comment. In the other contributions about deadlock analysis, a type system computes a partial order of the deadlocks in a program and a subject reduction theorem proves that tasks follow this order. On the contrary, our technique does not compute any ordering of deadlocks, thus being more flexible: a computation may acquire two deadlocks in different order at different stages, thus being correct in our case, but incorrect with the other techniques. A further difference with the above works is that we use behavioral types, which are terms in some simple process algebras [21]. The use of simple

[1] The code of a corresponding `ABS` program is available at the `DF4ABS` tool website [15], *c.f.* `UglyChain.abs`.

process algebras to guarantee the correctness (= deadlock freedom) of interacting parties is not new. This is the case of the exchange patterns in SSDL [27], which are based on CSP [4] and pi-calculus [23], of session types [12], or of the terms in [26] and [7], which use CCS [22]. In these proposals, the deadlock freedom follows by checking either a dual-type relation or a behavioral equivalence, which amounts to model checking deadlock freedom on the types.

As regards model checking techniques, in [5] circular dependencies among processes are detected as erroneous configurations, but dynamic creation of names is not treated. An alternative model checking technique is proposed in [2] for multi-threaded asynchronous communication languages with futures (as `ABS`). This technique is based on vector systems and addresses infinite-state programs that admit thread creation but not dynamic resource creation.

The problem of verifying deadlocks in infinite state models has been studied in other contributions. For example, [28] compare a number of unfolding algorithms for Petri Nets with techniques for safely cutting potentially infinite unfoldings. Also in this work, dynamic resource creation is not addressed. The techniques conceived for dealing with dynamic name creations are the so-called *nominal techniques*, such as nominal automata [29, 25] that recognize languages over infinite alphabets and HD-automata [24], where names are explicit part of the operational model. In contrast to our approach, the models underlying these techniques are finite state. Additionally, the dependency relation between names, which is crucial for deadlock detection, is not studied.

8 Conclusions and future work

We have defined an algorithm for the detection of deadlocks in infinite state programs, which is a decision procedure for linear recursive programs that feature dynamic resource creation. This algorithm has been prototyped [15] and currently experimented on programs written in an object-oriented language with futures [17]. The current prototype deals with nonlinear recursive programs by using a source-to-source transformation into linear ones. This transformation *may introduce fake dependencies* (which in turn may produce false positives in terms of circularities). To briefly illustrate the technique, consider the program

$$\big(\, \mathtt{h}(t) = (t,x)\&(t,y)\&\mathtt{h}(x)\&\mathtt{h}(y)\ ,\ \mathtt{h}(u)\ \big),$$

Our transformation returns the linear recursive one:

$$\begin{array}{l}\big(\, \mathtt{h}^{aux}(t,t') = (t,x)\&(t,x')\&(t',x)\&(t',x')\&\mathtt{h}^{aux}(x,x')\ ,\\ \ \ \mathtt{h}(u) = \mathtt{h}^{aux}(u,u)\ ,\ \ \mathtt{h}(u)\ \big)\ .\end{array}$$

To highlight the fake dependencies added by $\mathtt{h}^{aux}$, we notice that, after two unfoldings, $\mathtt{h}^{aux}(u,u)$ gives

$$(u,v)\&(u,w)\&(v,v')\&(v,w')\&(w,v')\&(w,w')\&\mathtt{h}^{aux}(v',w')$$

while $\mathtt{h}(u)$ has a corresponding state (obtained after four steps)

$$\begin{array}{l}(u,v)\&(u,w)\&(v,v')\&(v,v'')\&(w,w')\&(w,w'')\\ \&\mathtt{h}(v')\&\mathtt{h}(v'')\&\mathtt{h}(w')\&\mathtt{h}(w'')\ ,\end{array}$$

and this state has no dependency between names created by different invocations. It is worth to remark that these additional dependencies cannot be completely eliminated because of a cardinality argument. The evaluation of a function invocation $\mathtt{f}(\widetilde{u})$ in a linear recursive program may produce at most one invocation of $\mathtt{f}$, while an invocation of $\mathtt{f}(\widetilde{u})$ in a nonlinear recursive program may produce two or more. In turn, these invocations of $\mathtt{f}$ may create names (which are exponentially many in a nonlinear program). When this happens, the creations of different invocations must be *contracted* to names created by one invocation and explicit dependencies must be *added* to account for dependencies of each invocation. [Our source-to-source transformation is sound: if the transformed linear recursive program is circularity-free then the original nonlinear one is also circularity-free. So, for example, since our analysis lets us determine that the saturated state of $\mathtt{h}^{aux}$ is circularity-free, then we are able to infer the same property for $\mathtt{h}$.] We are exploring possible generalizations of our theory in Section 4 to nonlinear recursive programs that replace the notion of mutation with that of *group of mutations*. This research direction is currently at an early stage.

Another obvious research direction is to apply our technique to deadlocks due to process synchronizations, as those in process calculi [23, 19]. In this case, one may take advantage of Kobayashi's inference for deriving inter-channel dependency informations and manage recursive behaviors by using our algorithm (instead of the one in [20]).

There are several ways to develop the ideas here, both in terms of the language features of lams and the analyses addressed. As regards the lam language, [13] already contains an extension of lams with union types to deal with assignments, data structures, and conditionals. However, the extension of the theory of mutations and flashbacks to deal with these features is not trivial and may yield a weakening of Theorem 2. Concerning the analyses, the theory of mutations and flashbacks may be applied for verifying properties different than deadlocks, such as state reachability or livelocks, possibly using different lam languages and different notions of saturated state. Investigating the range of applications of our theory and studying the related models (corresponding to lams) are two issues that we intend to pursue.

References

[1] M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for Java. *TOPLAS*, 28, 2006.

[2] A. Bouajjani and M. Emmi. Analysis of recursively parallel programs. In *POPL'12*, pages 203–214. ACM, 2012.

[3] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe program.: preventing data races and deadlocks. In *OOPSLA*, pages 211–230. ACM, 2002.

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

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

[6] D. Caromel, L. Henrio, and B. P. Serpette. Asynchronous and deterministic objects. In *POPL*, pages 123–134. ACM, 2004.

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

[8] L. Comtet. *Advanced Combinatorics: The Art of Finite and Infinite Expansions*. Dordrecht, Netherlands, 1974.

[9] Requirement elicitation, August 2009. Deliverable 5.1 of project FP7-231620 (HATS), available at `http://www.hats-project.eu/sites/default/files/Deliverable51_rev2.pdf`.

[10] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *PLDI*, pages 338–349. ACM, 2003.

[11] A. Flores-Montoya, E. Albert, and S. Genaim. May-happen-in-parallel based deadlock analysis for concurrent objects. In *FORTE/FMOODS 2013*, volume 7892 of *LNCS*, pages 273–288. Springer, 2013.

[12] S. J. Gay and R. Nagarajan. Types and typechecking for communicating quantum processes. *MSCS*, 16(3):375–406, 2006.

[13] E. Giachino, C. A. Grazia, C. Laneve, M. Lienhardt, and P. Y. H. Wong. Deadlock analysis of concurrent objects: Theory and practice. In *iFM'13*, volume 7940 of *LNCS*, pages 394–411. Springer-Verlag, 2013.

[14] E. Giachino, C. Laneve, and M. Lienhardt. A Framework for Deadlock Detection in ABS. 2013. Submitted. Available at `www.cs.unibo.it/~laneve`.

[15] E. Giachino, C. Laneve, and M. Lienhardt. Deadlock Framework for ABS (DF4ABS) - online interface, 2013. at `www.cs.unibo.it/~giachino/siteDat/`.

[16] A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. *Theor. Comput. Sci.*, 311(1-3):121–163, 2004.

[17] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In *FMCO*, volume 6957 of *LNCS*, pages 142–164. Springer-Verlag, 2011.

[18] N. Kobayashi. A partially deadlock-free typed process calculus. *TOPLAS*, 20(2):436–482, 1998.

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

[20] N. Kobayashi. TyPiCal. at `kb.ecei.tohoku.ac.jp/~koba/typical/`, 2007.

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

[22] R. Milner. *A Calculus of Communicating Systems*. Springer, 1982.

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

[24] U. Montanari and M. Pistore. An introduction to history dependent automata. *Electr. Notes Theor. Comput. Sci.*, 10:170–188, 1997.

[25] F. Neven, T. Schwentick, and V. Vianu. Towards regular languages over infinite alphabets. In *MFCS*, volume 2136 of *LNCS*, pages 560–572. Springer, 2001.

[26] H. R. Nielson and F. Nielson. Higher-order concurrent programs with finite communication topology. In *POPL*, pages 84–97. ACM, 1994.

[27] S. Parastatidis and J. Webber. *MEP SSDL Protocol Framework*, Apr. 2005. `http://ssdl.org`.

[28] C. Schrter and J. Esparza. Reachability analysis using net unfoldings. In *CS&P'2000*, pages 255–270, 2000.

[29] L. Segoufin. Automata and logics for words and trees over an infinite alphabet. In *CSL*, volume 4207 of *LNCS*, pages 41–57. Springer, 2006.

[30] K. 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.

[31] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[32] V. T. Vasconcelos, F. Martins, and T. Cogumbreiro. Type inference for deadlock detection in a multithreaded polymorphic typed assembly language. In *PLACES*, volume 17 of *EPTCS*, pages 95–109, 2009.

A Java code of the factorial function

There are several JAVA programs implementing `factorial` in Section 1. However our goal is to convey some intuition about the differences between TYPICAL and our technique, rather than to analyze the possible options. One option is the code

```
synchronized void fact(final int n,final int m,final Maths x)
                                    throws InterruptedException {
        if (n==0) x.retresult(m) ;
        else {
            final Maths y = new Maths() ;
            Thread t = new Thread(new Runnable() {
                public void run() {
                    try { y.fact(n-1,n*m,x) ;
                    } catch (InterruptedException e) { }
                }   }) ;
            t.start();
            t.join() ;
        }
    }
```

Since `factorial` is `synchronized`, the corresponding thread acquires the lock of its object – let it be *this* – before execution and releases the lock upon termination. We notice that `factorial`, in case `n>0`, delegates the computation of factorial to a separate thread on a new object of `Maths`, called y. This means that no other synchronized thread on *this* may be scheduled until the recursive invocation on y terminates. Said formally, the runtime Java configuration contains an object dependency $(this, y)$. Repeating this argument for the recursive invocation, we get configurations with chains of dependencies $(this, y), (y, z), \cdots$, which are finite by the well-foundedness of naturals.

B Proof of Theorem 2.

This section develops the technical details for proving Theorem 2.

Definition 9. *A history α is*

`f`-*yielding*
if $\alpha = \alpha_1^{h_1}\beta_1 \cdots \alpha_n^{h_n}\beta_n$ such that, for every i, α_i is a recursive history, $\beta_i \preceq \alpha_i$, and $\alpha = \alpha'\mathtt{f}_i$ implies the program has the definition $\mathtt{f}_i(\widetilde{x}_i) = \mathfrak{L}[\mathtt{f}(\widetilde{u})]$, for some $\widetilde{u}$. The kernel *of α, denoted $[\alpha]$, is $\alpha_1^{h'_1}\beta_1 \cdots \alpha_n^{h'_n}\beta_n$, where $h'_i = \mathtt{min}(h_i, 1)$.*

By definition, if α is `f`-saturating then it is also `f`-yielding. In this case, the kernel $[\alpha]$ has a suffix that is `f`-complete. In the `flh`-program, $\mathfrak{o}_{\mathtt{f}} = 4$, $\mathfrak{o}_{\mathtt{l}} = 3$, and $\mathfrak{o}_{\mathtt{h}} = 1$, and the recursive histories of `f`, `l` and `h` are equal to `fl`, to `lf` and to `h`, respectively. Then $\alpha = (\mathtt{fl})^4$ is the `f`-complete history and $\alpha' = \mathtt{h}^2\mathtt{f}$ is `l`-yielding, with $[\alpha'] = \mathtt{hf}$.

We notice that every history of an informative lam (obtained by evaluating $\langle \mathbb{l}_{\mathtt{L}}, \varnothing, addh(\varepsilon, \mathtt{L})\rangle$) is a yielding sequence. We also notice that, for every $\mathtt{f}$, ε is $\mathtt{f}$-yielding. In fact, ε is the history of every function invocation in the initial lam, which may concern every function name of the program. As regards the kernel, in Lemma 1, we demonstrate that, if $\alpha = \alpha_1^{h_1}\beta_1 \cdots \alpha_n^{h_n}\beta_n$ is a $\mathtt{f}$-yielding history such that every $h_i \geqslant 2$, then every term ${}^{\alpha}\mathtt{f}(\widetilde{u})$ may be mapped by a flashback ρ to a term ${}^{[\alpha]}\mathtt{f}(\rho(\widetilde{u}))$; similarly for dependencies. This is the basic property that allows us to map circularities to past circularities (see Theorem 2).

Next we introduce an ordering relation over renamings, (in particular, flashbacks) and the operation of renaming composition. The definitions are almost standard:

- $\rho \preceq^{\mathtt{fb}} \rho'$ if, for every $x \in dom(\rho)$, $\rho(x) = \rho'(x)$.
- $\rho \circ \rho'$ be defined as follows:

$$(\rho \circ \rho')(x) \stackrel{def}{=} \begin{cases} \rho'(x) & \text{if } \rho'(x) \notin dom(\rho) \\ \rho(\rho'(x)) & \text{otherwise} \end{cases}$$

We notice that, if both

1. ρ and ρ' are flashbacks and
2. for every $x \in dom(\rho)$, $\rho'(x) = x$

then $\rho \preceq^{\mathtt{fb}} \rho \circ \rho'$ holds. In the following, lams $\flat(\mathtt{L})$ and $\flat(\mathbb{L})$, being $+$ of terms that are dependencies composed with $\&$, will be written $\mathtt{T}_1 + \cdots + \mathtt{T}_m$ and ${}^{\flat}\mathtt{T}_1 + \cdots + {}^{\flat}\mathtt{T}_m$, for some m, respectively, where $\mathtt{T}_i$ and ${}^{\flat}\mathtt{T}_i$ contain dependencies (x, y) and ${}^{\alpha}(x, y)$. Let also $\rho(\&_{i\in I}(x_i, y_i)) = \&_{i\in I}(\rho(x_i), \rho(y_i))$.

With an abuse of notation, we will use the set operation "$\in$" for $\mathtt{L}$ and ${}^{\flat}\mathtt{L}$. For instance, we will write $\mathtt{L}' \in \mathtt{L}$ when there is $\mathfrak{L}[\ ]$ such that $\mathtt{L} = \mathfrak{L}[\mathtt{L}']$. Similarly, we will write $\mathtt{T} \in \mathtt{T}_1 + \cdots + \mathtt{T}_n$ when there is $\mathtt{T}_i$ such that $\mathtt{T} \in \mathtt{T}_i$.

A consequence of the axiom $\mathtt{T}\&(\mathtt{L}' + \mathtt{L}'') = \mathtt{T}\&\mathtt{L}' + \mathtt{T}\&\mathtt{L}''$ is the following property of the informative operational semantics.

Proposition 6. *Let* $\langle \mathbb{V}_1, {}^{\flat}\mathbb{F}, {}^{\flat}\mathfrak{L}_0[{}^{\alpha}\mathtt{f}_1(\widetilde{u_1})]\rangle$ *be a state of an informative operational semantics. For every* $1 \leqslant i \leqslant n$, *let* $\mathtt{f}_i(\widetilde{u_i}) = \mathtt{L}'_i$ *and* $addh(\alpha\mathtt{f}_0 \cdots \mathtt{f}_i, \mathtt{L}'_i)$ *be* ${}^{\flat}\mathfrak{L}_i[{}^{\alpha\mathtt{f}_1\cdots\mathtt{f}_i}\mathtt{f}_{i+1}(\widetilde{u_{i+1}})]$. *Finally, let*

$$\flat({}^{\flat}\mathfrak{L}_1[\cdots {}^{\flat}\mathfrak{L}_n[{}^{\alpha\mathtt{f}_1\cdots\mathtt{f}_n}\mathtt{f}_{n+1}(\widetilde{u_{n+1}})]\cdots]) = {}^{\flat}\mathtt{T}_1 + \cdots + {}^{\flat}\mathtt{T}_r$$
$$\flat({}^{\flat}\mathfrak{L}_n[{}^{\alpha\mathtt{f}_1\cdots\mathtt{f}_n}\mathtt{f}_{n+1}(\widetilde{u_{n+1}})]) = {}^{\flat}\mathtt{T}'_1 + \cdots + {}^{\flat}\mathtt{T}'_{r'}\ .$$

If ${}^{\alpha\mathtt{f}_1\cdots\mathtt{f}_n}(x, y)\&addh(\alpha', \mathtt{T}) \in {}^{\flat}\mathtt{T}_1 + \cdots + {}^{\flat}\mathtt{T}_r$ *then, for every* $1 \leqslant j \leqslant r'$, ${}^{\flat}\mathtt{T}'_j\&addh(\alpha', \mathtt{T}) \in {}^{\flat}\mathtt{T}_1 + \cdots + {}^{\flat}\mathtt{T}_r$.

The next lemma allows us to map, through a flashback, terms in a saturated state to terms that have been produced in the past. The correspondence is defined by means of the (regular) structure of histories.

Lemma 1. *Let* $\langle \mathbb{l}_{\mathtt{L}}, \varnothing, addh(\varepsilon, \mathtt{L})\rangle \longrightarrow^* \langle \mathbb{V}, {}^{\flat}\mathbb{F}, \mathbb{L}\rangle$ *and* $\langle \mathbb{V}, {}^{\flat}\mathbb{F}, \mathbb{L}\rangle$ *be saturated and* $\flat(\mathbb{L}) = {}^{\flat}\mathtt{T}_1 + \cdots + {}^{\flat}\mathtt{T}_m$. *Then*

1. *if* ${}^{\beta\alpha^{n+2}\beta'}\mathtt{f}(\widetilde{u}) \in \mathbb{L}$, *where* $\beta\alpha^{n+2}\beta'$ *is* $\mathtt{f}$*-yielding, then there are* $n+1$ $\mathbb{V}$*-flashbacks* $\rho^{(2)}_{\beta,\alpha,\beta'}, \cdots, \rho^{(n+2)}_{\beta,\alpha,\beta'}$ *such that:*
   (a) ${}^{\beta\alpha^{n+1}\beta'}\mathtt{f}(\rho^{(n+2)}_{\beta,\alpha,\beta'}(\widetilde{u})) \in {}^{\flat}\mathbb{F}$;
   (b) $\&_{j\in J}\, addh(\beta\alpha^{k+1}\beta_j, \mathtt{T}'_j) \in {}^{\flat}\mathtt{T}_1 + \cdots + {}^{\flat}\mathtt{T}_m$ *where, for every* j, $\beta_j \preceq \alpha$, *implies* $\&_{j\in J}\, addh(\beta\alpha^{k}\beta_j, \rho^{(k+1)}_{\beta,\alpha,\beta'}(\mathtt{T}'_j)) \in {}^{\flat}\mathtt{T}_1 + \cdots + {}^{\flat}\mathtt{T}_m$;
   (c) ${}^{\beta\alpha^{k+1}\beta'}\mathtt{f}(\widetilde{u}) \in {}^{\flat}\mathbb{F}$ *implies* ${}^{\beta\alpha^{k}\beta'}\mathtt{f}(\rho^{(k+1)}_{\beta,\alpha,\beta'}(\widetilde{u})) \in {}^{\flat}\mathbb{F}$.
2. *if* $\alpha_1, \cdots, \alpha_k$ *are* $\mathtt{f}_1$*-yielding,* $\cdots$, $\mathtt{f}_k$*-yielding, respectively, then there are flashbacks* $\rho_{\alpha_1}, \cdots, \rho_{\alpha_k}$ *such that*
   (a) *if* ${}^{\alpha_1}\mathtt{f}_1(\widetilde{u}) \in \mathbb{L}$ *or* ${}^{\alpha_1}\mathtt{f}_1(\widetilde{u}) \in {}^{\flat}\mathbb{F}$ *then* ${}^{[\alpha_1]}\mathtt{f}(\rho_{\alpha_1}(\widetilde{u})) \in {}^{\flat}\mathbb{F}$;
   (b) *if* $\&_{1\leqslant j\leqslant k}\, addh(\alpha_j, \mathtt{T}_j) \in {}^{\flat}\mathtt{T}_1 + \cdots + {}^{\flat}\mathtt{T}_m$ *then* $\&_{1\leqslant j\leqslant k}\, addh([\alpha_j], \rho_{\alpha_j}(\mathtt{T})) \in {}^{\flat}\mathtt{T}_1 + \cdots + {}^{\flat}\mathtt{T}_m$;
   (c) *if* $\alpha_1 \preceq \alpha_2$ *then* $\rho_{\alpha_1} \preceq^{\mathtt{fb}} \rho_{\alpha_2}$.
   (In particular, if $\alpha_1 = \beta\alpha^{n+2}\beta'$, *with* $\beta' \preceq \alpha$, *and* $\alpha_2 = \beta\alpha^{n+3}$ *then* $\rho_{\alpha_1} \preceq^{\mathtt{fb}} \rho_{\alpha_2}$*).*

Proof. (Sketch) As regards item 1, let $\alpha = \beta'\beta''$ and let $\beta''\beta' = \mathtt{f}\mathtt{f}_1\cdots\mathtt{f}_m$ (therefore the length of α is $m+1$). The evaluation $\langle \mathbb{I}_{\mathtt{L}},\, \varnothing,\, addh(\varepsilon, \mathtt{L})\rangle \longrightarrow^* \langle \mathbb{V},\, {}^{\flat}\mathbb{F},\, \mathbb{L}\rangle$ may be decomposed as follows

$$\begin{array}{l}\langle \mathbb{I}_{\mathtt{L}},\, \varnothing,\, addh(\varepsilon, \mathtt{L})\rangle \longrightarrow^* \langle \mathbb{V}',\, {}^{\flat}\mathbb{F}',\, {}^{\flat}\mathfrak{L}[{}^{\beta\alpha^{n+1}\beta'}\mathtt{f}(\widetilde{u'})]\rangle \\ \qquad\qquad\qquad\qquad \longrightarrow^* \langle \mathbb{V},\, {}^{\flat}\mathbb{F},\, \mathbb{L}\rangle\end{array}$$

By definition of the operational semantics there is the *alternative* evaluation

$$\begin{array}{l}\langle \mathbb{V}',\, {}^{\flat}\mathbb{F}',\, {}^{\flat}\mathfrak{L}[{}^{\beta\alpha^{n+1}\beta'}\mathtt{f}(\widetilde{u'})]\rangle \\ \longrightarrow \langle \mathbb{V}'',\, {}^{\flat}\mathbb{F}'',\, {}^{\flat}\mathfrak{L}[{}^{\flat}\mathfrak{L}'[{}^{\beta\alpha^{n+1}\beta'\mathtt{f}}\mathtt{f}_1(\widetilde{u_1})]]\rangle \\ \longrightarrow^* \langle \mathbb{V}''',\, {}^{\flat}\mathbb{F}''',\, {}^{\flat}\mathfrak{L}[{}^{\flat}\mathfrak{L}'[{}^{\flat}\mathfrak{L}_1[\cdots {}^{\flat}\mathfrak{L}_m[{}^{\beta\alpha^{n+1}\beta'\mathtt{f}\mathtt{f}_1\cdots\mathtt{f}_m}\mathtt{f}(\widetilde{u''})]\cdots]]]\rangle\end{array}$$

[notice that $\beta\alpha^{n+1}\beta'\mathtt{f}\mathtt{f}_1\cdots\mathtt{f}_m = \beta\alpha^{n+2}\beta'$]. Property (1.a) is an immediate consequence of Proposition 5; let $\varrho^{(n+2)}_{\beta,\alpha,\beta'}$ be the flashback for the last state. The property (1.b), when $k=n$, is also an immediate consequence of Propositions 5 and of 6. In the general case, we need to iterate the arguments on shorter histories and the arguments are similar for (1.c). In order to conclude the proof of item 1, we need an additional argument. By Proposition 3, there exists an evaluation

$$\begin{array}{l}\langle \mathbb{V}''',\, {}^{\flat}\mathbb{F}''',\, {}^{\flat}\mathfrak{L}[{}^{\flat}\mathfrak{L}'[{}^{\flat}\mathfrak{L}_1[\cdots {}^{\flat}\mathfrak{L}_m[{}^{\beta\alpha^{n+1}\beta'\mathtt{f}\mathtt{f}_1\cdots\mathtt{f}_m}\mathtt{f}(\widetilde{u''})]\cdots]]]\rangle \\ \qquad \longrightarrow^* \langle \mathbb{V}^\sharp,\, {}^{\flat}\mathbb{F}^\sharp,\, \mathbb{L}^\sharp\rangle\end{array}$$

such that $\langle \mathbb{V}^\sharp,\, {}^{\flat}\mathbb{F}^\sharp,\, \mathbb{L}^\sharp\rangle$ and $\langle \mathbb{V},\, {}^{\flat}\mathbb{F},\, \mathbb{L}\rangle$ are identified by a bijective renaming, let it be $\jmath$. We define the $\rho^{(n+2)}_{\beta,\alpha,\beta'}$ corresponding to the evaluation $\langle \mathbb{I}_{\mathtt{L}},\, \varnothing,\, addh(\varepsilon, \mathtt{L})\rangle \longrightarrow^* \langle \mathbb{V},\, {}^{\flat}\mathbb{F},\, \mathbb{L}\rangle$ as $\rho^{(n+2)}_{\beta,\alpha,\beta'} \stackrel{def}{=} \jmath \circ \varrho^{(n+2)}_{\beta,\alpha,\beta'} \circ \jmath^{-1}$. Similarly for the other $\rho^{(k+1)}_{\beta,\alpha,\beta'}$. The properties of item 1 for $\langle \mathbb{V},\, {}^{\flat}\mathbb{F},\, \mathbb{L}\rangle$ follow by the corresponding ones for

$$\langle \mathbb{V}''',\, {}^{\flat}\mathbb{F}''',\, {}^{\flat}\mathfrak{L}[{}^{\flat}\mathfrak{L}'[{}^{\flat}\mathfrak{L}_1[\cdots {}^{\flat}\mathfrak{L}_m[{}^{\beta\alpha^{n+1}\beta'\mathtt{f}\mathtt{f}_1\cdots\mathtt{f}_m}\mathtt{f}(\widetilde{u''})]\cdots]]]\rangle\ .$$

We prove item 2. We observe that a term with history $\beta_0(\alpha'_1)^{h_1}\ \beta_1 \cdots \beta_{n-1}\ (\alpha'_n)^{h_n}\ \beta_n$ in ${}^{\mathfrak{h}}\mathbb{F}$ or in $\mathbb{L}$ may have no corresponding term (by a flashback) with history $\beta_0(\alpha'_1)^{h_1-1}\beta_1\ (\alpha'_2)^{h_2} \cdots \beta_{n-1}\ (\alpha'_n)^{h_n}\beta_n$. This is because the evaluation to the saturated state may have not expanded some invocations. It is however true that terms with histories $[\beta_0(\alpha'_1)^{h_1}\beta_1 \cdots \beta_{n-1}(\alpha'_n)^{h_n}\beta_n]$ (kernels) are either in ${}^{\mathfrak{h}}\mathbb{F}$ or in $\mathbb{L}$ and the item 2 is demonstrated by proving that a flashback to terms with histories that are kernels does exist.

Let $\alpha_1 = \beta_0(\alpha'_1)^{h_1}\beta_1 \cdots \beta_{n-1}(\alpha'_n)^{h_n}\beta_n$ be a $\mathtt{f}$-yielding sequence. We proceed by induction on n. When $n = 1$ there are two cases: $h_1 \leqslant 1$ and $h_1 \geqslant 2$. In the first case there is nothing to prove because $[\alpha] = \alpha$. When $h_1 \geqslant 2$, since α fits with the hypotheses of Item 1, there exist $\rho^{(2)}_{\beta_0,\alpha'_1,\beta_1}, \cdots, \rho^{(h_1)}_{\beta_0,\alpha'_1,\beta_1}$. Let $\delta^{(2)}_{\beta_0,\alpha'_1,\beta_1} = \rho^{(2)}_{\beta_0,\alpha'_1,\beta_1}$ and $\delta^{(i+1)}_{\beta_0,\alpha'_1,\beta_1} = \rho^{(i+1)}_{\beta_0,\alpha'_1,\beta_1}[x \mapsto x \mid x \in dom(\delta^{(i)}_{\beta_0,\alpha'_1,\beta_1})]$. We also let $\rho_{\alpha_1} = \delta^{(2)}_{\beta_0,\alpha'_1,\beta_1} \circ \cdots \circ \delta^{(h_1)}_{\beta_0,\alpha'_1,\beta_1}$ and we observe that, by definition of renaming composition, if $\alpha_1 \preceq \alpha_2$ then $\rho_{\alpha_1} \preceq^{\mathtt{fb}} \rho_{\alpha_2}$. In this case, the items 2.a and 2.b follow by item 1, Proposition 6 and the diamond property of Proposition 3.

We assume the statement holds for a generic n and we prove the case $n+1$. Let $\alpha_1 = \beta\beta_n(\alpha'_{n+1})^{h_{n+1}}\beta_{n+1}$ and $h_{n+1} > 0$ (because $[\beta_n(\alpha'_{n+1})^1\beta_{n+1}] = \beta_n\alpha'_{n+1}\beta_{n+1}$). We consider the map

$$\rho_{\alpha_1} \stackrel{def}{=} \rho_\beta \circ \delta^{(2)}_{\beta_n,\alpha'_{n+1},\beta_{n+1}} \circ \cdots \circ \delta^{(h_{n+1})}_{\beta_n,\alpha'_{n+1},\beta_{n+1}}$$

where $\delta^{(i)}_{\beta_n,\alpha'_{n+1},\beta_{n+1}}$, $2 \leqslant i \leqslant h_{n+1}$ are defined as above. As before, the items 2.a and 2.b follow by item 1 for $\delta^{(2)}_{\beta_n,\alpha'_{n+1},\beta_{n+1}} \circ \cdots \circ \delta^{(h_{n+1})}_{\beta_n,\alpha'_{n+1},\beta_{n+1}}$ and by Proposition 6 and the diamond property of Proposition 3. Then we apply the inductive hypothesis for ρ_β. The property (2.c) $\alpha_1 \preceq \alpha_2$ implies $\rho_{\alpha_1} \preceq^{\mathtt{fb}} \rho_{\alpha_2}$ is an immediate consequence of the definition.

Every preliminary statement is in place for our key theorem that details the mapping of circularities created by transitions of saturated states to past circularities. For readability sake, we restate the theorem.

Theorem 2. *Let* $\langle \mathbb{l}_{\mathtt{L}},\ \varnothing,\ addh(\varepsilon, \mathtt{L})\rangle \longrightarrow^* \langle \mathbb{V},\ {}^{\mathfrak{h}}\mathbb{F},\ \mathbb{L}\rangle$ *and* $\langle \mathbb{V},\ {}^{\mathfrak{h}}\mathbb{F},\ \mathbb{L}\rangle$ *be a saturated state. If* $\langle \mathbb{V},\ {}^{\mathfrak{h}}\mathbb{F},\ \mathbb{L}\rangle \longrightarrow \langle \mathbb{V}',\ {}^{\mathfrak{h}}\mathbb{F}',\ \mathbb{L}'\rangle$ *then*

1. $\langle \mathbb{V}',\ {}^{\mathfrak{h}}\mathbb{F}',\ \mathbb{L}'\rangle$ *is saturated;*
2. *if* $\mathbb{L}'$ *has a circularity then* $\mathbb{L}$ *has already a circularity.*

Proof. The item 1. is an immediate consequence of Proposition 5. We prove 2. Let

- $\mathbb{L} = {}^{\mathfrak{h}}\mathfrak{L}[{}^{\alpha}\mathtt{f}(\widetilde{u})]$;
- $\mathtt{f}(\widetilde{u}) = \mathtt{L}'$
- $\mathbb{L}' = {}^{\mathfrak{h}}\mathfrak{L}[addh(\alpha\mathtt{f}, \mathtt{L}')]$;
- $\mathfrak{h}(\mathbb{L}) = \mathfrak{h}({}^{\mathfrak{h}}\mathfrak{L}[{}^{\alpha}\mathtt{f}(\widetilde{u})]) = {}^{\mathfrak{h}}\mathtt{T}_1 + \cdots + {}^{\mathfrak{h}}\mathtt{T}_p$;

– $\flat(\mathtt{L}') = \mathtt{T}'_1 + \cdots + \mathtt{T}'_{p'}$;
– $\flat(\mathbb{L}') = {}^{\flat}\mathtt{T}''_1 + \cdots + {}^{\flat}\mathtt{T}''_q$;
– ${}^{\alpha_0}(x_0, x_1)\&\cdots\&{}^{\alpha_n}(x_n, x_0) \in {}^{\flat}\mathtt{T}''_1 + \cdots + {}^{\flat}\mathtt{T}''_q$ (it is a circularity).

Without loss of generality, we may reduce to the following case (the general case is demonstrated by iterating the arguments below).

Let $\alpha\mathtt{f} = \beta(\alpha')^{m+2}\beta'$ and let

$$\begin{aligned}{}^{\alpha_0}(x_0, x_1)\&\cdots\&{}^{\alpha_n}(x_n, x_0) = {}&\&_{0\leqslant j\leqslant n'}{}^{\beta(\alpha')^{m+1}\beta'\beta_j}(x_j, x_{j+1})\\ &\&{}^{\alpha_{n'+1}}(x_{n'+1}, x_{n'+2})\\ &\&\cdots\&{}^{\alpha_n}(x_n, x_0)\end{aligned}$$

with $\varepsilon \lneq \beta_j \preceq \beta''\beta'$, where $\beta'\beta'' = \alpha'$, and $n' < n$ (otherwise 2 is straightforward because the circularity may be mapped to a previous circularity by $\rho^{(m+2)}_{\beta,\alpha',\beta'}$, see Lemma 1(1.b), or it is already contained in $\mathbb{L}$). This is the case of crossover circularities, as discussed in Section 1.

By Lemma 1,

$$\begin{aligned}&{}^{\beta(\alpha')^m\beta'\beta_0}(\rho^{(m+2)}_{\beta,\alpha,\beta'}(x_0), \rho^{(m+2)}_{\beta,\alpha,\beta'}(x_1))\&\cdots\\ &\&\,{}^{\beta(\alpha')^{m+1}\beta'\beta_{n'}}(\rho^{(m+2)}_{\beta,\alpha,\beta'}(x_{n'}), \rho^{(m+2)}_{\beta,\alpha,\beta'}(x_{n'+1}))\end{aligned} \tag{1}$$

is in some ${}^{\flat}\mathtt{T}''_i$. There are two cases.

Case 1: for every $n'+1 \leqslant i \leqslant n$, $\alpha_i \lneq \beta(\alpha')^{m+1}\beta'$. Then, by Lemma 1(1), we have $\rho^{(m+2)}_{\beta,\alpha,\beta'}(x_0) = \rho^{(m+1)}_{\beta,\alpha,\beta'}(x_0)$ and $\rho^{(m+2)}_{\beta,\alpha,\beta'}(x_{n'+1}) = \rho^{(m+1)}_{\beta,\alpha,\beta'}(x_{n'+1})$. Therefore, by Lemma 1(2),

$$\begin{aligned}(1)\&\,{}^{\alpha'_{n'+1}}(\rho^{(m+1)}_{\beta,\alpha,\beta'}(x_{n'+1}), \rho^{(m+1)}_{\beta,\alpha,\beta'}(x_{n'+2}))\\ \&\cdots\&{}^{\alpha'_n}(\rho^{(m+1)}_{\beta,\alpha,\beta'}(x_n), \rho^{(m+1)}_{\beta,\alpha,\beta'}(x_0))\end{aligned}$$

with suitable $\alpha'_{n'+1}, \cdots, \alpha'_n$, is a circularity in ${}^{\flat}\mathtt{T}''_1 + \cdots + {}^{\flat}\mathtt{T}''_q$. In particular, whenever, for every $n'+1 \leqslant i \leqslant n$, $\alpha_i = \beta(\alpha')^m\beta'\beta_i$ with $\varepsilon \lneq \beta_i \preceq \beta''\beta'$, the flashback $\rho^{(m+1)}_{\beta,\alpha,\beta'}$ maps dependencies ${}^{\alpha_i}(x_i, x_{i+1})$ to dependencies

$${}^{\beta(\alpha')^{m-1}\beta'\beta_i}(\rho^{(m+1)}_{\beta,\alpha,\beta'}(x_i), \rho^{(m+1)}_{\beta,\alpha,\beta'}(x_{i+1}))$$

if $m > 0$. It is the identity, if $m = 0$.

Case 2: there is $n'+1 \leqslant i \leqslant n$ such that $\alpha_i \not\preceq \beta(\alpha')^{m+2}\beta'$. Let this i be $n'+1$. For instance, $\beta = \beta'_1(\alpha'')^{m'}\beta''_1$ and $\alpha_{n'+1} = \beta'_1(\alpha'')^{m'+1}\beta''_1(\alpha''')^{m''}\beta'''_1$ with $m' \geqslant 2$ and $m'' \geqslant 2$. In this case it is possible that there is no pair ${}^{\gamma}(y, y')$, with $\gamma \succeq \beta'_1(\alpha'')^{m'}$, to which map ${}^{\alpha_{n'+1}}(x_{n'+1}, x_{n'+2})$ by means of a flashback. To overcome this issue, we consider the flashbacks $\rho_{\alpha_0}, \cdots, \rho_{\alpha_{n'}}, \rho_{\alpha_{n'+1}}$ and we observe that

$$\begin{aligned}&{}^{[\alpha_0]}(\rho_{\alpha_0}(x_0), \rho_{\alpha_0}(x_1))\&\cdots\&\,{}^{[\alpha_{n'}]}(\rho_{\alpha_{n'}}(x_{n'}), \rho_{\alpha_{n'}}(x_{n'+1}))\\ &\&\,{}^{[\alpha_{n'+1}]}(\rho_{\alpha_{n'+1}}(x_{n'+1}), \rho_{\alpha_{n'+1}}(x_{n'+2}))\&\cdots\\ &\&\,{}^{[\alpha_n]}(\rho_{\alpha_n}(x_n), \rho_{\alpha_n}(x_1))\end{aligned} \tag{2}$$

verifies

(a) for every $0 \leqslant i < n$, $\rho_{\alpha_i}(x_{i+1}) = \rho_{\alpha_{i+1}}(x_{i+1})$ and $\rho_{\alpha_n}(x_0) = \rho_{\alpha_0}(x_0)$;
(b) the term (2) is a subterm of ${}^{\flat}\mathtt{T}''_1 + \cdots + {}^{\flat}\mathtt{T}''_q$.

As regards (a), the property derives by definition of the flashbacks ρ_{α_i} and $\rho_{\alpha_{i+1}}$ in Lemma 1. As regards (b), it follows by Lemma 1(2.b) because ${}^{\alpha_0}(x_0, x_1)\&\cdots\&{}^{\alpha_n}(x_n, x_1) \in {}^{\flat}\mathtt{T}''_1 + \cdots + {}^{\flat}\mathtt{T}''_q$.

C Nonlinear programs: technical aspects

When the lam program is not linear recursive, it is not possible to associate a unique mutation to a function. In the general case, our technique for verifying circularity-freedom consists of transforming a nonlinear recursive program into a linear recursive one and then running the algorithm of the previous section. As we will see, the transformation introduces inaccuracies, e.g. dependencies that are not present in the nonlinear recursive program.

C.1 The pseudo-linear case

In nonlinear recursive programs, recursive histories are no more adequate to capture the mutations defined by the functions. For example, in the nonlinear recursive program (called $\mathtt{f'g'}$-program)

$\big(\mathtt{f}'(x,y,z) = (x,y)\&\mathtt{g}'(y,z),\ \mathtt{g}'(x,y) = \mathtt{g}'(x,z)\&\mathtt{f}'(z,y,y),\ \mathtt{L}\big)$

the recursive history of $\mathtt{f'}$ is $\mathtt{f'g'}$. The sequence $\mathtt{f'g'g'}$ *is not* a recursive history because it contains multiple occurrences of the function $\mathtt{g'}$. However, if one computes the sequences of invocations $\mathtt{f}'(x,y,z)\cdots\mathtt{f}'(\widetilde{u})$, it is possible to derive the two sequences $\mathtt{f}'(x,y,z)\mathtt{g}'(y,z)\mathtt{f}'(z',z,z)$ and $\mathtt{f}'(x,y,z)\mathtt{g}'(y,z)\ \mathtt{g}'(y,u)\mathtt{f}'(u',u,u)$ that define two different mutations $[\![\,4,3,3\,]\!]$ and $[\![\,6,5,5\,]\!]$ (see the definition of mutation of a function).

Definition 10. *A program* $\big(\mathtt{f}_1(\widetilde{x_1}) = \mathtt{L}_1, \cdots, \mathtt{f}_\ell(\widetilde{x_\ell}) = \mathtt{L}_\ell, \mathtt{L}\big)$ *is* pseudo-linear recursive *if, for every* $\mathtt{f}_i$*, the set of functions* $\{\mathtt{f} \mid \mathit{closure}(\mathtt{f}) = \mathit{closure}(\mathtt{f}_i)\}$ *contains at most one function with a number of recursive histories greater than 1.*

The $\mathtt{f'g'}$-program above is pseudo-linear recursive, as well as the `fibonacci` program in Section 1 and the following $\mathtt{l'}$-program

$\big(\mathtt{l}'(x,y,z) = (x,y)\&\mathtt{l}'(y,z,x) + (x,u)\&\mathtt{l}'(u,u,y),\ \mathtt{L}\big)$.

In these cases, functions have a unique recursive history but there are multiple recursive invocations. On the contrary, the $\mathtt{f''g''}$-program below

$$\begin{pmatrix} \mathtt{f}''(x,y) = (x,z)\&\mathtt{f}''(y,z) + \mathtt{g}''(y,x)\ , \\ \mathtt{g}''(x,y) = (y,x)\&\mathtt{f}''(y,z)\&\mathtt{g}''(z,x)\ , \\ \mathtt{f}''(x_1,x_2) \quad \end{pmatrix}$$

is not pseudo-linear recursive.

Pseudo-linearity has been introduced because of the easiness of transforming them into linear recursive programs. The transformation consists of the three

$$\frac{\begin{array}{c}\mathit{rechis}(\mathtt{f}_i)=\{\mathtt{f}_i\mathtt{f}_k\alpha,\mathtt{f}_i\beta_0,\cdots,\mathtt{f}_i\beta_n\}\qquad\{\mathit{head}(\beta_0),\cdots,\mathit{head}(\beta_n)\}\backslash\mathtt{f}_k\neq\varnothing\\ \mathtt{L}_i=\mathfrak{L}[\mathtt{f}_k(\widetilde{u})]\qquad \mathit{var}(\mathtt{L}_k)\backslash\widetilde{x_k}=\widetilde{z}\qquad \widetilde{w}\text{ are fresh}\end{array}}{\big(\cdots\mathtt{f}_i(\widetilde{x_i})=\mathtt{L}_i,\cdots,\mathtt{L}\big)\ \overset{\mathtt{pl}\mapsto\mathtt{l}}{\Longmapsto}_1\ \big(\cdots\mathtt{f}_i(\widetilde{x_i})=\mathfrak{L}[\mathtt{L}_k[\widetilde{w}/\widetilde{z}][\widetilde{u}/\widetilde{x_i}]],\cdots,\mathtt{L}\big)}$$

$$\frac{\begin{array}{c}\mathit{rechis}(\mathtt{f}_i)=\{\mathtt{f}_i\alpha\}\qquad \mathtt{f}_k=\mathit{head}(\alpha)\\ \mathtt{L}_i=\mathfrak{L}[\mathtt{f}_k(\widetilde{u_0})]\cdots[\mathtt{f}_k(\widetilde{u_{n+1}})]\qquad \mathtt{f}_k\notin\mathfrak{L}\\ \mathit{var}(\mathtt{L}_k)\backslash\widetilde{x_k}=\widetilde{z}\qquad \widetilde{w_0},\cdots,\widetilde{w_{n+1}}\text{ are fresh}\\ \mathfrak{L}[\mathtt{L}_k[\widetilde{w_0}/\widetilde{z}][\widetilde{u_0}/\widetilde{x_k}]]\cdots[\mathtt{L}_k[\widetilde{w_{n+1}}/\widetilde{z}][\widetilde{u_{n+1}}/\widetilde{x_k}]]=\mathtt{L}'_i\end{array}}{\big(\cdots\mathtt{f}_i(\widetilde{x_i})=\mathtt{L}_i,\cdots,\mathtt{L}\big)\ \overset{\mathtt{pl}\mapsto\mathtt{l}}{\Longmapsto}_2\ \big(\cdots\mathtt{f}_i(\widetilde{x_i})=\mathtt{L}'_i,\cdots,\mathtt{L}\big)}$$

$$\frac{\begin{array}{c}\mathtt{L}_i=\mathfrak{L}[\mathtt{f}_i(\widetilde{u_0})]\cdots[\mathtt{f}_i(\widetilde{u_{n+1}})]\qquad \mathtt{f}_i\notin\mathfrak{L}\qquad \widetilde{w_0},\cdots,\widetilde{w_{n+1}}\text{ are fresh}\\ \mathtt{L}_i^{aux}=\mathtt{f}_i^{aux}(\widetilde{u_0}[\widetilde{w_0}/\widetilde{x_i}],\cdots,\widetilde{u_{n+1}}[\widetilde{w_{n+1}}/\widetilde{x_i}])\&(\&_{j\in 0..n+1}\flat_{\mathtt{f}_i}(\mathtt{L}_i)[\widetilde{w_j}/\widetilde{x_i}])\end{array}}{\begin{array}{l}\big(\cdots\mathtt{f}_i(\widetilde{x_i})=\mathtt{L}_i,\cdots,\mathtt{L}\big)\ \overset{\mathtt{pl}\mapsto\mathtt{l}}{\Longmapsto}_3\\ \quad\big(\cdots\mathtt{f}_i(\widetilde{x_i})=\mathtt{f}_i^{aux}(\underbrace{\widetilde{x_i},\cdots,\widetilde{x_i}}_{n+2\text{ times}}),\ \mathtt{f}_i^{aux}(\widetilde{w_0},\cdots,\widetilde{w_{n+1}})=\mathtt{L}_i^{aux},\cdots,\mathtt{L}\big)\end{array}}$$

Table 1. Pseudo-linear to linear transformation

steps specified in Table 1, which we discuss below. Let $\big(\mathtt{f}_1(\widetilde{x_1})=\mathtt{L}_1,\cdots,\mathtt{f}_\ell(\widetilde{x_\ell})=\mathtt{L}_\ell,\mathtt{L}\big)$ be a lam program, let $\mathit{rechis}(\mathtt{f}_i)$ be the set of recursive histories of $\mathtt{f}_i$, and let $\mathit{head}(\varepsilon)=\varepsilon$ and $\mathit{head}(\mathtt{f}\alpha)=\mathtt{f}$.

Transformation $\overset{\mathtt{pl}\mapsto\mathtt{l}}{\Longmapsto}_1$*: Removing multiple recursive histories.* We repeatedly apply the rule defining $\overset{\mathtt{pl}\mapsto\mathtt{l}}{\Longmapsto}_1$. Every instance of the rule selects a function $\mathtt{f}_i$ with a number of recursive histories greater than one – the hypotheses $\mathit{rechis}(\mathtt{f}_i)=\{\mathtt{f}_i\mathtt{f}_k\alpha,\mathtt{f}_i\beta_0,\cdots,\mathtt{f}_i\beta_n\}$ and $\{\mathit{head}(\beta_0),\cdots,\mathit{head}(\beta_n)\}\backslash\mathtt{f}_k\neq\varnothing$ – and expands the invocation of $\mathtt{f}_k$, with $\mathtt{f}_k\neq\mathtt{f}_i$. By definition of pseudo-linearity, the other function names in $\mathit{rechis}(\mathtt{f}_i)$ have one recursive history. At each application of the rule the sum of the lengths of the recursive histories of $\mathtt{f}_i$ decreases. Therefore we eventually unfold the (mutual) recursive invocations of $\mathtt{f}_i$ till the recursive history of $\mathtt{f}_i$ is unique. For example, the program

$$\big(\mathtt{f}(x)=(x,y)\&\mathtt{g}(x)\ ,\ \mathtt{g}(x)=(x,y)\&\mathtt{f}(x)+\mathtt{g}(y)\ ,\ \mathtt{L}\ \big)$$

is transformed into

$$\big(\mathtt{f}(x)=(x,y)\&\mathtt{g}(x)\ ,\ \mathtt{g}(x)=(x,y)\&(x,z)\&\mathtt{g}(x)+\mathtt{g}(y)\ ,\ \mathtt{L}\ \big).$$

Transformation $\overset{\mathtt{pl}\mapsto\mathtt{l}}{\Longmapsto}_2$*: Reducing the histories of pseudo-linear recursive functions.* By $\overset{\mathtt{pl}\mapsto\mathtt{l}}{\Longmapsto}_1$, we are reduced to functions that have one recursive history. Yet, this is not enough for a program to be linear recursive, such as the $\mathtt{l}'$-program or the following $\mathtt{h}''\mathtt{l}''$-program

$$\begin{array}{l}\big(\ \mathtt{h}''(x,y)=(x,z)\&\mathtt{l}''(y,z)+\mathtt{l}''(y,x)\ ,\\ \ \ \mathtt{l}''(x,y)=(y,x)\&\mathtt{h}''(y,z)\&\mathtt{h}''(z,x)\ ,\\ \ \ \mathtt{h}''(x_1,x_2)\quad\big)\end{array}$$

(the reason is that the bodies of functions may have different invocations of a same function). Rule $\overset{\mathtt{pl}\mapsto\mathtt{l}}{\Longmapsto}_2$ expands the bodies of pseudo-linear recursive functions till the histories of nonlinear recursive functions have length one. In this rule (and in the following), we use lam contexts with multiple holes, written $\mathfrak{L}[\,]\cdots[\,]$. We write $\mathtt{f} \notin \mathfrak{L}$whenever there is no invocation of $\mathtt{f}$ in $\mathfrak{L}$.

By the hypotheses of the rule, it applies to a function $\mathtt{f}_i$ whose next element in the recursive history is $\mathtt{f}_k$ (by definition of the recursive history, $\mathtt{f}_i \neq \mathtt{f}_k$) and whose body $\mathtt{L}_i$ contains *at least* two invocations of $\mathtt{f}_k$. The rule transforms $\mathtt{L}_i$ by expanding every invocation of $\mathtt{f}_k$. For example, the functions $\mathtt{h}''$ and $\mathtt{l}''$ in the $\mathtt{h}''\mathtt{l}''$-program are transformed into

$$\begin{aligned}\mathtt{h}''(x,y) &= (x,z)\&\mathtt{l}''(y,z) + \mathtt{l}''(y,x)\ ,\\ \mathtt{l}''(x,y) &= (y,x)\&((y,z')\&\mathtt{l}''(z,z') + \mathtt{l}''(z,y))\\ &\quad\ \&((z,z'')\&\mathtt{l}''(x,z'') + \mathtt{l}''(x,z)).\end{aligned}$$

The arguments about the termination of the transformation $\overset{\mathtt{pl}\mapsto\mathtt{l}}{\Longmapsto}_2$ are straightforward.

Transformation $\overset{\mathtt{pl}\mapsto\mathtt{l}}{\Longmapsto}_3$: Removing nonlinear recursive invocations. By $\overset{\mathtt{pl}\mapsto\mathtt{l}}{\Longmapsto}_2$ we are reduced to pseudo-linear recursive programs where the nonlinearity is due to recursive, but not mutually-recursive functions (such as `fibonacci`). The transformation $\overset{\mathtt{pl}\mapsto\mathtt{l}}{\Longmapsto}_3$ removes multiple recursive invocations of nonlinear recursive programs. This transformation is the one that introduces inaccuracies, e.g. pairs that are not present in the nonlinear recursive program.

In the rule of $\overset{\mathtt{pl}\mapsto\mathtt{l}}{\Longmapsto}_3$ we use the auxiliary operator $\flat_{\mathtt{f}}(\mathtt{L})$ defined as follows:

$$\begin{aligned}&\flat_{\mathtt{f}}(\mathtt{0}) = \mathtt{0}, && \flat_{\mathtt{f}}((x,y)) = (x,y),\\ &\flat_{\mathtt{f}}(\mathtt{f}(\widetilde{x})) = \mathtt{0}, && \flat_{\mathtt{f}}(\mathtt{g}(\widetilde{x})) = \mathtt{g}(\widetilde{x}),\ \text{if } (\mathtt{f} \neq \mathtt{g}),\\ &\flat_{\mathtt{f}}(\mathtt{L}\&\mathtt{L}') = \flat_{\mathtt{f}}(\mathtt{L})\&\flat_{\mathtt{f}}(\mathtt{L}'), && \flat_{\mathtt{f}}(\mathtt{L} + \mathtt{L}') = \flat_{\mathtt{f}}(\mathtt{L}) + \flat_{\mathtt{f}}(\mathtt{L}').\end{aligned}$$

The rule of $\overset{\mathtt{pl}\mapsto\mathtt{l}}{\Longmapsto}_3$ selects a function $\mathtt{f}_i$ whose body contains multiple recursive invocations and extracts all of them – the term $\flat_{\mathtt{f}_i}(\mathtt{L}_i)$. This term is put in parallel with an auxiliary function invocation – the function $\mathtt{f}_i^{aux}$ – that collects the arguments of each invocation $\mathtt{f}_i$ (with names that have been properly renamed). The resulting term, called $\mathtt{L}_i^{aux}$ is the body of the new function $\mathtt{f}_i^{aux}$ that is invoked by $\mathtt{f}_i$ in the transformed program. For example, the function `fibonacci`

$$\mathtt{fibonacci}(r,s) = (r,s)\&(t,s)\&\mathtt{fibonacci}(r,t)\&\mathtt{fibonacci}(t,s)$$

is transformed into

$$\begin{aligned}\mathtt{fibonacci}(r,s) &= \mathtt{fibonacci}^{aux}(r,s,r,s),\\ \mathtt{fibonacci}^{aux}(r,s,r',s') &= (r,s)\&(r',s')\\ &\quad\ \&\mathtt{fibonacci}^{aux}(r,t,t,s')\end{aligned}$$

where different invocations ($\mathtt{fibonacci}(r,s)$ and $\mathtt{fibonacci}(r',s')$) in the original program are contracted into one auxiliary function invocation ($\mathtt{fibonacci}^{aux}(r,s,r',s')$). As a consequence of this step, the creations of names performed by different invocations are contracted to names created by one invocation. This

leads to merging dependencies, which, in turn, reduces the precision of the analysis. (As discussed in Section 1, a cardinality argument prevents the inaccuracies introduced by $\overset{\mathtt{pl}\mapsto\mathtt{l}}{\Longmapsto}_3$ from being totally eliminated.)

As far as the correctness of the transformations in Table 1 is concerned, we begin by defining a correspondence between states of a pseudo-linear program and those of a linear one. We focus on $\overset{\mathtt{pl}\mapsto\mathtt{l}}{\Longmapsto}_3$ because the proofs of the correctness of the other transformations are straightforward.

Definition 11. *Let $\mathcal{L}_2$ be the linear program returned by the Transformation 3 of Table 1 applied to $\mathcal{L}_1$. A state $\langle \mathbb{V}_1, \mathtt{L}_1 \rangle$ of $\mathcal{L}_1$ is* linearized to *a state $\langle \mathbb{V}_2, \mathtt{L}_2 \rangle$ of $\mathcal{L}_2$, written $\langle \mathbb{V}_1, \mathtt{L}_1 \rangle \ni_{\mathtt{lin}} \langle \mathbb{V}_2, \mathtt{L}_2 \rangle$, if there exists a surjection σ such that:*

1. *if $(x, y) \in \mathbb{V}_1$ then $(\sigma(x), \sigma(y)) \in \mathbb{V}_2$.*
2. *if $\flat(\mathtt{L}_1) = \mathtt{T}_1 + \cdots + \mathtt{T}_m$ and $\flat(\mathtt{L}_2) = \mathtt{T}'_1 + \cdots + \mathtt{T}'_n$, then for every $1 \leqslant i \leqslant m$, there exists $1 \leqslant j \leqslant n$, such that $\sigma(\mathtt{T}_i) \in \mathtt{T}'_j$;*
3. *if $\mathtt{f}(\widetilde{x}_1) \in \mathtt{L}_1$ then either (1) $\mathtt{f}(\sigma(\widetilde{x}_1))$ in $\mathtt{L}_2$ or (2) there are $\mathtt{f}(\widetilde{x}_2) \cdots \mathtt{f}(\widetilde{x}_k)$ in $\mathtt{L}_1$ and $\mathtt{f}^{aux}(\widetilde{y}_1, \cdots, \widetilde{y}_h)$ in $\mathtt{L}_2$ such that, for every $1 \leqslant k' \leqslant k$ there exists h' with $\sigma(\widetilde{x}_{k'}) = \widetilde{y}_{h'}$;*

In the following lemma we use the notation $\mathfrak{L}[\mathtt{L}_1] \cdots [\mathtt{L}_n]$ defined in terms of standard lam context by $(\cdots((\mathfrak{L}[\mathtt{L}_1])[\mathtt{L}_2]) \cdots)[\mathtt{L}_n]$.

Lemma 2. *Let $\langle \mathbb{V}_1, \mathtt{L}_1 \rangle \ni_{\mathtt{lin}} \langle \mathbb{V}_2, \mathtt{L}_2 \rangle$. Then, $\langle \mathbb{V}_2, \mathtt{L}_2 \rangle \longrightarrow \langle \mathbb{V}'_2, \mathtt{L}'_2 \rangle$ implies there exists $\langle \mathbb{V}_1, \mathtt{L}_1 \rangle \longrightarrow^* \langle \mathbb{V}'_1, \mathtt{L}'_1 \rangle$ such that $\langle \mathbb{V}'_1, \mathtt{L}'_1 \rangle \ni_{\mathtt{lin}} \langle \mathbb{V}'_2, \mathtt{L}'_2 \rangle$*

Proof. Base case. Initially $\mathtt{L}_1 = \mathtt{L}_2$ because the main lam is not affected by the transformation. Therefore the first step can only be an invocation of a standard function belonging to both programs. We have two cases:

1. the function was linear already in the original program, thus it was not modified by the transformation. In this case the two programs performs the same reduction step and end up in the same state.
2. the function has been *linearized* by the transformation. In this case the invocation at the linear side will reduce to an invocation of an *aux*-function and it will not produce new pairs nor new names. The corresponding reduction in $\langle \mathbb{V}_1, \mathtt{L}_1 \rangle$ is a zero-step reduction. It is easy to verify that $\langle \mathbb{V}_1, \mathtt{L}_1 \rangle \ni_{\mathtt{lin}} \langle \mathbb{V}'_2, \mathtt{L}'_2 \rangle$.

Inductive case. We consider only the case in which the selected function is an *aux*-function. The other case is as in the base case. Let

$$\begin{array}{l} \langle \mathbb{V}_1^{(n)},\ \mathfrak{L}_1^{(n)}[\mathtt{f}(\widetilde{v}_1)] \cdots [\mathtt{f}(\widetilde{v}_k)] \rangle \\ \qquad\qquad\qquad \ni_{\mathtt{lin}} \quad \langle \mathbb{V}_2^{(n)},\ \mathfrak{L}_2^{(n)}[\mathtt{f}^{aux}(\widetilde{u}_1, \cdots, \widetilde{u}_h)] \rangle \end{array}$$

Without loss of generality we can assume that $\mathfrak{L}_1^{(n)}$ does not contain other invocations to $\mathtt{f}$ and the "linearized to" relationship makes $\mathtt{f}(\widetilde{v}_1) \& \cdots \& \mathtt{f}(\widetilde{v}_k)$ correspond to $\mathtt{f}^{aux}(\widetilde{u}_1, \cdots, \widetilde{u}_h)$.Then we have

$$\begin{array}{l} \langle \mathbb{V}_2^{(n)},\ \mathfrak{L}_2^{(n)}[\mathtt{f}^{aux}(\widetilde{u}_1, \cdots, \widetilde{u}_h)] \rangle \longrightarrow \\ \langle \mathbb{V}_2^{(n)} \oplus \widetilde{u}_1, \cdots, \widetilde{u}_h \mathrel{<} \widetilde{w},\ \mathfrak{L}_2^{(n)}[\mathtt{L}_{\mathtt{f}^{aux}}[\widetilde{w}/\widetilde{z}][\widetilde{u}_1, \cdots, \widetilde{u}_h / \widetilde{y}_1, \ldots, \widetilde{y}_h]] \rangle \end{array}$$

where, $\mathtt{f}^{aux}(\widetilde{y}_1, \cdots, \widetilde{y}_h) = \mathtt{L}_{\mathtt{f}^{aux}}$, $var(\mathtt{L}_{\mathtt{f}^{aux}}) \backslash \widetilde{y}_1 \cdots \widetilde{y}_h = \widetilde{z}$ and $\widetilde{w}$ are fresh names. By construction,

$$\mathtt{L}_{\mathtt{f}^{aux}} = \mathtt{f}^{aux}(\widetilde{y'_1}[\widetilde{y_1}/\widetilde{y}], \cdots, \widetilde{y'_k}[\widetilde{y_k}/\widetilde{y}]) \,\&\, \&_{i \in 1..k}(\flat_{\mathtt{f}}(\mathtt{L}_{\mathtt{f}})[\widetilde{y_i}/\widetilde{y}])$$

where $\mathtt{f}(\widetilde{y}) = \mathfrak{L}_{\mathtt{f}}[\mathtt{f}(\widetilde{y'_1})] \cdots [\mathtt{f}(\widetilde{y'_k})] = \mathtt{L}_{\mathtt{f}}$ and $\mathtt{f} \notin \mathfrak{L}_{\mathtt{f}}$.

The corresponding reduction steps of $\langle \mathbb{V}_1^{(n)}, \mathfrak{L}_1^{(n)}[\mathtt{f}(\widetilde{v}_1)] \cdots [\mathtt{f}(\widetilde{v}_k)] \rangle$ are the following ones:

$$\begin{array}{l} \langle \mathbb{V}_1^{(n)}, \mathfrak{L}_1^{(n)}[\mathtt{f}(\widetilde{v}_1)] \cdots [\mathtt{f}(\widetilde{v}_k)] \rangle \xrightarrow{\mathtt{f}(\widetilde{v}_1)} \cdots \xrightarrow{\mathtt{f}(\widetilde{v}_k)} \\ \langle \mathbb{V}_1^{(n)} \oplus \widetilde{v}_1 < \widetilde{w}_1 \oplus \cdots \oplus \widetilde{v}_k < \widetilde{w}_k, \ \mathfrak{L}_1^{(n)}[\mathtt{L}_{\mathtt{f}}[\widetilde{v}_1/\widetilde{y}]] \cdots [\mathtt{L}_{\mathtt{f}}[\widetilde{v}_k/\widetilde{y}]] \rangle \end{array}$$

and $\widetilde{w}_i$ are the fresh names created by the invocation $\mathtt{f}(\widetilde{v}_i)$, $1 \leqslant i \leqslant k$. We need to show that:

$$\begin{array}{l} \langle \mathbb{V}_1^{(n)} \oplus \widetilde{v}_1 < \widetilde{w}_1 \oplus \cdots \oplus \widetilde{v}_k < \widetilde{w}_k, \ \mathfrak{L}_1^{(n)}[\mathtt{L}_{\mathtt{f}}[\widetilde{v}_1/\widetilde{y}]] \cdots [\mathtt{L}_{\mathtt{f}}[\widetilde{v}_k/\widetilde{y}]] \rangle \\ \qquad\qquad \ni_{\mathtt{lin}} \\ \langle \mathbb{V}_2^{(n)} \oplus \widetilde{u}_1, \cdots, \widetilde{u}_h < \widetilde{w}, \ \mathfrak{L}_2^{(n)}[\mathtt{L}_{\mathtt{f}}^{aux}] \rangle \end{array}$$

where

$$\mathtt{L}_{\mathtt{f}}^{aux} = \mathtt{f}^{aux}(\widetilde{y'_1}[\widetilde{u_1}/\widetilde{y}], \cdots, \widetilde{y'_k}[\widetilde{u_k}/\widetilde{y}]) \,\&\, \mathtt{L}^{aux} \text{ and } \mathtt{L}^{aux} = \&_{i \in 1..k}(\flat_{\mathtt{f}}(\mathtt{L}_{\mathtt{f}})[\widetilde{u_i}/\widetilde{y}])[\widetilde{w}/\widetilde{z}].$$

To this aim we observe that:

- for every $1 \leqslant k' \leqslant k$ there exists h' such that $\sigma(\widetilde{v}_{k'}) = \widetilde{u}_{h'}$; moreover $\widetilde{w} = \sigma(\widetilde{w_1}) = \cdots = \sigma(\widetilde{w_k})$. This satisfies condition *1* of Definition 11;
- if $(a, b) \in \mathtt{L}_{\mathtt{f}}[\widetilde{v_i}/\widetilde{y}]$, with $a, b \in \widetilde{w}_i, \widetilde{v}_i$, then $(\sigma(a), \sigma(b)) \in \flat_{\mathtt{f}}(\mathtt{L}_{\mathtt{f}})[\widetilde{u_i}/\widetilde{y}][\widetilde{w}/\widetilde{z}]$, being σ defined as in the previous item, therefore $\sigma(a), \sigma(b) \in \widetilde{w}, \widetilde{u}_i$. Notice that, due to the $\&_{i \in 1..k}$ composition in the body of $\mathtt{f}^{aux}$, two pairs sequentially composed in $\mathtt{L}_{\mathtt{f}}$ may end up in parallel (through σ). The converse never happens. Therefore condition *2* of Definition 11 is satisfied.
- if $\mathtt{g}(\widetilde{a}) \in \mathtt{L}_{\mathtt{f}}$ we can reason as in the previous item. We notice that function invocations $\mathtt{g}(\widetilde{u})$ that have no counterpart (through σ) in $\mathtt{L}_{\mathtt{f}}[\widetilde{v_i}/\widetilde{y}]$ may be cointained in $\&_{i \in 1..k}(\flat_{\mathtt{f}}(\mathtt{L}_{\mathtt{f}})[\widetilde{u_i}/\widetilde{y}])[\widetilde{w}/\widetilde{z}]$. We do not have to mind about them because the lemma guarantees the converse containment.
- in $\mathtt{L}_{\mathtt{f}}[\widetilde{v_i}/\widetilde{y}]$ we have k new invocations of $\mathtt{f}(\widetilde{b}_{i,1}) \cdots \mathtt{f}(\widetilde{b}_{i,k})$, where $\widetilde{b}_{i,j} = \widetilde{y'_j}[\widetilde{v_j}/\widetilde{y}][\widetilde{w_j}/\widetilde{z}]$. Therefore in the pseudolinear lam we have k^2 invocations of $\mathtt{f}$, while in the corresponding linear lam we find just one invocation of $\mathtt{f}^{aux}(\widetilde{y'_1}[\widetilde{u_1}/\widetilde{y}][\widetilde{w}/\widetilde{z}], \cdots, \widetilde{y'_k}[\widetilde{u_k}/\widetilde{y}][\widetilde{w}/\widetilde{z}])$. We notice that the surjection σ is such that $(\widetilde{y'_j}[\widetilde{u_j}/\widetilde{y}][\widetilde{w}/\widetilde{z}] = \sigma(\widetilde{b}_{1,j}) = \cdots = \sigma(\widetilde{b}_{k,j})$, with $1 \leqslant j \leqslant k$. This, together with the previous item, satisfies condition *3* of Definition 11.

Lemma 3. *Let* $\langle \mathbb{V}_1, \mathtt{L}_1 \rangle \ni_{\mathtt{lin}} \langle \mathbb{V}_2, \mathtt{L}_2 \rangle$ *and* $\langle \mathbb{V}_1, \mathtt{L}_1 \rangle \longrightarrow^* \langle \mathbb{V}'_1, \mathtt{L}'_1 \rangle$. *Then there are* $\langle \mathbb{V}'_1, \mathtt{L}'_1 \rangle \longrightarrow^* \langle \mathbb{V}''_1, \mathtt{L}''_1 \rangle$ *and* $\langle \mathbb{V}_2, \mathtt{L}_2 \rangle \longrightarrow^* \langle \mathbb{V}'_2, \mathtt{L}'_2 \rangle$ *such that* $\langle \mathbb{V}''_1, \mathtt{L}''_1 \rangle \ni_{\mathtt{lin}} \langle \mathbb{V}'_2, \mathtt{L}'_2 \rangle$

Proof. A straightforward induction on the length of $\langle \mathbb{V}_1, \mathtt{L}_1 \rangle \longrightarrow^* \langle \mathbb{V}'_1, \mathtt{L}'_1 \rangle$. In the inductive step, we need to expand the recursive invocations "at a same level" in order to mimic the behavior of functions $\mathtt{f}^{aux}$.

Theorem 4. *Let $\mathcal{L}_1$ be a pseudo-linear program and $\mathcal{L}_2$ be the result of the transformations in Table 1. If a saturated state of $\mathcal{L}_2$ has no circularity then no state of $\mathcal{L}_1$ has a circularity.*

Proof. The transformations $\overset{\mathtt{pl}\mapsto\mathtt{l}}{\Longmapsto}_1$ and $\overset{\mathtt{pl}\mapsto\mathtt{l}}{\Longmapsto}_2$ perform expansions and do not introduce inaccuracies. By Lemma 2, for every $\langle \mathbb{V}_2, \mathtt{L}_2 \rangle$ reached by evaluating $\mathcal{L}_2$, there is $\langle \mathbb{V}_1, \mathtt{L}_1 \rangle$ that is reached by evaluating $\mathcal{L}_1$ such that $\langle \mathbb{V}_1, \mathtt{L}_1 \rangle \ni_{\mathtt{lin}} \langle \mathbb{V}_2, \mathtt{L}_2 \rangle$. This guarantees that every circularity in $\langle \mathbb{V}_1, \mathtt{L}_1 \rangle$ is also present in $\langle \mathbb{V}_2, \mathtt{L}_2 \rangle$. We conclude by Lemma 3 and Theorem 2.

We observe that, our analysis returns that the `fibonacci` program is circularity-free.

C.2 The general case

In non-pseudo-linear recursive programs, more than one mutual recursive function may have several recursive histories. The transformation $\overset{\mathtt{npl}\mapsto\mathtt{pl}}{\Longmapsto}$ in Table 2 takes a non-pseudo-linear recursive program and returns a program where the "non-pseudo-linearity" is simpler. Repeatedly applying the transformation, at the end, one obtains a pseudo-linear recursive program.

More precisely, let $\big(\mathtt{f}_1(\widetilde{x_1}) = \mathtt{L}_1, \cdots, \mathtt{f}_\ell(\widetilde{x_\ell}) = \mathtt{L}_\ell, \mathtt{L}\big)$ be a non-pseudo-linear recursive program. Therefore, there are at least two functions with more than one recursive history. One of this function is $\mathtt{f}_j$, which is the one that is being explored by the rule $\overset{\mathtt{npl}\mapsto\mathtt{pl}}{\Longmapsto}$. Let also $\mathtt{f}_i$ be another function such that $\mathit{closure}(\mathtt{f}_j) = \mathit{closure}(\mathtt{f}_i)$ (this $\mathtt{f}_i$ must exists otherwise the program would be already pseudo-linear recursive). These constraints are those listed in the first line of the premises of the rule. The idea of this transformation is to defer the invocations of the functions in $\{\mathit{head}(\alpha_1\mathtt{f}_j), \cdots, \mathit{head}(\alpha_{h+1}\mathtt{f}_j)\}\backslash\mathtt{f}_i$, i.e., the functions different from $\mathtt{f}_i$ that can be invoked within $\mathtt{f}_j$'s body, to the body of the function $\mathtt{f}_i$. The meaning of the second and third lines of the premises of the rule is to identify the p_k different invocations of these m functions ($k \geqslant m$). Notice that every $\alpha_1, \cdots, \alpha_{h+1}$ could be empty, meaning that $\mathtt{f}_j$ is directly called. At this point, what we need to do is (1) to store the arguments of each invocation of $\mathtt{f}_{i_1}, \cdots, \mathtt{f}_{i_m}$ into those of an invocation of $\mathtt{f}_i$ – actually, a suitable tuple of them, thus the arity of $\mathtt{f}_i$ is augmented correspondingly – and (2) to perform suitable expansions in the body of $\mathtt{f}_i$. In order to augment the arguments of the invocations of $\mathtt{f}_i$ that occur in the other parts of the program, we use the auxiliary rule $\overset{\mathtt{f}_i,n}{\Longmapsto}$ that extends every invocation of $\mathtt{f}_i$ with n additional arguments that are always fresh names. The fourth line of the premises calculates the number n of additional arguments, based on the number of arguments of the functions that are going to be moved into $\mathtt{f}_i$'s body. The last step, described in the last

$$\frac{\mathtt{f} \notin \mathfrak{L} \qquad \widetilde{z_1}, \cdots, \widetilde{z_m} \text{ are } n\text{-tuple of fresh names}}{\mathfrak{L}[\mathtt{f}(\widetilde{u_1})] \cdots [\mathtt{f}(\widetilde{u_m})] \overset{\mathtt{f},n}{\Longmapsto} \mathfrak{L}[\mathtt{f}(\widetilde{u_1}, \widetilde{z_1})] \cdots [\mathtt{f}(\widetilde{u_m}, \widetilde{z_m})]}$$

$$\frac{\begin{array}{c} rechis(\mathtt{f}_j) = \{\mathtt{f}_j\mathtt{f}_i\alpha_0, \mathtt{f}_j\alpha_1, \cdots, \mathtt{f}_j\alpha_{h+1}\} \qquad \mathtt{f} \in \mathtt{f}_i\alpha_0 \qquad \sharp(rechis(\mathtt{f})) > 1 \\ \{\mathtt{f}_{i_1}, \cdots, \mathtt{f}_{i_m}\} = \{head(\alpha_1\mathtt{f}_j), \cdots, head(\alpha_{h+1}\mathtt{f}_j)\} \backslash \mathtt{f}_i \\ \mathtt{L}_j = \mathfrak{L}[\mathtt{f}_{p_1}(\widetilde{u_1})] \cdots [\mathtt{f}_{p_k}(\widetilde{u_k})] \qquad \{\mathtt{f}_{p_1}, \cdots, \mathtt{f}_{p_k}\} = \{\mathtt{f}_{i_1}, \cdots, \mathtt{f}_{i_m}\} \qquad \mathtt{f}_{i_1}, \cdots, \mathtt{f}_{i_m} \notin \mathfrak{L} \\ n = \sharp(\widetilde{u_1} \cdots \widetilde{u_k}) \qquad (\mathtt{L}_h \overset{\mathtt{f}_i,n}{\Longmapsto} \mathtt{L}'_h)^{h \in \{1, \cdots, \ell+1\}} \qquad \mathtt{L}'_j = \mathfrak{L}'[\mathtt{f}_{i_1}(\widetilde{u_1})] \cdots [\mathtt{f}_{i_m}(\widetilde{u_k})] \\ \widetilde{z_1^1}, \cdots, \widetilde{z_k^1}, \cdots, \widetilde{z_1^k}, \cdots, \widetilde{z_k^k}, \widetilde{z_1}, \cdots, \widetilde{z_k}, \text{ are fresh} \\ \mathtt{L}''_j = \mathfrak{L}'[\mathtt{f}_i(\widetilde{z_1^1}, \widetilde{u_1}, \widetilde{z_2^1}, \cdots, \widetilde{z_k^1})] \cdots [\mathtt{f}_i(\widetilde{z_1^k}, \cdots, \widetilde{z_k^k}, \widetilde{u_k})] \end{array}}{\begin{array}{l} (\mathtt{f}_1(\widetilde{x_1}) = \mathtt{L}_1, \cdots \mathtt{f}_i(\widetilde{x_i}) = \mathtt{L}_i, \cdots, \mathtt{f}_j(\widetilde{x_j}) = \mathtt{L}_j, \cdots, \mathtt{f}_\ell(\widetilde{x_\ell}) = \mathtt{L}_\ell, \mathtt{L}_{\ell+1}) \overset{\mathtt{npl}\mapsto\mathtt{pl}}{\Longmapsto} \\ (\mathtt{f}_1(\widetilde{x_1}) = \mathtt{L}'_1, \cdots \mathtt{f}_i(\widetilde{x_i}, \widetilde{z_1}, \cdots, \widetilde{z_k}) = \mathtt{L}'_i \& (\&_{q \in 1..k} \mathtt{f}_{pq}(\widetilde{z_q})), \cdots, \mathtt{f}_j(\widetilde{x_j}) = \mathtt{L}''_j, \cdots, \mathtt{f}_\ell(\widetilde{x_\ell}) = \mathtt{L}'_\ell, \mathtt{L}'_{\ell+1}) \end{array}}$$

Table 2. Non-pseudo-linear to pseudo-linear transformation

line of the premises of the rule, is to replace the invocations of the functions $\mathtt{f}_{i_1}, \cdots, \mathtt{f}_{i_m}$ with invocations of $\mathtt{f}_i$. Notice that, in each invocation, the position of the actual arguments is different. In the body of $\mathtt{f}_i$, after the transformation, the invocations of those functions will be performed passing the right arguments. For example, the $\mathtt{f}''\mathtt{g}''$-program

$$\begin{array}{l} \big(\ \mathtt{f}''(x,y) = (x,z)\&\mathtt{f}''(y,z) + \mathtt{g}''(y,x) \ , \\ \quad \mathtt{g}''(x,y) = (y,x)\&\mathtt{f}''(y,z)\&\mathtt{g}''(z,x) \ , \\ \quad \mathtt{f}''(x_1,x_2) \quad \big) \end{array}$$

is rewritten into

$$\begin{array}{l} \big(\ \mathtt{f}''(x,y) = (x,z)\&\mathtt{g}''(x',y',y,z) + \mathtt{g}''(y,x,z',z'') \ , \\ \quad \mathtt{g}''(x,y,u,v) = (y,x)\&\mathtt{f}''(y,z)\&\mathtt{g}''(z,x,x',y')\&\mathtt{f}''(u,v) \ , \\ \quad \mathtt{f}''(x_1,x_2) \quad \big) \ . \end{array}$$

The invocation $\mathtt{f}''(y,z)$ is moved into the body of $\mathtt{g}''$. The function $\mathtt{g}''$ has an augmented arity, so that its first two arguments refer to the arguments of the invocations of $\mathtt{g}''$ in the original program, and the last two arguments refer to the invocation of $\mathtt{f}''$. Looking at the body of $\mathtt{g}''$, the unchanged part (with the augmented arity of $\mathtt{g}''$) covers the first two arguments; whilst the last two arguments are only used for a new invocation of $\mathtt{f}''$.

The correctness of $\overset{\mathtt{npl}\mapsto\mathtt{pl}}{\Longmapsto}$ is demonstrated in a similar way to the proof of the correctness of $\overset{\mathtt{pl}\mapsto\mathtt{l}}{\Longmapsto}_3$. We begin by defining a correspondence between states of a non-pseudo-linear program and those of a pseudo-linear one.

Definition 12. *Let $\mathcal{L}_2$ be the pseudo-linear program returned by the transformation of Table 2 applied to $\mathcal{L}_1$. A state $\langle \mathbb{V}_1, \mathtt{L}_1 \rangle$ of $\mathcal{L}_1$ is* pseudo-linearized to *a state $\langle \mathbb{V}_2, \mathtt{L}_2 \rangle$ of $\mathcal{L}_2$, written $\langle \mathbb{V}_1, \mathtt{L}_1 \rangle \ni_{\mathtt{pl}} \langle \mathbb{V}_2, \mathtt{L}_2 \rangle$, if there exists a surjection σ such that:*

1. *if $(x,y) \in \mathbb{V}_1$ then $(\sigma(x), \sigma(y)) \in \mathbb{V}_2$.*
2. *if $\flat(\mathtt{L}_1) = \mathtt{T}_1 + \cdots + \mathtt{T}_m$ and $\flat(\mathtt{L}_2) = \mathtt{T}'_1 + \cdots + \mathtt{T}'_n$, then for every $1 \leqslant i \leqslant m$, there exists $1 \leqslant j \leqslant n$, such that $\sigma(\mathtt{T}_i) \in \mathtt{T}'_j$;*

3. *if* $\mathtt{f}(\widetilde{x}) \in \mathtt{L}_1$ *then either (1)* $\mathtt{f}(\sigma(\widetilde{x}))$ *in* $\mathtt{L}_2$ *or (2) there is* $\mathtt{f}(\widetilde{y}_1 \cdots \widetilde{y}_k)$ *in* $\mathtt{L}_2$ *such that, for some* $1 \leqslant i \leqslant k$, $\sigma(\widetilde{x}) = \widetilde{y}_i$;

We use the same notational convention for contexts as in Lemma 2.

Lemma 4. *Let* $\langle \mathbb{V}_1, \mathtt{L}_1 \rangle \ni_{\mathtt{pl}} \langle \mathbb{V}_2, \mathtt{L}_2 \rangle$. *Then,* $\langle \mathbb{V}_1, \mathtt{L}_1 \rangle \longrightarrow \langle \mathbb{V}'_1, \mathtt{L}'_1 \rangle$ *implies there exists* $\langle \mathbb{V}_2, \mathtt{L}_2 \rangle \longrightarrow^+ \langle \mathbb{V}'_2, \mathtt{L}'_2 \rangle$ *such that* $\langle \mathbb{V}'_1, \mathtt{L}'_1 \rangle \ni_{\mathtt{pl}} \langle \mathbb{V}'_2, \mathtt{L}'_2 \rangle$

Proof. Base case. $\mathtt{L}_1$ is the main lam of the nonlinear program, and $\mathtt{L}_2$ its pseudolinear transformation.

$$\mathtt{L}_1 = \mathfrak{L}_1[\mathtt{f}_1(\widetilde{u}_1)] \cdots [\mathtt{f}_m(\widetilde{u}_k)],$$

where $\mathfrak{L}_1$ does not contain any other function invocations, and $m \leqslant k$, meaning that some of the $\mathtt{f}_i$, $1 \leqslant i \leqslant m$, can be invoked more than once on different parameters.

After the transformation, $\mathtt{L}_2$ contains the same pairs as $\mathtt{L}_1$ and the same function invocations, but with possibly more arguments:

$$\mathtt{L}_2 = \mathfrak{L}_1[\mathtt{f}_1(\widetilde{u}_1, \widetilde{z}_1)] \cdots [\mathtt{f}_m(\widetilde{u}_k, \widetilde{z}_k)].$$

Notice that some of the $\widetilde{z}_j$, $1 \leqslant j \leqslant k$, may be empty if the corresponding function has not been expanded during the transformation. Moreover $\mathbb{V}_1$ and $\mathbb{V}_2$ contains only the identity relations on the arguments, so we have $\mathbb{V}_1 \subseteq \mathbb{V}_2$. Therefore, all conditions of definition 12 are trivially verified.

Inductive case. We have

$$\mathtt{L}_1 = \mathfrak{L}_1[\mathtt{f}_1(\widetilde{u}_1)] \cdots [\mathtt{f}_m(\widetilde{u}_k)],$$

where $\mathfrak{L}_1$ does not contain any other function invocations, and $m \leqslant k$, meaning that some of the $\mathtt{f}_i$, $1 \leqslant i \leqslant m$, can be invoked more than once on different parameters.

We have

$$\mathtt{L}_2 = \mathfrak{L}_2[\mathtt{f}_1(\widetilde{u}_1, \widetilde{z}_1)] \cdots [\mathtt{f}_m(\widetilde{u}_k, \widetilde{z}_k)].$$

where $\mathfrak{L}_2$ may contain other function invocations, but by inductive hypothesis we know that Definition 12 is verified. In particular condition *3* guarantees that at least the invocations of $\mathtt{f}_1, \ldots, \mathtt{f}_m$, with suitable arguments, are in $\mathtt{L}_2$.

Now, let us consider the reduction

$$\langle \mathbb{V}_1, \mathtt{L}_1 \rangle \longrightarrow \langle \mathbb{V}'_1, \mathtt{L}'_1 \rangle.$$

Without loss of generality, we can assume the reduction step performed an invocation of function $\mathtt{f}_1(\widetilde{u}_1)$.

We have different cases:

1. the function's lam $\mathtt{L}_{\mathtt{f}_1}$ has not been modified by the transformation. In this case the result follows trivially.

2. the function's lam $\mathtt{L}_{\mathtt{f}_1}$ has been affected only in that some function invocations in it have an updated arity. Meaning that it was only trasformed by $\overset{\mathtt{g},l}{\Longmapsto}$, for some $\mathtt{g}$ and l, as a side effect of other function expansions. It follows that $\flat(\mathtt{L}_{\mathtt{f}_1}) = \flat(\mathtt{L}'_{\mathtt{f}_1})$, where $\mathtt{L}'_{\mathtt{f}_1}$ is the body of $\mathtt{f}_1$ after the transoformation has been applied. This satisfies condition *2* of Definition 12. Those function invocations that have not been modified satisfy trivially the condition *3* of Definition 12. Regarding the other function invocations we have, by construction, that if $\mathtt{g}(\widetilde{x}) \in \mathtt{L}_{\mathtt{f}_1}$ then $\mathtt{g}(\widetilde{x}, \widetilde{y}) \in \mathtt{L}'_{\mathtt{f}_1}$, where $\widetilde{y}$ are fresh names. This satisfies condition *3* of Definition 12, as well. As for condition *1*, we have

$$\mathbb{V}'_1 = \mathbb{V}_1 \oplus (\widetilde{u}_1 < \widetilde{w}_1),$$

where $\widetilde{w}$ are fresh names created in $\mathtt{L}_{\mathtt{f}_1}$, and

$$\mathbb{V}'_2 = \mathbb{V}_2 \oplus (\widetilde{u}_1, \widetilde{z}_1 < \widetilde{w}_1, \widetilde{y}_1, \cdots, \widetilde{y}_s),$$

where $\widetilde{y}_1, \cdots, \widetilde{y}_s$ are the fresh names augmenting the function arities within $\mathtt{L}'_{\mathtt{f}_1}$. We choose the same fresh names $\widetilde{w}_1$ and condition *1* is satisfied.
3. the function's lam $\mathtt{L}_{\mathtt{f}_1}$ has been subject of the expansion of a function. Let

$$\mathtt{L}_{\mathtt{f}_1} = \mathfrak{L}_{\mathtt{f}_1}[\mathtt{g}_1(\widetilde{v}_1)] \cdots [\mathtt{g}_h(\widetilde{v}_n)],$$

where $\mathfrak{L}_{\mathtt{f}_1}$ contains only pairs, then, assuming without loss of generality that $\mathtt{g}_1$ was expanded:

$$\mathtt{L}'_{\mathtt{f}_1} = \mathfrak{L}_{\mathtt{f}_1}[\mathtt{g}_1(\widetilde{v}_1, \widetilde{z}_1^1, \ldots, \widetilde{z}_r^1)] \cdots [\mathtt{g}_1(\widetilde{v}_n, \widetilde{z}_1^r, \ldots, \widetilde{z}_r^r)],$$

where r is obtained by subtracting from the number of invocations n the number of occurrences of invocations of $\mathtt{g}_1$ in $\mathtt{L}'_{\mathtt{f}_1}$.
Now, the psedulinear program has to perform the r invocations of $\mathtt{g}_1$ that were not present in the original program, since they have been replaced r invocations of $\mathtt{g}_2 \cdots \mathtt{g}_h$, in order to reveal the actual invocations $\mathtt{g}_2 \cdots \mathtt{g}_h$ that has been delegated to $\mathtt{g}_1$ body. By construction, the arguments of the invocations where preserved by the transformation, so that if $\mathtt{g}_2(\widetilde{x})$ is produced by reduction of the nonlinear program, then the pseudolinear program will produce $\mathtt{g}_2(\widetilde{x}, \widetilde{y})$, with $\widetilde{y}$ fresh and possibily empty. This satisfy condition *3* of Definition 12.
However the body of $\mathtt{g}_1$ may have been transformed in a similar way by expanding another method, let us say $\mathtt{g}_2$. Then all the invocations of $\mathtt{g}_2$ in $\mathtt{g}_1$'s body that corresponds to the previously delegated function invocations $\mathtt{g}_2 \cdots \mathtt{g}_h$ have to be invoked as well. This procedure has to be iterated until all the corresponding invocations are encountered. Each step of reduction will produce spurious pairs and function invocations, but all of these will be on different new names.

Lemma 5. *Let* $\langle \mathbb{V}_1, \mathtt{L}_1 \rangle \ni_{\mathtt{pl}} \langle \mathbb{V}_2, \mathtt{L}_2 \rangle$ *and* $\langle \mathbb{V}_1, \mathtt{L}_1 \rangle \longrightarrow^* \langle \mathbb{V}'_1, \mathtt{L}'_1 \rangle$. *Then there are* $\langle \mathbb{V}'_1, \mathtt{L}'_1 \rangle \longrightarrow^* \langle \mathbb{V}''_1, \mathtt{L}''_1 \rangle$ *and* $\langle \mathbb{V}_2, \mathtt{L}_2 \rangle \longrightarrow^* \langle \mathbb{V}'_2, \mathtt{L}'_2 \rangle$ *such that* $\langle \mathbb{V}''_1, \mathtt{L}''_1 \rangle \ni_{\mathtt{pl}} \langle \mathbb{V}'_2, \mathtt{L}'_2 \rangle$

Proof. A straightforward induction on the length of $\langle \mathbb{V}_1, \mathtt{L}_1 \rangle \longrightarrow^* \langle \mathbb{V}'_1, \mathtt{L}'_1 \rangle$.

Every preliminary result is in place for the correctness of the transformation $\overset{\mathtt{npl} \mapsto \mathtt{pl}}{\Longmapsto}$.

Theorem 5. *Let $\mathcal{L}_1$ be a non-pseudo-linear program and $\mathcal{L}_2$ be the result of the transformations in Table 2. If $\mathcal{L}_2$ is circularity-free then $\mathcal{L}_1$ is circularity-free.*

Proof. By Lemma 4, for every $\langle \mathbb{V}_1, \mathtt{L}_1 \rangle$ reached by evaluating $\mathcal{L}_1$, there is $\langle \mathbb{V}_2, \mathtt{L}_2 \rangle$ that is reached by evaluating $\mathcal{L}_2$ such that $\langle \mathbb{V}_1, \mathtt{L}_1 \rangle \ni_{\mathtt{pl}} \langle \mathbb{V}_2, \mathtt{L}_2 \rangle$. This guarantees that every circularity in $\langle \mathbb{V}_1, \mathtt{L}_1 \rangle$ is also present in $\langle \mathbb{V}_2, \mathtt{L}_2 \rangle$. We conclude by Lemma 5.