New Lower Bounds for Reachability in Vector Addition Systems

Wojciech Czerwiński, Ismaël Jecker, Slawomir Lasota, Jérôme Leroux, Łukasz Orlikowski

New Lower Bounds for Reachability in Vector Addition Systems

Wojciech Czerwiński
University of Warsaw

Ismaël Jecker
University of Warsaw

Sławomir Lasota
University of Warsaw, Poland

Jérôme Leroux
LaBRI, CNRS, Univ. Bordeaux, France

Łukasz Orlikowski
University of Warsaw

Abstract

We investigate the dimension-parametric complexity of the reachability problem in vector addition systems with states (VASS) and its extension with pushdown stack (pushdown VASS). Up to now, the problem is known to be $\mathcal{F}_d$-hard for VASS of dimension $3d + 2$ (the complexity class $\mathcal{F}_d$ corresponds to the kth level of the fast-growing hierarchy), and no essentially better bound is known for pushdown VASS. We provide a new construction that improves the lower bound for VASS: $\mathcal{F}_k$-hardness in dimension $2d + 3$. Furthermore, building on our new insights we show a new lower bound for pushdown VASS: $\mathcal{F}_k$-hardness in dimension $\frac{d}{2} + 6$. This dimension-parametric lower bound is strictly stronger than the upper bound for VASS, which suggests that the (still unknown) complexity of the reachability problem in pushdown VASS is higher than in plain VASS (where it is Ackermann-complete).

2012 ACM Subject Classification Theory of computation → Concurrency; Theory of computation → Verification by model checking; Theory of computation → Logic and verification

Keywords and phrases vector addition systems, reachability problem, pushdown vector addition system, lower bounds

Digital Object Identifier 10.4230/LIPIcs.STACS.2022.35

Funding *Wojciech Czerwiński*: Supported by the ERC grant INFSYS, agreement no. 950398.
Sławomir Lasota: Supported by the ERC project 'Lipa' within the EU Horizon 2020 research and innovation programme (No. 683080) and by the NCN grant 2021/41/B/ST6/00535.
Jérôme Leroux: Supported by the grant ANR-17-CE40-0028 of the French National Research Agency ANR (project BRAVAS).

1 Introduction

Petri nets, equivalently presentable as vector addition systems with states (VASS), are an established model of concurrency with widespread applications. The central algorithmic problem for this model is the *reachability problem* which asks whether from a given initial configuration there exists a sequence of valid execution steps reaching a given final configuration. For a long time the complexity of this problem remained one of the hardest open questions in verification of concurrent systems. In 2019 Leroux and Schmitz made a significant breakthrough by providing an Ackermannian upper bound [14]. With respect to the hardness, the exponential space lower bound, shown by Lipton already in 1976 [16], remained the only known for over 40 years until a breakthrough non-elementary lower bound by Czerwiński, Lasota, Lazic, Leroux and Mazowiecki in 2019 [3, 4]. Finally, a matching Ackermannian lower bound announced in 2021 independently by two teams, namely Czerwiński and Orlikowski [5] and Leroux [12], established the exact complexity of the problem.

However, despite the fact that the exact complexity of the reachability problem for VASS is settled, there are still significant gaps in our understanding of the problem. One such gap is the complexity of the reachability problem *parametrised by the dimension*, namely deciding the reachability problem for d-dimensional VASS (d-VASS) for fixed $d \in \mathbb{N}$. Currently, the exact complexity bounds are only known for dimensions one and two. In these cases, the complexity depends on the representations of numbers in the transitions, either unary or binary For binary VASS (where the numbers are represented in binary) the reachability problem is known to be NP-complete for 1-VASS [9] and PSpace-complete for 2-VASS [2]. For unary VASS the problem is NL-complete for both 1-VASS (folklore) and 2-VASS [8].

Much less is known for higher dimensions, and it is striking that even in the case of 3-VASS we have a huge complexity gap. The best complexity upper bound comes from the above mentioned work of Leroux and Schmitz [14], where it is proved that the reachability problem for $(d-4)$-VASS is in $\mathcal{F}_d$ (here $\mathcal{F}_d$ denotes the dth level of the Grzegorczyk hierarchy of complexity classes, which corresponds to the fast growing function hierarchy). In particular this shows that the reachability problem for 3-VASS is in $\mathcal{F}_7$ (recall that $\mathcal{F}_3 = \text{Tower}$).

The recent Ackermann-hardness results provide lower bounds for the reachability problem in fixed dimensions. The result of Czerwiński and Orlikowski [5] yields $\mathcal{F}_d$-hardness for $6d$-VASS, while the result of Leroux [12] establishes $\mathcal{F}_d$-hardness for $(4d+5)$-VASS. Lasota improved upon these results and showed $\mathcal{F}_d$-hardness of the problem for $(3d+2)$-VASS [10]. In [6], additional lower bound results were obtained for specific fixed dimensions: PSpace-hardness for unary 5-VASS, ExpSpace-hardness for binary 6-VASS and Tower-hardness for unary 8-VASS.

To summarise, despite significant research efforts there are still several natural problems related to the VASS reachability problem that present significant complexity gaps:

$\mathbf{Q_1}$: What is the complexity of the reachability problem for VASS of dimension 3? It is known to be PSpace-hard and in $\mathcal{F}_7$;

$\mathbf{Q_2}$: What is highest dimension for which the complexity of the reachability problem is elementary? It is known to fall within the range of 2 to 8;

$\mathbf{Q_3}$: What is the smallest constant C such that the complexity of the reachability problem for d-VASS is in $\mathcal{F}_{Cd+o(d)}$? It is known to fall within the range of $\frac{1}{3}$ to 1.

In this work, we focus on addressing Question $\mathbf{Q}_3$. We present new and improved lower bounds, first in the standard setting of VASS, and then in the setting of pushdown VASS (PVASS) which extend the VASS model by incorporating a pushdown stack.

VASS reachability. Our first main result is a new complexity lower bound for the reachability problem which improves the gap:

▶ **Theorem 1.** *The reachability problem for $(2d+3)$-VASS is $\mathcal{F}_d$-hard.*

A preliminary version of this result was presented in in [11]. In this revised version, we aim to present the result in a conceptually simple framework by using the notion of *triples*, a formalism that was originally developed in [3], and was heavily used in [5] and [10]. We view this contribution as an important step towards understanding the complexity of the VASS reachability problem parametrised by the dimension.

PVASS reachability. The decidability of the reachability problem for PVASS has been an important open problem for over a decade [1]. Despite efforts of the community, it remains unknown even for PVASS of dimension 1 (1-PVASS), namely automata with one counter and one pushdown stack.

▶ **Conjecture 2.1.** *The reachability problem for PVASS is decidable.*

It is important to acknowledge that the PVASS setting is complex, and few decidability results are known. Some progress has been made in the study of the *coverability* problem, a variant of the reachability problem which asks whether from a given initial configuration there exists a sequence of valid execution steps that reaches a configuration *greater* than the given final configuration. Notably, the coverability problem has been shown to be decidable for 1-PVASS [15] (and mentioned to be in ExpSpace).

Interestingly, despite the slow progress on determining upper bounds for PVASS, there is limited knownledge about lower bounds as well. To the best of our knowledge, the only lower bound that is not directly implied by the results on VASS concerns (again) the coverability problem for 1-PVASS, which has been established as PSpace-hard [7].

As for our contribution, our second main result is the first complexity lower bound for the PVASS reachability problem that is not immediately inherited from VASS:

▶ **Theorem 2.** *The reachability problem for $(\lfloor \frac{d}{2} \rfloor + 6)$-PVASS is $\mathcal{F}_d$-hard.*

Notably, Theorem 2 implies that for sufficiently large d the reachability problem for d-PVASS is harder than the problem for $(d+1)$-VASS (which is a subclass of d-PVASS as the pushdown stack can keep track of one VASS counter). Indeed the problem for d-PVASS is $\mathcal{F}_{2d-12}$-hard by Theorem 2, while the problem for $(d+1)$-VASS is in $\mathcal{F}_{d+5}$ by [14].

While our results indicate that the reachability problem for PVASS is harder than for VASS, some known results about PVASS hint that even higher lower bounds might be proved: In [13] it was shown that PVASS are able to weakly compute functions of hyper-Ackermannian growth rate. Based on this observation we propose the following two conjectures:

▶ **Conjecture 2.2.** *There exists a fixed dimension $d \in \mathbb{N}$ such that the reachability problem for d-PVASS is Ackermann-hard.*

▶ **Conjecture 2.3.** *The reachability problem for PVASS is hyperAckermann-hard.*

2 Preliminaries

Fast-growing hierarchy. Let $\mathbb{N}_+ = \mathbb{N} \setminus \{0\}$ be the set of positive integers. We define the complexity classes $\mathcal{F}_i$ corresponding to the ith level in the Grzegorczyk Hierarchy [18,

Sect. 2.3, 4.1]. To this aim we choose to use the following family of functions $\mathbf{F}_i : \mathbb{N}_+ \to \mathbb{N}_+$, indexed by $i \in \mathbb{N}$:

$$\mathbf{F}_0(n) = n + 2, \qquad \mathbf{F}_{i+1} = \widetilde{\mathbf{F}_i} \quad \text{where} \quad \widetilde{F}(n) = F^{n-1}(1) = \underbrace{F \circ F \circ \ldots \circ F}_{n-1}(1). \tag{1}$$

In particular, $\mathbf{F}_1(n) = 2n - 1$ and $\mathbf{F}_i(1) = 1$ for all $i \in \mathbb{N}_+$. Using the functions $\mathbf{F}_i$, we define the complexity classes $\mathcal{F}_i$, indexed by $i \in \mathbb{N}_+$, of problems solvable in deterministic time $\mathbf{F}_i \circ \mathbf{F}_{i-1}^m(n)$ for some $m \in \mathbb{N}$:

$$\mathcal{F}_i = \bigcup_{m \in \mathbb{N}} \text{DTime}(\mathbf{F}_i \circ \mathbf{F}_{i-1}^m(n)).$$

Intuitively speaking, the class $\mathcal{F}_i$ contains all problems solvable in time $\mathbf{F}_i(n)$, and is closed under reductions computable in time of lower order $\mathbf{F}_{i-1}^m(n)$, for some fixed $m \in \mathbb{N}_+$. In particular, $\mathcal{F}_3 = \text{Tower}$ (problems solvable in a tower of exponentials of time or space whose height is an elementary function of input size). The classes $\mathcal{F}_i$ are robust with respect to the choice of fast-growing function hierarchy (see [18, Sect.4.1]). For $i \geq 3$, instead of deterministic time, one could equivalently take nondeterministic time, or space as all these definitions collapse.

2.1 Counter programs

A *counter program* (or simply a *program*) is a sequence of (line-numbered) commands, each of which is of one of the following types:

x += 1	(increment the counter x by one)
x −= 1	(decrement the counter x by one)
goto L **or** L'	(nondeterministically jump to either line L or line L')
zero? x	(zero test: continue if counter x equals 0)

Counter programs *with a pushdown stack* (or simply *programs with stack*) are enhanced versions of plain counter programs that incorporate a stack containing a word over a fixed stack alphabet **S**. The stack content is modified by using two additional types of commands:

PUSH(s)	(push the symbol $s \in$ S at the top of the stack)
POP(s)	(pop the symbol $s \in$ S if it is at the top of the stack)

The command POP(s) fails if the stack is empty or if the top symbol is different from s. A *configuration* of a counter program with pushdown stack consists, as expected, of a valuation of its counters plus a stack content.

Counters are only allowed to have nonnegative values.

Conventions:. We are particularly interested in counter programs *without zero tests*, i.e., ones that use no zero test command. In the sequel, unless specified explicitly, counter programs are implicitly assumed to be without zero tests. Moreover, we use the syntactic sugar **loop**, which iterates a sequence of command a nondeterministic number of times (see Example 3). Finally, we write consecutive increments and decrements of different variables on a single line and we use the following shorthands:

x += m	(increment the counter x by m)
x −= m	(decrement the counter x by m)
x ⟶ y	(decrement the counter x by one and increment the counter y by one)
x $\xrightarrow{\text{m}}$ y	(decrement the counter x by m and increment the counter y by m)

▶ **Example 3.** As an illustration, consider three different presentations of the same the program with three counters $\mathsf{C} = \{\mathsf{x}, \mathsf{y}, \mathsf{z}\}$:

```
1: goto 2 or 6          1: loop              1: loop  x ⟶ y   z += 2
2: x −= 1               2:    x −= 1         2: z += 1
3: y += 1               3:    y += 1
4: z += 2               4:    z += 2
5: goto 1 or 1          5: z += 1
6: z += 1
```

The program repeats the block of commands in lines 2–4 some number of times chosen nondeterministically (possibly zero, although not infinite because x is decreasing, and hence its initial value bounds the number of iterations) and then increments z.

We emphasise that counters are only permitted to have nonnegative values. In the program above, that is why the decrement in line 2 works also as a non-zero test.

Runs. Consider a program with counters X. By $\mathbb{N}^{\mathsf{X}}$ we denote the set of all valuations of counters. Given an initial valuation of counters, a *run* (or *execution*) of a counter program is a finite sequence of executions of commands, as expected. A run which has successfully finished we call *complete*; otherwise, the run is *partial*. Observe that, due to a decrement that would cause a counter to become negative, a partial run may fail to continue because it is blocked from further execution. Moreover, due to nondeterminism of **goto**, a program may have various runs from the same initial valuation.

Two programs $\mathcal{P}, \mathcal{Q}$ may be *composed* by concatenating them, written $\mathcal{P}\ \mathcal{Q}$. We silently assume the appropriate re-numbering of lines referred to by **goto** command in $\mathcal{Q}$.

Consider a distinguished set of counters $\mathsf{Z} \subseteq \mathsf{X}$. A run of $\mathcal{P}$ is Z-*zeroing* if it is complete and all counters from Z are zero at the end. Given two counter valuations $r, r' \in \mathbb{N}^{\mathsf{X}}$ we say that the program Z-*computes* the valuation r from r' if $\mathcal{P}$ has exactly one Z-zeroing run from r' and the end configuration is r. We also say that the program Z-*computes nothing* from r' if $\mathcal{P}$ has no Z-zeroing run from r'. For instance, in Example 3 the programs $\{\mathsf{x}\}$-compute, from any valuation satisfying $\mathsf{x} = \mathsf{n} \in \mathbb{N}$ and $\mathsf{y} = \mathsf{z} = 0$, the valuation satisfying $\mathsf{x} = 0$ (trivially), $\mathsf{y} = \mathsf{n}$ and $\mathsf{z} = 2\mathsf{n} + 1$.

Maximal iteration. The proofs of this paper often focus on the number of iterations of the **loop** construct. Consider a program $\mathcal{P}$ containing a *flat* **loop**, i.e., a loop whose body consists of only increment or decrement commands, such that each counter appears in at most one of these commands (like the programs in Example 3). We say that this loop is *maximally iterated* in a given run of a $\mathcal{P}$ if some counter that is decremented in its body is zero at the exit from the loop. In particular, a maximally iterated loop could not be iterated any further without violation of the nonnegativity constraint on the decremented counter. For instance, the loop in Example 3 is maximally iterated by the $\{\mathsf{x}\}$-zeroing runs. Needless to say, maximal iteration needs not happen in general, for instance the program in Example 3 has multiple complete runs that do not admit this property.

3 Main results and structure of the paper

Counter programs (without zero test) provide an equivalent presentation to the standard models of Petri nets and VASS, and the transformations between these different models are straightforward. For instance, a program can be transformed into a VASS by taking one

state for each line of the program, and adding an appropriate transition corresponding to each counter update instruction. Note that the dimension of the VASS obtained is equal to the number of counters of the program. In this paper, we focus solely on counter programs, and we prove that the following problem is $\mathcal{F}_d$-hard for every $d \geq 3$:

▶ **Problem 1.** ***Input:*** *A program $\mathcal{P}$ using $2d+3$ counters.*
Question: *Is there a complete run of $\mathcal{P}$ that starts and ends with all counters equal to 0?*

The equivalence between programs and VASS then directly leads to our first main result:

▶ **Theorem 1.** *The reachability problem for $(2d+3)$-VASS is $\mathcal{F}_d$-hard.*

For programs with a pushdown stack, $\mathcal{F}_d$-hardness can be achieved with less counters. We show that the following problem is $\mathcal{F}_d$-hard for every $d \geq 3$:

▶ **Problem 2.** ***Input:*** *A program with stack $\mathcal{Q}$ using $\lfloor \frac{d}{2} \rfloor + 6$ counters.*
Question: *Is there a complete run of $\mathcal{Q}$ that starts and ends with all counters equal to 0?*

Again, the equivalence between programs and VASS yields our second main result:

▶ **Theorem 2.** *The reachability problem for $(\lfloor \frac{d}{2} \rfloor + 6)$-PVASS is $\mathcal{F}_d$-hard.*

Let us now introduce the known $\mathcal{F}_d$-hard problem that we will reduce to Problems 1 and 2. Fortunately, we do not have to search too far for it: counter programs with two counters *and zero tests* are Turing-complete [17]. This implies that the reachability problem is undecidable in that setting. However, similarly to Turing machines, decidability is recovered by imposing limitations on the executions, such as bounding their length, the maximal counter size, or the number of zero tests. For our purposes the latter is the easiest to use, thus, we present the problem that we will reduce from, a variation of the "$\mathbf{F}_k$-bounded Minsky Machine Halting Problem" proved to be $\mathcal{F}_d$-complete in [18, Section 2.3.2]:

▶ **Problem 3.** ***Input:*** *A program $\mathcal{P}$ with two zero-tested counters, and a bound $n \in \mathbb{N}$.*
Question: *Is there a complete run of $\mathcal{P}$ that starts and ends with all counters equal to 0 and does exactly $\mathbf{F}_d(n)$ zero tests?*

The rest of this section is devoted to the presentation of the tools we use to reduce Problem 3 to Problems 1 and 2. Following the structure of similar reductions presented in [5] and [10], our reduction is divided into two main steps.

In the first step, we show how a program *without zero test* can simulate a bounded number of zero tests. To achieve this we rely on the concept of *triples*, which are specific counter valuations that allow to eliminate the zero tests and instead verify whether a particular invariant still holds at the term of the run. However, doing so requires an initial triple directly proportional to the number of zero tests we aim to simulate. Since we intend to simulate $\mathbf{F}_d(n)$ zero tests, which is a rather large number, directly applying this approach would result in an excessively large program.

Thus, in the second step, we construct compact *amplifiers*. These amplifiers are small programs that compute functions of substantial magnitude (such as $\mathbf{F}_d$) while using a small number of counters (namely $2d+4$, or $\lfloor \frac{d}{2} \rfloor + 4$ counters along with a stack).

We now define formally the notions required for these two steps. This will allow us to state the main lemmas proved in this paper, and use them to construct the reduction proving our main result. The proofs of the lemmas can then be found in the following sections.

Triples. The concept of *triples* plays a central role in all the constructions presented within this paper. Given a set of counters X, three distinguished counters $\mathsf{a}, \mathsf{b}, \mathsf{c} \in \mathsf{X}$ and $A, B \in \mathbb{N}$, we denote by $\textsc{Triple}(A, B, \mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{X})$ the counter valuation satisfying

$$\mathsf{a} = A, \quad \mathsf{b} = B, \quad \mathsf{c} = A \cdot (4^{\mathsf{B}} - 1), \quad \mathsf{x} = 0 \text{ for every } \mathsf{x} \in \mathsf{X} \setminus \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}. \tag{2}$$

Informally we sometimes call such valuation a *B-triple*, or simply a *triple* over the counters $\mathsf{a}, \mathsf{b}, \mathsf{c}$. The interest of triples lies in their ability to establish invariants that enable the detection of unwanted behaviours in counter programs. For instance, in Section 4, we show how to use triples to replace zero tests by proving the following lemma:

▶ **Lemma 4.** *Let $\mathcal{P}$ be a program using two zero-tested counters. There exists a program $\mathcal{P}'$ with six counters X such that for every $B \in \mathbb{N}_+$ the two following conditions are equivalent:*

- *There exists a complete run of $\mathcal{P}$ that starts and ends with all counters equal to 0 and performs exactly B zero tests;*
- *There exists a complete run of $\mathcal{P}'$ that starts in some configuration $\textsc{Triple}(A, B, \mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{X})$ with $A \in \mathbb{N}_+$ and ends in a configuration where all the counters except a are zero.*

Amplifiers. The key contribution of this paper consists in the construction of two families of programs that transform B-triples into $\mathbf{F}_d(B)$-triples. We formalise this type of programs through the notion of *amplifiers*. Let $F : \mathbb{N}_+ \to \mathbb{N}_+$ be a monotone function satisfying $F(n) \geq n$ for every $n \in \mathbb{N}_+$. Consider a program $\mathcal{P}$ using the set of counters X, out of which we distinguish six counters $\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{b}', \mathsf{c}', \mathsf{t} \in \mathsf{X}$, and a subset of counters $\mathsf{Z} \subseteq \mathsf{X} \setminus \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{b}', \mathsf{t}\}$ that contains c'. The tuple $(\mathcal{P}, (\mathsf{a}, \mathsf{b}, \mathsf{c}), (\mathsf{a}, \mathsf{b}', \mathsf{c}'), \mathsf{t}, \mathsf{Z})$ is called *F-amplifier* if for all $A, B \in \mathbb{N}_+$:

- $\mathcal{P}$ Z-computes $\textsc{Triple}(A \cdot 4^{B-F(B)}, F(B), \mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{X})$ from $\textsc{Triple}(A, B, \mathsf{a}, \mathsf{b}', \mathsf{c}', \mathsf{X})$ if A is divisible by $4^{(F(B)-B)}$;
- $\mathcal{P}$ Z-computes nothing from $\textsc{Triple}(A, B, \mathsf{a}, \mathsf{b}', \mathsf{c}', \mathsf{X})$ if A is not divisible by $4^{(F(B)-B)}$.

An amplifier transforms B-triples on its *input counters* $\mathsf{a}, \mathsf{b}', \mathsf{c}'$ into $F(B)$-triples on its *output counters* $\mathsf{a}, \mathsf{b}, \mathsf{c}$. Remark that the counter a is involved in both input and output. The counters in Z, called *end counters*, are intuitively speaking assumed to be 0-checked after the completion of a run of $\mathcal{P}$. The auxiliary counter t does not play a direct role apart from *not* being an input counter, an output counter nor an end counter. This will prove useful in our constructions. We note that no condition is imposed on the runs that start from a counter valuation that is not a triple on the input counters, nor on the runs that are not Z-zeroing.

In Section 5 we construct of a family of $\mathbf{F}_d$-amplifiers:

▶ **Lemma 5.** *For every $d \in \mathbb{N}_+$ there exists an $\mathbf{F}_d$-amplifier of size $\mathcal{O}(d)$ that uses $2d + 4$ counters out of which d are end counters.*

Furthermore, in Section 6 we extend the notion of amplifiers to programs with stack, and we demonstrate how using a stack in an efficient manner can replace three quarters of the counters used in the previous construction:

▶ **Lemma 6.** *For every $d \in \mathbb{N}_+$ there exists an $\mathbf{F}_d$-amplifier of size $\mathcal{O}(d)$ that uses a stack and $\lfloor \frac{d}{2} \rfloor + 4$ counters out of which $\lfloor \frac{d}{2} \rfloor$ are end counters.*

Proof of the main theorems. While the proofs of our key lemmas are delegated to the appropriate sections, we can already show how these lemmas yield a reduction from Problem 3 to Problems 1 and 2. Let us consider an instance of Problem 3, that is, a 2-counter program

with zero tests $\mathcal{P}$ and an integer $d \in \mathbb{N}$. We transform this instance into an instance $\mathcal{P}''$ of Problem 1 and an instance $\mathcal{Q}''$ of Problems 2. These two programs rely on the program $\mathcal{P}'$ given by Lemma 4, and the $\mathbf{F}_d$-amplifiers $\mathcal{P}_d$ and $\mathcal{Q}_d$ given by Lemmas 5 and 6.

Program $\mathcal{P}''$:

1: $\mathsf{b}' \mathrel{+}= n$
2: **loop** $\mathsf{a} \mathrel{+}= 1 \quad \mathsf{c}' \mathrel{+}= 4^n - 1$
3: $\mathcal{P}_d$
4: $\mathcal{P}'$
5: **loop** $\mathsf{a} \mathrel{-}= 1$

Program $\mathcal{Q}''$:

1: $\mathsf{b}' \mathrel{+}= n$
2: **loop** $\mathsf{a} \mathrel{+}= 1 \quad \mathsf{c}' \mathrel{+}= 4^n - 1$
3: $\mathcal{Q}_d$
4: $\mathcal{P}'$
5: **loop** $\mathsf{a} \mathrel{-}= 1$

We now prove that this is a valid reduction from Problem 3 to Problem 1. The proof for Problem 2 is analogous since the programs $\mathcal{P}_d$ and $\mathcal{Q}_d$ have identical effect on triples.

We need to show that $\mathcal{P}''$ has a complete run that starts and ends with all counters equal to 0 if and only if $\mathcal{P}$ has a complete run that starts and ends with all counters equal to 0 and that does exactly $\mathbf{F}_d(n)$ zero tests. In order to prove it, let us consider the structure of a hypothetical run π of $\mathcal{P}''$ that starts and ends with all counters having value zero. Starting from the configuration where all the counters are zero, Lines 1–2 generate an arbitrary n-triple: Progressing through line 1 and performing $A \in \mathbb{N}_+$ iterations of line 2 results in the configuration $\textsc{Triple}(A, n, \mathsf{a}, \mathsf{b}', \mathsf{c}', \mathsf{X})$. Next, it is important to note that the subrun of π involving $\mathcal{P}_d$ zeroes all the end counters, since these counters remain unchanged after the invocation of $\mathcal{P}_d$ and they have value zero at the end of π. Consequently, according to the definition of an amplifier, $\mathcal{P}_d$ transforms the n-triple $\textsc{Triple}(A, n, \mathsf{a}, \mathsf{b}', \mathsf{c}', \mathsf{X})$ into an $\mathbf{F}_d(n)$-triple $\textsc{Triple}(A', \mathbf{F}_d(n), \mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{X})$. From there, the subrun involving $\mathcal{P}'$ must end in a configuration where every counter except a has value zero since the final line of $\mathcal{P}''$ can only decrement a. Therefore, the run π exists if and only if there exists a run of $\mathcal{P}'$ that bridges the gap, starting from an $\mathbf{F}_d(n)$-triple $\textsc{Triple}(A', \mathbf{F}_d(n), \mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{X})$ and ending in a configuration where all the counters except a are 0. By Lemma 4 we know that such a run of $\mathcal{P}'$ exists if and only if $\mathcal{P}$ has a complete run that starts and ends with all counters equal to 0 and does exactly $\mathbf{F}_d(n)$ zero tests, which shows that our reduction is valid.

To conclude, let us remark that, as defined here, the program $\mathcal{P}''$ uses $2d + 7$ counters: the call to $\mathcal{P}_d$ requires $2d + 4$ counters, and while $\mathcal{P}'$ shares the output counters a, b and c of $\mathcal{P}_d$, it uses three more counters. We now argue that four counters can be saved, so that our program matches the definition of Problem 1.

- The value of the input counter b' is never incremented in the program $\mathcal{P}_d$ we construct.[1] Therefore, since in $\mathcal{P}''$ the call to $\mathcal{P}_d$ *always* starts with the value $\mathsf{b}' = n$, we can get rid of the counter b' by replacing the call to $\mathcal{P}_d$ with n consecutive copies of $\mathcal{P}_d$ in which each instruction decrementing b' is replaced with a jump to the next copy.
- The second optimisation consists in reusing the counters of $\mathcal{P}_d$. Since $\mathcal{P}_d$ is an amplifier, at the term of the run it is sufficient to check that the end counters are 0 to ensure that *all* the counters except the output counters a and c are 0. Therefore, while the call to $\mathcal{P}'$ in $\mathcal{P}''$ needs to keep the d end counters of $\mathcal{P}_d$ untouched, there are still d freely reusable counters (not counting a, b and c that are already reused), and we can pick any three of these to use in $\mathcal{P}'$ instead of adding fresh ones.

For the program $\mathcal{Q}''$ it is not possible to save counters is that way, but one of the extra counters of $\mathcal{P}'$ can be loaded on the stack, which results in a program with $2d + 6$ counters.

[1] Remark that this is *not* stated explicitly by Lemma 5, but it is a trivial property of the corresponding construction presented in Section 5.

4 Triples as a replacement for zero tests

Let $\mathcal{P}$ be a program with zero tests using two counters x and y. The goal of this subsection is to construct a program that simulates $\mathcal{P}$ *without using zero tests.*

▶ **Lemma 4.** *Let $\mathcal{P}$ be a program using two zero-tested counters. There exists a program $\mathcal{P}'$ with six counters X such that for every $B \in \mathbb{N}_+$ the two following conditions are equivalent:*

- *There exists a complete run of $\mathcal{P}$ that starts and ends with all counters equal to 0 and performs exactly B zero tests;*
- *There exists a complete run of $\mathcal{P}'$ that starts in some configuration $\textsc{Triple}(A, B, \mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{X})$ with $A \in \mathbb{N}_+$ and ends in a configuration where all the counters except a are zero.*

The six counters of program $\mathcal{P}'$ will consists in two counters x, y simulating the two counters of $\mathcal{P}$, three counters $\mathsf{a}, \mathsf{b}, \mathsf{c}$ containing the initial triple, and an auxiliary counter t. The idea behind the construction is that we will replace the zero tests with the two gadgets $\mathcal{Z}ero(\mathsf{x})$ and $\mathcal{Z}ero(\mathsf{y})$ defined as follows:

Program $\mathcal{Z}ero(\mathsf{x})$:

1: **loop** $\mathsf{a} \longrightarrow \mathsf{t}$ $\mathsf{c} \longrightarrow \mathsf{t}$
2: **loop** $\mathsf{y} \longrightarrow \mathsf{x}$ $\mathsf{c} \longrightarrow \mathsf{t}$
3: **loop** $\mathsf{t} \longrightarrow \mathsf{a}$ $\mathsf{c} \longrightarrow \mathsf{a}$
4: **loop** $\mathsf{x} \longrightarrow \mathsf{y}$ $\mathsf{c} \longrightarrow \mathsf{a}$
5: $\mathsf{b} \mathrel{-}= 1$

Program $\mathcal{Z}ero(\mathsf{y})$:

1: **loop** $\mathsf{a} \longrightarrow \mathsf{t}$ $\mathsf{c} \longrightarrow \mathsf{t}$
2: **loop** $\mathsf{x} \longrightarrow \mathsf{y}$ $\mathsf{c} \longrightarrow \mathsf{t}$
3: **loop** $\mathsf{t} \longrightarrow \mathsf{a}$ $\mathsf{c} \longrightarrow \mathsf{a}$
4: **loop** $\mathsf{y} \longrightarrow \mathsf{x}$ $\mathsf{c} \longrightarrow \mathsf{a}$
5: $\mathsf{b} \mathrel{-}= 1$

The functioning of these two programs revolves around the following invariant:

$$\textsc{Invariant:} \qquad (\mathsf{a}+\mathsf{x}+\mathsf{y}+\mathsf{t}) \cdot 4^{\mathsf{b}} = \mathsf{a}+\mathsf{x}+\mathsf{y}+\mathsf{t}+\mathsf{c} \quad \text{and} \quad \mathsf{t} = 0;$$

$$\textsc{Broken invariant:} \qquad (\mathsf{a}+\mathsf{x}+\mathsf{y}+\mathsf{t}) \cdot 4^{\mathsf{b}} < \mathsf{a}+\mathsf{x}+\mathsf{y}+\mathsf{t}+\mathsf{c}.$$

Remark that the broken invariant is more specific than the negation of the invariant. We now present a technical claim showing that $\mathcal{Z}ero(\mathsf{x})$ and $\mathcal{Z}ero(\mathsf{y})$ accurately replace zero tests.

▶ **Claim 6.1.** *Let $\mathsf{z} \in \{\mathsf{x}, \mathsf{y}\}$. From each configuration of $\mathcal{Z}ero(\mathsf{z})$ where $\mathsf{b} > 0$, $\mathsf{z} = 0$ and the invariant holds there is a unique complete run that maintains the invariant and preserves the values of x and y. All other runs starting from this configuration, as well as all runs starting with a broken invariant or a holding invariant with a value of z greater than 0, end with a broken invariant.*

Proof. We show the result for $\mathcal{Z}ero(\mathsf{x})$: the proof can easily be transferred to $\mathcal{Z}ero(\mathsf{y})$ by exchanging the roles of x and y. Let us start by observing that $\mathcal{Z}ero(\mathsf{x})$ globally decrements the value of b by 1 and preserves the sum $\mathsf{a}+\mathsf{x}+\mathsf{y}+\mathsf{t}+\mathsf{c}$ since every line preserves it. Therefore, in order to maintain a holding invariant, $\mathcal{Z}ero(\mathsf{x})$ needs to quadruple the value of the sum of counters $\mathsf{a}+\mathsf{x}+\mathsf{y}+\mathsf{t}$. Similarly, to repair a broken invariant $\mathcal{Z}ero(\mathsf{x})$ needs to increase the value of $\mathsf{a}+\mathsf{x}+\mathsf{y}+\mathsf{t}$ by more than quadrupling it. We now study $\mathcal{Z}ero(\mathsf{x})$ in detail to show that the latter is impossible, and that the former only occurs under specific conditions that imply the statement of the claim. We split our analysis in two:

Lines 1–2: The loop on line 1, respectively 2, increases $\mathsf{a}+\mathsf{x}+\mathsf{y}+\mathsf{t}$ by at most a, respectively y. Therefore the value of $\mathsf{a}+\mathsf{x}+\mathsf{y}+\mathsf{t}$ is at most doubled, which occurs only if initially $\mathsf{t} = \mathsf{x} = 0$ and if then both loops are maximally iterated, resulting in $\mathsf{a} = \mathsf{y} = 0$.

Lines 3–4: The loop on line 3, respectively 4, increases $\mathsf{a}+\mathsf{x}+\mathsf{y}+\mathsf{t}$ by at most t, respectively x. Therefore the value of $\mathsf{a}+\mathsf{x}+\mathsf{y}+\mathsf{t}$ is at most doubled, which happens only if $\mathsf{a} = \mathsf{y} = 0$ upon reaching line 3 and if then both loops are maximally iterated, resulting in $\mathsf{t} = \mathsf{x} = 0$.

Combining both parts, we get that the value of $\mathsf{a}+\mathsf{x}+\mathsf{y}+\mathsf{t}$ is at most quadrupled through a run of $\mathcal{Z}ero(\mathsf{x})$, which only occurs if initially $\mathsf{t}=\mathsf{x}=0$ and all the loops are maximally iterated. This immediately implies that $\mathcal{Z}ero(\mathsf{x})$ cannot repair a broken invariant. Moreover, to maintain a holding invariant it is necessary to actually quadruple this value, therefore x needs to be 0 at the start; the maximal iteration of the loops will cause t to be 0 at the end; and the content of y will be completely moved to x by line 2 and then back to y by line 4, remaining unchanged as required by the statement. ◀

We now use $\mathcal{Z}ero(\mathsf{x})$ and $\mathcal{Z}ero(\mathsf{y})$ to transform $\mathcal{P}$ into a program $\mathcal{P}'$ satisfying Lemma 4. Formally, the program $\mathcal{P}'$ is obtained by applying the following modifications to $\mathcal{P}$:

- We add an increment (resp. decrement) of a to each line of $\mathcal{P}$ featuring a decrement (resp. increment) of x or y so that every line preserves the value of $\mathsf{a}+\mathsf{x}+\mathsf{y}+\mathsf{t}$;
- We replace each zero tests "**zero?** x" with a copy of the program $\mathcal{Z}ero(\mathsf{x})$, and each zero test "**zero?** y" with a copy of the program $\mathcal{Z}ero(\mathsf{y})$;

The first modification ensures that all the lines of $\mathcal{P}'$ except the calls to $\mathcal{Z}ero(\mathsf{x})$ and $\mathcal{Z}ero(\mathsf{y})$ maintain the invariant. We observe that for every complete run π' of $\mathcal{P}'$ that starts in a triple $\textsc{Triple}(A,B,\mathsf{a},\mathsf{b},\mathsf{c},\mathsf{X})$ and ends in a configuration where all counters except a are zero, the invariant is satisfied both at the beginning (by the definition of a triple) and at the end. Therefore, according to Claim 6.1 the invariant is never broken throughout π', indicating that the calls to $\mathcal{Z}ero(\mathsf{x})$ and $\mathcal{Z}ero(\mathsf{y})$ accurately simulate zero tests. Consequently, π' can be transformed into a matching run π of $\mathcal{P}$ with same values of x and y. In particular, π starts and ends with both counters equal to 0. Additionally, the value of b goes from B to 0 along π'. Since this value is decremented by one by each call to $\mathcal{Z}ero(\mathsf{x})$ or $\mathcal{Z}ero(\mathsf{y})$, π' goes through exactly B such calls, which translates into π performing exactly B zero tests.

To conclude the proof of Lemma 4, remark that we can also transform every run of $\mathcal{P}$ that starts and ends with both counters equal to 0 and performs B zero tests into a matching run of $\mathcal{P}'$ starting from some triple $\textsc{Triple}(A,B,\mathsf{a},\mathsf{b},\mathsf{c},\mathsf{X})$ and ending in a configuration where all counters except a are zero However, we must be cautious in choosing the initial value A of a to be sufficiently high. This ensures that we can increment x and y as high as required, despite the matching decrements of a added in $\mathcal{P}'$.

5 Amplifiers defined by counter programs

In this section we build amplifiers of polynomial size for the family of functions $(\mathbf{F}_d)_{d\in\mathbb{N}}$:

▶ **Lemma 5.** *For every $d\in\mathbb{N}_+$ there exists an $\mathbf{F}_d$-amplifier of size $\mathcal{O}(d)$ that uses $2d+4$ counters out of which d are end counters.*

The rest of this section is devoted to an inductive proof of Lemma 5. First we build an $\mathbf{F}_1$-amplifier $\mathcal{P}_1$ with 6 counters out of which one is an end counter (Lemma 7). Next we show how to lift an arbitrary F-amplifier with d counters into an $\widetilde{F}$-amplifier by adding two counters out of which one is an end counter (Lemma 8). Applying $d-1$ times our lifting process to the program $\mathcal{P}_1$ yields an $\mathbf{F}_d$-amplifier using $2d+4$ counters, proving Lemma 5.

Strong amplifiers. For the purpose of induction step, namely for lifting F-amplifiers to $\widetilde{F}$-amplifiers, we need a slight strengthening of the notion of amplifier. An F-amplifier $(\mathcal{P},(\mathsf{a},\mathsf{b},\mathsf{b}),(\mathsf{a},\mathsf{b}',\mathsf{c}'),\mathsf{t},\mathsf{Z})$ is called *strong* if every run π of $\mathcal{P}$ satisfies the following conditions (let $\Sigma\mathsf{Z}$ stand for the sum of all counters in Z):

1. The value of the sum $\mathsf{a}+\mathsf{c}+\mathsf{t}+\Sigma\mathsf{Z}$ is the same at the start and at the end of π;
2. If $(\mathsf{a}+\mathsf{c}+\mathsf{t}+\Sigma\mathsf{Z}-\mathsf{c}')\cdot 4^{\mathsf{b}'} < \mathsf{a}+\mathsf{c}+\mathsf{t}+\Sigma\mathsf{Z}$ holds at the start of π, it also holds at the end.

5.1 Construction of the $\mathbf{F}_1$-amplifier $\mathcal{P}_1$

Consider the following program with 6 counters $\mathsf{X} = \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{b}', \mathsf{c}', \mathsf{t}\}$:

Program $\mathcal{P}_1$:

1: **loop** $\mathsf{a} \longrightarrow \mathsf{t}$
2: **loop** $\mathsf{t} \longrightarrow \mathsf{a}$ $\quad \mathsf{c}' \mathrel{-}= 3 \quad \mathsf{c} \mathrel{+}= 3$
3: $\mathsf{b}' \mathrel{-}= 1 \quad \mathsf{b} \mathrel{+}= 1$
4: **loop**
5: $\quad$ **loop** $\mathsf{c} \longrightarrow \mathsf{t} \quad \mathsf{c}' \mathrel{-}= 1 \quad \mathsf{t} \mathrel{+}= 1$
6: $\quad$ **loop** $\mathsf{a} \xrightarrow{4} \mathsf{c} \quad \mathsf{c}' \mathrel{-}= 4 \quad \mathsf{c} \mathrel{+}= 1 \quad \mathsf{t} \mathrel{+}= 3$
7: $\quad$ **loop** $\mathsf{c} \longrightarrow \mathsf{a} \quad \mathsf{c}' \mathrel{-}= 1 \quad \mathsf{t} \mathrel{+}= 1$
8: $\quad$ **loop** $\mathsf{t} \longrightarrow \mathsf{c} \quad \mathsf{c}' \mathrel{-}= 1 \quad \mathsf{c} \mathrel{+}= 1$
9: $\quad \mathsf{b}' \mathrel{-}= 1 \quad \mathsf{b} \mathrel{+}= 2$

The program consists of an *initialisation* step lines 1–3, and an *iteration* step lines 4–9.

▶ **Lemma 7.** *The program* $(\mathcal{P}_1, \mathsf{X}, (\mathsf{a}, \mathsf{b}, \mathsf{c}), (\mathsf{a}, \mathsf{b}', \mathsf{c}'), \mathsf{t}, \{\mathsf{c}'\})$ *is a strong* $\mathbf{F}_1$*-amplifier.*

As an $\mathbf{F}_1$-amplifier, $\mathcal{P}_1$ is expected to map each input $\textsc{Triple}(A, B, \mathsf{a}, \mathsf{b}', \mathsf{c}', \mathsf{X})$ such that $4^{\mathbf{F}_1(B)-B}$ divides A to the output $\textsc{Triple}(A \cdot 4^{B-\mathbf{F}_1(B)}, \mathbf{F}_1(B), \mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{X})$. Since $\mathbf{F}_1(B) = 2B-1$, transforming the initial value $\mathsf{b}' = B$ into the final value $\mathsf{b} = \mathbf{F}_1(B)$ is easy: $\mathcal{P}_1$ first decrements b by 1 and increments b by 1 once (line 3), and then increments b by 2 whenever it decrements b' by 1 (line 9). It is more complicated to transform the initial value $\mathsf{a} = A$ into the final value $\mathsf{a} = A \cdot 4^{B-\mathbf{F}_1(B)}$: we need to divide $\mathbf{F}_1(B) - B = B - 1$ times the content of a by 4. We prove that $\mathcal{P}_1$ does so by studying the following invariant:

$$\textsc{Invariant:} \qquad (\mathsf{a}+\mathsf{c}+\mathsf{t}) \cdot 4^{\mathsf{b}'} = \mathsf{a}+\mathsf{c}+\mathsf{t}+\mathsf{c}' \text{ and } \mathsf{t} = 0;$$

$$\textsc{Broken invariant:} \qquad (\mathsf{a}+\mathsf{c}+\mathsf{t}) \cdot 4^{\mathsf{b}'} < \mathsf{a}+\mathsf{c}+\mathsf{t}+\mathsf{c}'.$$

Notice that saying that the invariant is broken is more specific than saying that the invariant does not hold. We now present two technical claims describing how the invariant evolves along both steps of $\mathcal{P}_1$. The proof of the claims can be found in Appendix A.

▶ **Claim 7.1.** *From each configuration where* $\mathsf{b}' > 0$, $\mathsf{c} = 0$ *and the invariant holds, there is a unique run through the initialisation step that maintains the invariant and preserves the value of* a*. All the other runs starting from this configuration, as well as the runs starting with a broken invariant, end with a broken invariant.*

▶ **Claim 7.2.** *From each configuration where* $\mathsf{b}' > 0$, a *is divisible by* 4 *and the invariant holds, there is a unique run through the iteration step that maintains the invariant and divides the value of* a *by* 4*. All the other runs starting from this configuration, as well as those starting with a broken invariant or a holding invariant with a value of* a *not divisible by* 4*, end with a broken invariant.*

We proceed with the proof of Lemma 7. Let $A, B \in \mathbb{N}$ and let π be a $\{c'\}$-zeroing run of the program $\mathcal{P}_1$ starting from $\textsc{Triple}(A, B, \mathsf{a}, \mathsf{b}', \mathsf{c}', \mathsf{X})$. Initially the counters satisfy:

$$\mathsf{a} = A, \quad \mathsf{b}' = B, \quad \mathsf{c}' = A \cdot (4^B - 1), \quad \mathsf{b} = \mathsf{c} = \mathsf{t} = 0.$$

This directly implies that the invariant holds at the beginning of π. Let us analyse the values of the counters at the end of π. We immediately get $\mathsf{c}' = 0$ since π is $\{c'\}$-zeroing. Note that this implies that the invariant cannot be broken as the counters always hold

non-negative integer values. As a consequence, Claims 7.1 and 7.2 imply that the invariant still holds at the end of π, and that π is the unique $\{c'\}$-zeroing run of of $\mathcal{P}_1$ starting from $\textsc{Triple}(A, B, \mathsf{a}, \mathsf{b}', \mathsf{c}', \mathsf{X})$. To conclude the proof, we now show that at the end of π all the counters match $\textsc{Triple}(A \cdot 4^{B-\mathbf{F}_1(B)}, \mathbf{F}_1(B), \mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{X})$:

$$\mathsf{a} = A \cdot 4^{B-\mathbf{F}_1(B)}, \quad \mathsf{b} = \mathbf{F}_1(B), \quad \mathsf{c} = A \cdot (4^B - 4^{B-\mathbf{F}_1(B)}), \quad \mathsf{b}' = \mathsf{c}' = \mathsf{t} = 0.$$

First, the invariant directly yields $\mathsf{t} = 0$, and also $\mathsf{b}' = 0$ by using the fact that $\mathsf{c}' = 0$. As b' starts with value B and is decremented by one along the initialisation step and each iteration step, we get that π visits the iteration step $B - 1$ times. In turn, this implies that the final value of b is $2B - 1 = \mathbf{F}_1(B)$, and by Claims 7.1 and 7.2 we also get that the final value of a is $\frac{A}{4^{B-1}} = A \cdot 4^{B-\mathbf{F}_1(B)}$. Combining this with the fact that the initial value $A \cdot 4^B$ of the sum $\mathsf{a} + \mathsf{c} + \mathsf{t} + \mathsf{c}'$ is preserved along π finally yields the appropriate value for c.

Note that the run π exists if and only if $4^{B-\mathbf{F}_1(B)}$ divides A, otherwise Claim 7.2 implies that the invariant is broken before the end of the run. This proves that $\mathcal{P}_1$ is an $\mathbf{F}_1$-amplifier. The fact that $\mathcal{P}_1$ is a *strong* $\mathbf{F}_1$-amplifier then directly follows from Claims 7.1 and 7.2.

5.2 Construction of the $\widetilde{F}$-amplifier $\widetilde{\mathcal{P}}$ from an F-amplifier $\mathcal{P}$

Let $(\mathcal{P}, \mathsf{X}, (\mathsf{a}, \mathsf{b}, \mathsf{c}), (\mathsf{a}, \mathsf{b}', \mathsf{c}'), \mathsf{t}, \mathsf{Z})$ be a strong F-amplifier for some function $F : \mathbb{N}_+ \to \mathbb{N}_+$. We construct a strong $\widetilde{F}$-amplifier $\widetilde{\mathcal{P}}$ out of $\mathcal{P}$. The program $\widetilde{\mathcal{P}}$ uses the counters of $\mathcal{P}$ plus two fresh input counters b'' and c'', and it shares the output counters of $\mathcal{P}$:

Program $\widetilde{\mathcal{P}}$:

1: **loop** $\mathsf{a} \longrightarrow \mathsf{t}$
2: **loop** $\mathsf{t} \longrightarrow \mathsf{a}$ $\quad \mathsf{c}'' \mathrel{-}= 3 \quad \mathsf{c} \mathrel{+}= 3$
3: $\mathsf{b}'' \mathrel{-}= 1 \quad \mathsf{b} \mathrel{+}= 1$
4: **loop**
5: $\quad$ **loop** $\mathsf{a} \longrightarrow \mathsf{t}$
6: $\quad$ **loop** $\mathsf{t} \longrightarrow \mathsf{a}$ $\quad \mathsf{c}'' \mathrel{-}= 3 \quad \mathsf{a} \mathrel{+}= 3$
7: $\quad$ **loop** $\mathsf{c} \longrightarrow \mathsf{c}'$ $\quad \mathsf{c}'' \mathrel{-}= 3 \quad \mathsf{c}' \mathrel{+}= 3$
8: $\quad$ **loop** $\mathsf{b} \longrightarrow \mathsf{b}'$
9: $\quad \mathcal{P}$
10: $\quad \mathsf{b}'' \mathrel{-}= 1$

Similarly to $\mathcal{P}_1$ the program $\widetilde{\mathcal{P}}$ consists of an *initialisation* step lines 1–3 (differing from $\mathcal{P}_1$ only by renaming counters), and an *iteration* step lines 4–10 (differing significantly from $\mathcal{P}_1$).

▶ **Lemma 8.** *For every strong F-amplifier $(\mathcal{P}, \mathsf{X}, (\mathsf{a}, \mathsf{b}, \mathsf{c}), (\mathsf{a}, \mathsf{b}', \mathsf{c}'), \mathsf{t}, \mathsf{Z})$, the program $(\widetilde{\mathcal{P}}, \mathsf{X} \cup \{\mathsf{b}'', \mathsf{c}''\}, (\mathsf{a}, \mathsf{b}, \mathsf{c}), (\mathsf{a}, \mathsf{b}'', \mathsf{c}''), \mathsf{Z} \cup \{\mathsf{c}''\})$ is a strong $\widetilde{F}$-amplifier.*

The proof of Lemma 8 can be found in Appendix B. Here, we provide an overview of the main intuitions behind it. As an $\widetilde{F}$-amplifier, $\widetilde{\mathcal{P}}$ is expected to map each input $\textsc{Triple}(A, B, \mathsf{a}, \mathsf{b}'', \mathsf{c}'', \mathsf{X} \cup \{\mathsf{b}'', \mathsf{c}''\})$ such that $4^{\widetilde{F}(B)-F(B)}$ divides A to the output $\textsc{Triple}(A \cdot 4^{F(B)-\widetilde{F}(B)}, \widetilde{F}(B), \mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{X} \cup \{\mathsf{b}'', \mathsf{c}''\})$. The intended behaviour of $\widetilde{\mathcal{P}}$ is straightforward: Since for all $n \in \mathbb{N}_+$ the value $\widetilde{F}(n)$ is obtained by applying the function F to 1 for $n - 1$ consecutive times, we expect $\widetilde{\mathcal{P}}$ to apply the program $\mathcal{P}$ exactly $B - 1$ times to transform a 1-triple into an $\widetilde{\mathcal{F}}(B)$-triple. To show that $\widetilde{\mathcal{P}}$ behaves as expected, we study the following invariant:

$$\textsc{Invariant:} \qquad (\mathsf{a} + \mathsf{c} + \mathsf{t}) \cdot 4^{\mathsf{b}''} = \mathsf{a} + \mathsf{c} + \mathsf{t} + \mathsf{c}'' \text{ and } \mathsf{t} = 0;$$

$$\textsc{Broken invariant:} \qquad (\mathsf{a} + \mathsf{c} + \mathsf{t}) \cdot 4^{\mathsf{b}''} < \mathsf{a} + \mathsf{c} + \mathsf{t} + \mathsf{c}''.$$

The starting configuration $\textsc{Triple}(A, B, \mathsf{a}, \mathsf{b}'', \mathsf{c}'', \mathsf{X} \cup \{\mathsf{b}'', \mathsf{c}''\})$ satisfies the invariant. The program $\widetilde{\mathcal{P}}$ is designed such that every run π starting from such a configuration then satisfies:

- If along π all the loops are maximally iterated and all the calls to $\mathcal{P}$ are Z-zeroing, then the invariant holds until the end of π. Moreover, π then matches the expected behaviour of $\widetilde{\mathcal{P}}$ described above. In particular, π will correctly amplify $B-1$ times via $\mathcal{P}$ a triple $\textsc{Triple}(A, 1, \mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{X})$, thus ending in $\textsc{Triple}(A \cdot 4^{F(B) - \widetilde{F}(B)}, \widetilde{F}(B), \mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{X} \cup \{\mathsf{b}'', \mathsf{c}''\})$.
- However, if π fails to maximally iterate one loop, or does a call to $\mathcal{P}$ that is not Z-zeroing, then the invariant is irremediably broken, which implies that π is not $(Z \cup \{c''\})$-zeroing.

This proves that $\widetilde{\mathcal{P}}$ is a strong $\widetilde{F}$-amplifier.

6 Amplifiers defined by counter programs with a pushdown stack

In this section, we implement more efficient $\mathbf{F}_d$-amplifiers using a stack:

▶ **Lemma 6.** *For every $d \in \mathbb{N}_+$ there exists an $\mathbf{F}_d$-amplifier of size $\mathcal{O}(d)$ that uses a stack and $\lfloor \frac{d}{2} \rfloor + 4$ counters out of which $\lfloor \frac{d}{2} \rfloor$ are end counters.*

Our construction is based on the amplifiers from Section 5. The main intuitive idea is to 'delegate' some counters to the stack. The stack alphabet consists exactly of those counters which are delegated, and the value of each delegated counter x corresponds to the number of occurrences of the symbol x on the stack. Therefore, 'delegated counters' can be understood as a synonym of 'stack symbols' in the sequel. This idea motivates the following definition. Let S and X be two disjoint sets of delegated, respectively non-delegated, counters. We define the function

$$h_{\mathsf{X},\mathsf{S}} : \mathbb{N}^{\mathsf{X}} \times \mathsf{S}^* \to \mathbb{N}^{\mathsf{X} \cup \mathsf{S}}$$

that maps a configuration, i.e., a valuation v of the non-delegated counters of X together with a stack content $s \in \mathsf{S}^*$, to a valuation of all the counters from $\mathsf{X} \cup \mathsf{S}$, as follows: $h_{\mathsf{X},\mathsf{S}}(v, s) = v'$, where $v'(\mathsf{x}) = v(\mathsf{x})$ for $\mathsf{x} \in \mathsf{X}$, and $v'(\mathsf{x})$ is the number of occurrences of x in s for $\mathsf{x} \in \mathsf{S}$.

Using this definition, we establish a notion of *simulation* between programs with or without stack. Given an F-amplifier $\mathcal{P}$ with set of counters $\mathsf{X} \cup \mathsf{S}$, we say that a counter program with a stack $\mathcal{Q}$ *simulates* $\mathcal{P}$ if it satisfies the two following conditions:

- For every $A, B \in \mathbb{N}_+$ such that A is divisible by $4^{(F(B)-B)}$ there exists a run of $\mathcal{Q}$ between two configurations x and y satisfying $h_{\mathsf{X},\mathsf{S}}(x) = \textsc{Triple}(A, B, \mathsf{a}, \mathsf{b}', \mathsf{c}', \mathsf{X})$ and $h_{\mathsf{X},\mathsf{S}}(y) = \textsc{Triple}(A \cdot 4^{(B-F(B))}, F(B), \mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{X})$.
- For every run of $\mathcal{Q}$ between two configurations x and y there exists a run of $\mathcal{P}$ between $h_{\mathsf{X},\mathsf{S}}(x)$ and $h_{\mathsf{X},\mathsf{S}}(y)$;

We say that such a program with stack $\mathcal{Q}$ is an an F-amplifier.

The rest of this section is devoted to the proof of Lemma 6. We rely on the constructions of Section 5, and similarly proceed in two steps. First, we transform the $\mathbf{F}_1$-amplifier $\mathcal{P}_1$ into a program with stack $\mathcal{Q}_1$ that simulates $\mathcal{P}_1$ with four counters, as it delegates the two other counters to the stack (Lemma 9). Next, we adapt the constructions used to lift F-amplifiers into $\widetilde{F}$-amplifier. This time, we will have *two* constructions that can be applied alternatively: the first introduces one counter and one delegated counter, and the second introduces two delegated counters (Lemma 10). Therefore, for every $d \in \mathbb{N}$, starting with the program $\mathcal{Q}_1$ and applying alternatively our two lifting constructions yields a $\mathbf{F}_d$-amplifier with $\lfloor \frac{d}{2} \rfloor + 4$ counters (as the other counters are delegated to the stack), which proves Lemma 6.

6.1 Construction of the $\mathbf{F}_1$-amplifier $\mathcal{Q}_1$

Consider the following program with 4 counters $\mathsf{X} = \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{t}\}$ and the stack alphabet $\mathsf{S} = \{\mathsf{b}', \mathsf{c}'\}$, which is obtained from the program $\mathcal{P}_1$ defined in Section 5 by replacing each decrement on b' and c' by the corresponding pop operation:

Program $\mathcal{Q}_1$:

1: **loop** $\mathsf{a} \longrightarrow \mathsf{t}$
2: **loop** $\mathsf{t} \longrightarrow \mathsf{a}$ $\quad \text{POP}(\mathsf{c}'\mathsf{c}'\mathsf{c}')$ $\quad \mathsf{c} \mathrel{+}= 3$
3: $\text{POP}(\mathsf{b}')$ $\quad \mathsf{b} \mathrel{+}= 1$
4: **loop**
5: $\quad$ **loop** $\mathsf{c} \longrightarrow \mathsf{t}$ $\quad \text{POP}(\mathsf{c}')$ $\quad \mathsf{t} \mathrel{+}= 1$
6: $\quad$ **loop** $\mathsf{a} \xrightarrow{4} \mathsf{c}$ $\quad \text{POP}(\mathsf{c}'\mathsf{c}'\mathsf{c}'\mathsf{c}')$ $\quad \mathsf{c} \mathrel{+}= 1$ $\quad \mathsf{t} \mathrel{+}= 3$
7: $\quad$ **loop** $\mathsf{c} \longrightarrow \mathsf{a}$ $\quad \text{POP}(\mathsf{c}')$ $\quad \mathsf{t} \mathrel{+}= 1$
8: $\quad$ **loop** $\mathsf{t} \longrightarrow \mathsf{c}$ $\quad \text{POP}(\mathsf{c}')$ $\quad \mathsf{c} \mathrel{+}= 1$
9: $\quad \text{POP}(\mathsf{b}')$ $\quad \mathsf{b} \mathrel{+}= 2$

▶ **Lemma 9.** *The program $\mathcal{Q}_1$ simulates the $\mathbf{F}_1$-amplifier $(\mathcal{P}_1, \mathsf{X}, (\mathsf{a}, \mathsf{b}, \mathsf{c}), (\mathsf{a}, \mathsf{b}', \mathsf{c}'), \mathsf{t}, \{\mathsf{c}''\})$.*

Proof. Let h denote the function $h_{\{\mathsf{a},\mathsf{b},\mathsf{c},\mathsf{t}\},\{\mathsf{b}',\mathsf{c}'\}}$ that transforms configurations of $\mathcal{Q}_1$ into configurations of $\mathcal{P}_1$. The program $\mathcal{Q}_1$ is a constrained version of $\mathcal{P}_1$: every line is identical with the added restriction that lines 2, 3, 6 and 9 can only be fired if the appropriate symbol is at the top of the stack. Therefore, we immediately get the second condition required for $\mathcal{Q}_1$ to simulate $\mathcal{P}_1$: for every run π of $\mathcal{Q}_1$ between two configurations x and y, the run π' of $\mathcal{P}$ that starts in $h(x)$ and uses the same lines as π is a valid run of $\mathcal{P}$ that ends in $h(y)$.

To conclude, we show that we can transform the $\{\mathsf{b}', \mathsf{c}'\}$-zeroing runs of $\mathcal{P}_1$ (thus in particular the runs that witness the $\mathbf{F}_1$-amplifier behaviour) into runs of $\mathcal{Q}_1$. To do so we rely on the fact that the counters b' and c' are only decreasing along the runs of $\mathcal{P}_1$. Formally, given a $\{\mathsf{b}', \mathsf{c}'\}$-zeroing run π of $\mathcal{P}_1$ between two configurations x and y, let $u_\pi \in \{\mathsf{b}', \mathsf{c}'\}^*$ be the word listing, in order, the occurrences of the decrements of b' and c' along π. We define a configuration x' of $\mathcal{Q}_1$ as follows: the counters $\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{t}$ match the content they have in the starting configuration x of π, and the stack content is the reverse of the word u_π so that the first letter of u_π is at the top of the stack. This definition guarantees that:

- We have $h(x') = x$. Indeed, since π is $\{\mathsf{b}', \mathsf{c}'\}$-zeroing, the value of the counters b' and c' in the initial configuration x is equal to the number of times these counters are decremented;
- There exists a run π' of $\mathcal{Q}_1$ that starts from x' and follows the same lines as π: whenever a popping instruction appears the adequate symbol will be at the top of the stack. As the lines of $\mathcal{P}_1$ and $\mathcal{Q}_1$ are analogous, the configuration y' reached by π' satisfies $h(y') = y$. ◀

6.2 Construction of the $\widetilde{F}$-amplifiers $\widetilde{\mathcal{Q}}$ and $\overline{\mathcal{Q}}$ from an F-amplifier $\mathcal{Q}$

In Section 5, we showed how to lift an F-amplifier $\mathcal{P}$ into an $\widetilde{F}$-amplifier $\widetilde{\mathcal{P}}$. We now show two different manners of adapting the construction of $\widetilde{\mathcal{P}}$ in order to lift a program with stack $\mathcal{Q}$ simulating $\mathcal{P}$ into a program with stack simulating $\widetilde{\mathcal{P}}$.

▶ **Lemma 10.** *Let $\mathcal{Q}$ be a program simulating a strong F-amplifier $(\mathcal{P}, \mathsf{X}, (\mathsf{a}, \mathsf{b}, \mathsf{c}), (\mathsf{a}, \mathsf{b}', \mathsf{c}'), \mathsf{t}, \mathsf{Z})$ without delegating the counters $\mathsf{a}, \mathsf{b}, \mathsf{c}$ and t.*

- *If $\mathcal{Q}$ delegates b' but not c', then $\widetilde{\mathcal{Q}}$ simulates $(\widetilde{\mathcal{P}}, \mathsf{X} \cup \{\mathsf{b}'', \mathsf{c}''\}, (\mathsf{a}, \mathsf{b}, \mathsf{c}), (\mathsf{a}, \mathsf{b}'', \mathsf{c}''), \mathsf{Z} \cup \{\mathsf{c}''\})$ while delegating two input counters b'' and c'' in addition to the counters delegated by $\mathcal{Q}$.*

- *If $\mathcal{Q}$ delegates b' and c', then $\overline{\mathcal{Q}}$ simulates $(\widetilde{\mathcal{P}}, \mathsf{X} \cup \{\mathsf{b}'', \mathsf{c}''\}, (\mathsf{a}, \mathsf{b}, \mathsf{c}), (\mathsf{a}, \mathsf{b}'', \mathsf{c}''), \mathsf{Z} \cup \{\mathsf{c}''\})$ while delegating only one input counter b'' in addition to the counters delegated by $\mathcal{Q}$.*

Program $\widetilde{\mathcal{Q}}$:

```
1: loop a ⟶ t
2: loop t ⟶ a   POP(c″c″c″)   c += 3
3: POP(b″)   b += 1
4: loop
5:     loop a ⟶ t
6:     loop t ⟶ a  POP(c″c″c″)  a += 3
7:     loop c ⟶ c′  POP(c″c″c″)  c′ += 3
8:     loop b −= 1   PUSH(b′)
9:     Q
10:    POP(b″)
```

Program $\overline{\mathcal{Q}}$:

```
1: loop a ⟶ t
2: loop t ⟶ a   c″ −= 3   c += 3
3: POP(b″)   b += 1
4: loop
5:     loop a ⟶ t
6:     loop t ⟶ a   c″ −= 3   a += 3
7:     loop
8:         loop b −= 1   PUSH(b′)
9:         c −= 1   c″ −= 3   PUSH(c′)
10:        loop b −= 1   PUSH(b′)
11:        PUSH(c′)
12:        loop b −= 1   PUSH(b′)
13:        PUSH(c′)
14:        loop b −= 1   PUSH(b′)
15:        PUSH(c′)
16:        loop b −= 1   PUSH(b′)
17:    Q
18:    POP(b″)
```

The proof of Lemma 10 can be found in Appendix C. To convey the intuition behind it we analyse the differences between the two programs. The main difference concerns the counters delegated to the stack: If $\mathcal{Q}$ delegates only b', then the starting configurations for the calls to $\mathcal{Q}$ are easy to setup as the stack simply contains a sequence of b'. Therefore $\mathcal{Q}$ can be lifted via $\widetilde{\mathcal{Q}}$ which delegates both b'' and c''. However, if $\mathcal{Q}$ delegates both b' and c', then the starting configurations required for the calls to $\mathcal{Q}$ are more complex: the stack needs to contain the symbols b' and c' in a specific order. This prevents us from delegating both b'' and c'' to the stack, thus we need to lift $\mathcal{Q}$ via $\overline{\mathcal{Q}}$ which delegates only b''. and keeps c'' as a standard counter. A second difference between $\widetilde{\mathcal{Q}}$ and $\overline{\mathcal{Q}}$ concerns the loops updating b' and c' in the iteration step. To understand what is happening here, let us have a look at what happens when we replace the push and pop instructions by increments and decrements:

```
1: loop c ⟶ c′   c″ −= 3   c′ += 3
2: loop b ⟶ b′
```

```
1: loop
2:     loop b ⟶ b′
3:     c −= 1   c′ += 1   c″ −= 3
4:     loop b ⟶ b′
5:     c′ += 1
6:     loop b ⟶ b′
7:     c′ += 1
8:     loop b ⟶ b′
9:     c′ += 1
10:    loop b ⟶ b′
```

While these two sequences of instructions are different, we can remark that their global effect is identical, in the sense that every counter update realisable by the left one is also realisable by the right one, and reciprocally. However, if b' and c' are delegated to the stack then the sequence of instruction on the right is more powerful, as it performs the same number of increments of b' and c', but *in any order*, which allows to create many different stack contents. This is required so that $\overline{\mathcal{Q}}$ can construct the stack contents needed to call $\mathcal{Q}$.

References

1. Mohamed Faouzi Atig and Pierre Ganty. Approximating Petri Net Reachability Along Context-free Traces. In *Proceedings of FSTTCS 2011*, volume 13 of *LIPIcs*, pages 152–163, 2011.
2. Michael Blondin, Alain Finkel, Stefan Göller, Christoph Haase, and Pierre McKenzie. Reachability in Two-Dimensional Vector Addition Systems with States Is PSPACE-Complete. In *Proceedings of LICS 2015*, pages 32–43, 2015.
3. Wojciech Czerwinski, Slawomir Lasota, Ranko Lazic, Jérôme Leroux, and Filip Mazowiecki. The reachability problem for Petri nets is not elementary. In *Proceedings of STOC 2019*, pages 24–33. ACM, 2019.
4. Wojciech Czerwinski, Slawomir Lasota, Ranko Lazic, Jérôme Leroux, and Filip Mazowiecki. The Reachability Problem for Petri Nets Is Not Elementary. *Journal of the ACM*, 68(1):7:1–7:28, 2021.
5. Wojciech Czerwinski and Lukasz Orlikowski. Reachability in Vector Addition Systems is Ackermann-complete. In *Proceedings of FOCS 2021*, pages 1229–1240. IEEE, 2021.
6. Wojciech Czerwinski and Lukasz Orlikowski. Lower Bounds for the Reachability Problem in Fixed Dimensional VASSes. In *Proceedings of LICS 2022*, pages 40:1–40:12. ACM, 2022.
7. Matthias Englert, Piotr Hofman, Slawomir Lasota, Ranko Lazic, Jérôme Leroux, and Juliusz Straszynski. A lower bound for the coverability problem in acyclic pushdown VAS. *Inf. Process. Lett.*, 167:106079, 2021.
8. Matthias Englert, Ranko Lazić, and Patrick Totzke. Reachability in Two-Dimensional Unary Vector Addition Systems with States is NL-Complete. In *Proceedings of LICS 2016*, pages 477–484. ACM, 2016.
9. Christoph Haase, Stephan Kreutzer, Joël Ouaknine, and James Worrell. Reachability in succinct and parametric one-counter automata. In *Proceeding of CONCUR 2009*, volume 5710, pages 369–383. Springer, 2009.
10. Slawomir Lasota. Improved Ackermannian Lower Bound for the Petri Nets Reachability Problem. In *Proceedings of STACS 2022*, volume 219 of *LIPIcs*, pages 46:1–46:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.
11. Jérôme Leroux. The reachability problem for petri nets is not primitive recursive. *CoRR*, abs/2104.12695, 2021.
12. Jérôme Leroux. The Reachability Problem for Petri Nets is Not Primitive Recursive. In *Proceedings of FOCS 2021*, pages 1241–1252. IEEE, 2021.
13. Jérôme Leroux, M. Praveen, and Grégoire Sutre. Hyper-Ackermannian bounds for pushdown vector addition systems. In *Proceedings of CSL-LICS 2014*, pages 63:1–63:10. ACM, 2014.
14. Jérôme Leroux and Sylvain Schmitz. Reachability in Vector Addition Systems is Primitive-Recursive in Fixed Dimension. In *Proceedings of LICS 2019*, pages 1–13. IEEE, 2019.
15. Jérôme Leroux, Grégoire Sutre, and Patrick Totzke. On the Coverability Problem for Pushdown Vector Addition Systems in One Dimension. In *Proceedings of ICALP 2015*, volume 9135 of *Lecture Notes in Computer Science*, pages 324–336. Springer, 2015.
16. Richard J. Lipton. The Reachability Problem Requires Exponential Space. Technical report, Yale University, 1976.
17. Marvin L. Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall Series in Automatic Computation. Prentice-Hall, 1967.
18. Sylvain Schmitz. Complexity Hierarchies beyond Elementary. *TOCT*, 8(1):3:1–3:36, 2016.

A Proof of Claims 7.1 and 7.2

▶ **Claim 7.1.** *From each configuration where* $\mathsf{b}' > 0$, $\mathsf{c} = 0$ *and the invariant holds, there is a unique run through the initialisation step that maintains the invariant and preserves the value of* a. *All the other runs starting from this configuration, as well as the runs starting with a broken invariant, end with a broken invariant.*

▶ **Claim 7.2.** *From each configuration where* $\mathsf{b}' > 0$, a *is divisible by* 4 *and the invariant holds, there is a unique run through the iteration step that maintains the invariant and divides the value of* a *by* 4. *All the other runs starting from this configuration, as well as those starting with a broken invariant or a holding invariant with a value of* a *not divisible by* 4, *end with a broken invariant.*

Let us recall the program and the invariant mentioned in these two statements:

Program $\mathcal{P}_1$:

1: **loop** $\mathsf{a} \longrightarrow \mathsf{t}$
2: **loop** $\mathsf{t} \longrightarrow \mathsf{a}$ $\quad \mathsf{c}' \mathrel{-}= 3 \quad \mathsf{c} \mathrel{+}= 3$
3: $\mathsf{b}' \mathrel{-}= 1 \quad \mathsf{b} \mathrel{+}= 1$
4: **loop**
5: $\quad$ **loop** $\mathsf{c} \longrightarrow \mathsf{t} \quad \mathsf{c}' \mathrel{-}= 1 \quad \mathsf{t} \mathrel{+}= 1$
6: $\quad$ **loop** $\mathsf{a} \xrightarrow{4} \mathsf{c} \quad \mathsf{c}' \mathrel{-}= 4 \quad \mathsf{c} \mathrel{+}= 1 \quad \mathsf{t} \mathrel{+}= 3$
7: $\quad$ **loop** $\mathsf{c} \longrightarrow \mathsf{a} \quad \mathsf{c}' \mathrel{-}= 1 \quad \mathsf{t} \mathrel{+}= 1$
8: $\quad$ **loop** $\mathsf{t} \longrightarrow \mathsf{c} \quad \mathsf{c}' \mathrel{-}= 1 \quad \mathsf{c} \mathrel{+}= 1$
9: $\quad \mathsf{b}' \mathrel{-}= 1 \quad \mathsf{b} \mathrel{+}= 2$

INVARIANT: $(\mathsf{a}+\mathsf{c}+\mathsf{t}) \cdot 4^{\mathsf{b}'} = \mathsf{a}+\mathsf{c}+\mathsf{t}+\mathsf{c}'$ and $\mathsf{t} = 0$;

BROKEN INVARIANT: $(\mathsf{a}+\mathsf{c}+\mathsf{t}) \cdot 4^{\mathsf{b}'} < \mathsf{a}+\mathsf{c}+\mathsf{t}+\mathsf{c}'$.

The initialisation step and the iteration step both decrement b' by one and preserve the right-hand side $\mathsf{a}+\mathsf{c}+\mathsf{t}+\mathsf{c}'$ of the invariant as every line of $\mathcal{P}_1$ preserves this sum. Hence, to maintain the invariant the sum $\mathsf{a}+\mathsf{c}+\mathsf{t}$ needs to be quadrupled, and to repair a broken invariant the sum $\mathsf{a}+\mathsf{c}+\mathsf{t}$ needs to be increased by an even larger amount. We show that in both steps the latter is impossible, and the former only happens if all loops are maximally iterated, which implies the modification of the counter a required by the statements.

Proof of Claim 7.1. The value of $\mathsf{a}+\mathsf{c}+\mathsf{t}$ is at most increased by $3 \cdot (\mathsf{a}+\mathsf{t})$ along the initialisation part, as line 1 preserves this sum and moves the content of a to t, then line 2 increases this sum by at most 3 times the value of t. Therefore $\mathsf{a}+\mathsf{c}+\mathsf{t}$ is at most quadrupled, which implies that the initialisation step cannot repair a broken invariant. Moreover, to maintain a holding invariant the program $\mathcal{P}_1$ needs to quadruple this sum. This happens if and only if initially $\mathsf{b}' > 0$, $\mathsf{c} = 0$ and $\mathsf{c}' \geq 3 \cdot (\mathsf{a}+\mathsf{t})$ (this last condition is implied by the invariant); and if then both loops are maximally iterated. Finally, remark that upon maximal iteration of the loops a and t keep their initial values, as required. ◀

Proof of Claim 7.2. We divide our analysis of the iteration step in two parts:

Lines 5–6 The loop on line 5, respectively line 6, increases $\mathsf{a}+\mathsf{c}+\mathsf{t}$ by at most the value of c, respectively a. Therefore the value of $\mathsf{a}+\mathsf{c}+\mathsf{t}$ is at most doubled, which occurs only if initially $\mathsf{t} = 0$ and then both loops are maximally iterated, resulting in $\mathsf{a} = 0$.

Lines 7–8 The loop on line 7, respectively 8, increases $\mathsf{a}+\mathsf{c}+\mathsf{t}$ by at most the value of c, respectively t. Therefore the value of $\mathsf{a}+\mathsf{c}+\mathsf{t}$ is at most doubled, which occurs only if $\mathsf{a}=0$ upon reaching line 7 and then both loops are maximally iterated, resulting in $\mathsf{t}=0$.

Combining the two parts, we get that the value of $\mathsf{a}+\mathsf{c}+\mathsf{t}$ is at most quadrupled by the iteration step. This directly implies that it is not possible to repair a broken invariant. Moreover, to maintain a holding invariant this sum needs to be quadrupled. This happens if and only if at the start of the iteration step $\mathsf{b}'>0$, a is divisible by 4, $\mathsf{t}=0$ and $\mathsf{c}'\geq 3\cdot(\mathsf{a}+\mathsf{c})$ (note that the last two conditions are implied by the invariant); and if then the four loops are maximally iterated.[2] To conclude, remark that maximally iterating lines 6–7 results in dividing the value of a by 4: line 6 transfers one fourth of the value of a to c, which is then transferred back to a by line 7. ◀

B Proof of Lemma 5

▶ **Lemma 8.** *For every strong F-amplifier $(\mathcal{P},\mathsf{X},(\mathsf{a},\mathsf{b},\mathsf{c}),(\mathsf{a},\mathsf{b}',\mathsf{c}'),\mathsf{t},\mathsf{Z})$, the program $(\widetilde{\mathcal{P}},\mathsf{X}\cup\{\mathsf{b}'',\mathsf{c}''\},(\mathsf{a},\mathsf{b},\mathsf{c}),(\mathsf{a},\mathsf{b}'',\mathsf{c}''),\mathsf{Z}\cup\{\mathsf{c}''\})$ is a strong $\widetilde{F}$-amplifier.*

The proof is structured as follows: We begin with a technical lemma implying that $\widetilde{\mathcal{P}}$ satisfies the two invariants required to be a *strong* amplifier (Claim 10.1). Then, to show that $\widetilde{\mathcal{P}}$ is an $\widetilde{\mathcal{F}}$-amplifier, we formalise the expected behaviour of the runs of $\widetilde{\mathcal{P}}$ (Equations (3)–(7)), we show that the runs that fit this expected behaviour $(Z\cup\{c''\})$-compute $\widetilde{\mathcal{F}}$ (Claim 10.2), and that the runs that do not fit this expected behaviour are not $(Z\cup\{c''\})$-zeroing (Claim 10.3).

Invariants of $\widetilde{\mathcal{P}}$. Before delving into the intricate functioning of $\widetilde{\mathcal{P}}$, we show two invariants that hold for every run. On top of being prerequisites for $\widetilde{\mathcal{P}}$ to qualify as a *strong* amplifier, these invariants offer valuable assistance in proving the next results of this section.

▶ **Claim 10.1.** *The initialisation step and the iteration step of $\widetilde{\mathcal{P}}$ both preserve the value of $\mathsf{a}+\mathsf{c}+\mathsf{t}+\Sigma\mathsf{Z}+\mathsf{c}''$ and either preserve or decrease the value of $(\mathsf{a}+\mathsf{c}+\mathsf{t}+\Sigma\mathsf{Z})\cdot 4^{\mathsf{b}''}$.*

Proof. We start by observing that the sum $\mathsf{a}+\mathsf{c}+\mathsf{t}+\Sigma\mathsf{Z}+\mathsf{c}''$ stays constant along every run of $\widetilde{\mathcal{P}}$: every command line preserves it, including line 9 since $\mathcal{P}$ is a strong F-amplifier. We now study the effect of the initialisation and iteration steps on the value of $(\mathsf{a}+\mathsf{c}+\mathsf{t}+\Sigma\mathsf{Z})\cdot 4^{\mathsf{b}''}$.

Initialisation: The sum $\mathsf{a}+\mathsf{c}+\mathsf{t}+\Sigma\mathsf{Z}$ is preserved in lines 1 and 3 and is increased in line 2 by at most three times the value of t, thus it is at most quadrupled by the initialisation step. Since b'' is decremented by 1 during the initialisation step, this proves that the value of $(\mathsf{a}+\mathsf{c}+\mathsf{t}+\Sigma\mathsf{Z})\cdot 4^{\mathsf{b}''}$ is either preserved or decreased.

Iteration: The sum $\mathsf{a}+\mathsf{c}+\mathsf{t}+\Sigma\mathsf{Z}$ is increased in lines 5–6 by at most three times the value of $\mathsf{a}+\mathsf{t}$; it is increased in line 7 by at most three times the value of c; and it is then preserved in lines 8, 9 and 10 (since $\mathcal{P}$ is a strong amplifier). Hence the value of $\mathsf{a}+\mathsf{c}+\mathsf{t}+\Sigma\mathsf{Z}$ is at most quadrupled by each occurrence of the iteration step. Since b'' is decremented by 1, this shows that the value of $(\mathsf{a}+\mathsf{c}+\mathsf{t}+\Sigma\mathsf{Z})\cdot 4^{\mathsf{b}''}$ is either preserved or decreased. ◀

[2] The fact that a is divisible by 4 is what allows line 6 to be maximally iterated: if it is not the case, the run would eventually get stuck with a content of a smaller than four but greater than zero.

Expected behaviour of $\widetilde{\mathcal{P}}$. The intended behaviour of $\widetilde{\mathcal{P}}$ is straightforward. We start with a B-triple over the counters $\mathsf{a}, \mathsf{b}'', \mathsf{c}''$. The initialisation step establishes a 1-triple over $\mathsf{a}, \mathsf{b}, \mathsf{c}$. Next, in the iteration step, this 1-triple is first moved to $\mathsf{a}, \mathsf{b}', \mathsf{c}'$, and then $\mathcal{P}$ is invoked to transform it into a $F(1)$-triple over $\mathsf{a}, \mathsf{b}, \mathsf{c}$. By repeating the iteration step $B-2$ more times, we obtain a $F^{B-1}(1) = \widetilde{\mathcal{F}}(B)$-triple over $\mathsf{a}, \mathsf{b}, \mathsf{c}$, as expected from a $\widetilde{\mathcal{F}}$-amplifier. We now formalise this expected behaviour as a set of equations.

Let π be a run of $\widetilde{\mathcal{P}}$ that visits the iteration step n times. Let $w_0(\pi)$ denote the valuation of the counter set $\mathsf{X} \cup \{\mathsf{b}'', \mathsf{c}''\}$ at the start of π, and $x_n(\pi)$ denote the valuation of the counter set X at the end of π. Moreover, for every $i = 0, 1 \ldots, n-1$, we use $x_i(\pi)$ and $y_i(\pi)$ to denote the valuation of X at the start of the $(i+1)$th iteration step of π and at the start of the $(i+1)$th call to the program $\mathcal{P}$, respectively. This notation allows us to formally express the expected behaviour described earlier:

$$w_0(\pi) = \textsc{Triple}(A, B, \mathsf{a}, \mathsf{b}'', \mathsf{c}'', \mathsf{X} \cup \{\mathsf{b}'', \mathsf{c}''\}) \tag{3}$$

$$x_0(\pi) = \textsc{Triple}(A, 1, \mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{X}) \tag{4}$$

$$x_i(\pi) = \textsc{Triple}(A \cdot 4^{i+1-F^i(1)}, F^i(1), \mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{X}) \tag{5}$$

$$y_i(\pi) = \textsc{Triple}(A \cdot 4^{i+2-F^i(1)}, F^i(1), \mathsf{a}, \mathsf{b}', \mathsf{c}', \mathsf{X}) \tag{6}$$

$$x_{B-1}(\pi) = \textsc{Triple}(A \cdot 4^{B-\widetilde{F}(B)}, \widetilde{F}(B), \mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{X}) \tag{7}$$

The individual counter values corresponding to these equations are listed in Figure 1.

We split the set of runs of $\widetilde{\mathcal{P}}$ in two parts: the *good* runs, for which we show that Equations (3)–(7) hold, and the *bad* runs, that we prove to be non $(Z \cup \{c''\})$-zeroing. Formally, we say that a run of $\widetilde{\mathcal{P}}$ is *good* if it goes through $B-1$ iteration steps; if all the loops visited along it are maximally iterated; and if all its calls to the program $\mathcal{P}$ are Z-zeroing. By opposition, we describe as *bad* the runs that fail to satisfy at least one of these conditions.

Good runs. We prove that the good runs of $\widetilde{\mathcal{P}}$ compute the function $\widetilde{F}$:

▶ **Claim 10.2.** *Let $A, B \in \mathbb{N}_+$ be two positive integers. Every good run of $\widetilde{\mathcal{P}}$ starting in* $\textsc{Triple}(A, B, \mathsf{a}, \mathsf{b}'', \mathsf{c}'', \mathsf{X} \cup \{\mathsf{b}'', \mathsf{c}''\})$ *satisfies Equations* (3)–(7)*, thus in particular it ends in* $\textsc{Triple}(A \cdot 4^{B-\widetilde{F}(B)}, \widetilde{F}(B), \mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{X} \cup \{\mathsf{b}'', \mathsf{c}''\})$*. Moreover, there exists such a run if and only if $4^{\widetilde{F}(B)-B}$ divides A.*

Proof. Let π be a good run of $\widetilde{\mathcal{P}}$ starting in $\textsc{Triple}(A, B, \mathsf{a}, \mathsf{b}'', \mathsf{c}'', \mathsf{X})$. We immediately get that Equation (3) is satisfied. To show that π satisfies Equations (4)–(7), we prove via three inductive steps that Figure 1 is an accurate depictions of the valuations $x_i(\pi)$ and $y_i(\pi)$ for every $i \in \{0, 2, \ldots, B-1\}$:

	a	b	b'	c	c'
$w_0(\pi)$:	A	0	0	0	0
$x_0(\pi)$:	A	1	0	$A \cdot 2 - \mathsf{a}$	0
$x_i(\pi)$:	$A \cdot 4^{i+1-F^i(1)}$	$F^i(1)$	0	$A \cdot 4^{i+1} - \mathsf{a}$	0
$y_i(\pi)$:	$A \cdot 4^{i+2-F^i(1)}$	0	$F^i(1)$	0	$A \cdot 4^{i+2} - \mathsf{a}$
$x_{B-1}(\pi)$:	$A \cdot 4^{B-\widetilde{F}(B)}$	$\widetilde{F}(B)$	0	$A \cdot 4^B - \mathsf{a}$	0

Figure 1 Individual counter values corresponding to the expressions in Equations (3)–(7).

1. First, starting from a valuation satisfying $\mathsf{c}'' \geq \mathsf{a}$, the effect of the initialisation step with maximal iteration of the flat loops is equivalent to the following sequence of assignments:

$$\mathsf{b}'' := \mathsf{b}'' - 1, \qquad \mathsf{b} := \mathsf{b} + 1, \qquad \mathsf{c}'' := \mathsf{c}'' - \mathsf{a}, \qquad \mathsf{c} := \mathsf{a}.$$

This maps the first row of Figure 1 to its second row, thus Equation (4) holds.

2. Next, starting from a valuation satisfying $\mathsf{c}'' \geq \mathsf{a}$, the effect of lines 5–8 of $\widetilde{\mathcal{P}}$ with maximal iteration of the flat loops is equivalent to the following sequence of assignments:

$$\mathsf{a} := 2 \cdot \mathsf{a}, \qquad \mathsf{b}' := \mathsf{b}, \qquad \mathsf{b} := 0, \qquad \mathsf{c}'' := \mathsf{c}'' - (\mathsf{a} + \mathsf{c}), \qquad \mathsf{c}' := \mathsf{c}' + 2\mathsf{c}, \qquad \mathsf{c} := 0.$$

This maps the third row of Figure 1 to its fourth row, thus whenever Equation (5) holds for some $0 \in \{1, 2, \ldots, B-2\}$, so does Equation (6).

3. Finally, as the program $\mathcal{P}$ is a strong F-amplifier, for all $i \in \{0, 1, \ldots, B-2\}$ it Z-computes $\textsc{Triple}(A \cdot 4^{i+2-F^{i+1}(1)}, F^{i+1}(1), \mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{X})$ from $\textsc{Triple}(A \cdot 4^{i+2-F^{i}(1)}, F^{i}(1), \mathsf{a}, \mathsf{b}', \mathsf{c}', \mathsf{X})$. Therefore, as every call to $\mathcal{P}$ along π is Z-zeroing since π a good run, we get that if Equation (6) holds for some $i \in \{0, 1, \ldots, B-2\}$ then Equation (5) holds for $i+1$.

To show that the run π ends in $\textsc{Triple}(A \cdot 4^{B-\widetilde{F}(B)}, \widetilde{F}(B), \mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{X} \cup \{\mathsf{b}'', \mathsf{c}''\})$, we still need to address the values of counters b'' and c'' (as Equation 7 only specifies the value of the counter set X). We directly get that the value of b'' is 0 at the end of π: b'' starts with value B and is decremented once in the initialisation step and in each of the $B-1$ occurrences of the iteration step. Moreover, we also get that c'' is 0 at the end of π since Claim 10.1 yields that the value of $\mathsf{a} + \mathsf{c} + \mathsf{t} + \Sigma\mathsf{Z} + \mathsf{c}''$ is constantly equal to $A \cdot 4^B$ along π.

Finally, concerning the existence of the run π, remark that, while the register updates mentioned in Item 1 and 2 can be applied irrespective of the values of A and B, the Z-zeroing calls to $\mathcal{P}$ described in Item 3 can be fulfilled if and only if A is divisible by a sufficiently large power of 2. More specifically, the run π described in this proof exists if and only if $4^{\widetilde{F}(B)-B}$ divides A. ◀

Bad runs. We prove that the bad runs of $\widetilde{\mathcal{P}}$ do not $(Z \cup \{c''\})$-compute anything:

▶ **Claim 10.3.** *Let $A, B \in \mathbb{N}_+$ be two positive integers. Every bad run of $\widetilde{\mathcal{P}}$ starting in $\textsc{Triple}(A, B, \mathsf{a}, \mathsf{b}'', \mathsf{c}'', \mathsf{X} \cup \{\mathsf{b}'', \mathsf{c}''\})$ is not $(Z \cup \{c''\})$-zeroing.*

Proof. Let π be a run of $\widetilde{\mathcal{P}}$ starting in $\textsc{Triple}(A, B, \mathsf{a}, \mathsf{b}'', \mathsf{c}'', \mathsf{X})$. At the start of π we have:

$$\mathsf{a} = \mathsf{A}, \quad \mathsf{b}'' = \mathsf{B}, \quad \mathsf{c}'' = \mathsf{a} \cdot (4^{\mathsf{b}''} - 1),$$

and all other counters are 0. In particular, this implies $\mathsf{b} = \mathsf{b}' = \mathsf{c} = \mathsf{c}' = \mathsf{t} = 0$, thus

$$(\mathsf{a} + \mathsf{c} + \mathsf{t} + \Sigma\mathsf{Z}) \cdot 4^{\mathsf{b}''} = \mathsf{a} + \mathsf{c} + \mathsf{t} + \Sigma\mathsf{Z} + \mathsf{c}''. \tag{8}$$

We start by observing that Claim 10.1 implies that π is Z-zeroing if and only if Equation (8) holds after every step of π and $b' = 0$ at the end of π: The right-hand of Equation (8) side is preserved, and the value of the left hand-side never increases, thus if it ever decreases it remains smaller than the right-hand side until the term of π, which in particular implies that the value of c'' is not 0.

Therefore we can immediately deduce that if π visits the iteration step less than $B-1$ times, then $b' > 0$ at the end of π, thus π is not $(Z \cup \{c''\})$-zeroing by Equation (8). For the rest of this proof, let us suppose that π visits the iteration step $B-1$ times. Whenever π goes through the initialisation step or the iteration step, it decrements b' by 1, while gaining

the opportunity to increment the sum $\mathsf{a}+\mathsf{c}+\mathsf{t}+\Sigma\mathsf{Z}$, that we denote $\Sigma\mathsf{Z}'$ in order to maintain Equation (8). As we showed when we studied the good runs, in an ideal scenario $\Sigma\mathsf{Z}'$ is quadrupled, which compensates the decrement of b', and Equation (8) still holds. We now show that the occurrence of a single mistake at any point results in $\Sigma\mathsf{Z}'$ not being quadrupled along a step: We suppose that the run π is bad, we list all the possible errors it can commit, and show that each one breaks Equation (8):

- If π fails to maximally iterate one of the flat loops at lines 1 or 2 then the sum $\Sigma\mathsf{Z}'$ is not quadrupled during the initialisation step: Maximally iterating both loops (that is, iterating both of them a times) increments $\Sigma\mathsf{Z}'$ by $3\cdot\mathsf{a}$, which exactly quadruples it since initially the other variables occurring in S have value 0. However, since line 2 increases $\Sigma\mathsf{Z}'$, not maximally iterating it results in a smaller value. Moreover, while line 1 has no direct effect on $\Sigma\mathsf{Z}'$, not maximally iterating it reduces the number of times line 2 can be iterated, which in turn reduces the value of $\Sigma\mathsf{Z}'$.
- If π fails to maximally iterate one of the flat loops at lines 5, 6 or 7 then the sum $\Sigma\mathsf{Z}'$ is not quadrupled during the corresponding iteration step: Maximally iterating the three loops (that is, iterating lines 5 and 6 a times and line 7 c times) increments $\Sigma\mathsf{Z}'$ by $3\cdot(\mathsf{a}+\mathsf{c})$, which exactly quadruples it as long as the other variables occurring in $\Sigma\mathsf{Z}'$ had value 0 to start with. However, since lines 6 and 7 increase $\Sigma\mathsf{Z}'$, not maximally iterating them results in a smaller value. Moreover, while line 5 has no direct effect on $\Sigma\mathsf{Z}'$, not maximally iterating it reduces the number of times line 6 can be iterated.
- If π does a non Z-zeroing call to the program $\mathcal{P}$, we differentiate two cases. If this happens in the last iteration step we get that π is not $(Z\cup\{c''\})$-zeroing as it is not Z-zeroing. If this happens in one of the previous iteration steps then the sum $\Sigma\mathsf{Z}'$ is not quadrupled in the *next* iterations step: as we just saw the iteration step increases S by at most $3\cdot(\mathsf{a}+\mathsf{c})$, which fails to quadruple it if there are nonzero counters in Z.
- Finally, let us consider the case where the first error committed by π is failing to maximally iterate the flat loop at line 8. In this case, we show that the subsequent call to $\mathcal{P}$ is not Z-zeroing, which, as we have just shown, implies that π is not $(Z\cup c'')$-zeroing. Since we assume that this is the first error committed by π, up to this point, π has behaved as a good run. To analyse this situation, let π' be the run that behaves as π up to this point but then maximally iterates the flat loop at line 8. By Lemma 10.2, we know that π' enters the call to $\mathcal{P}$ with a counter valuation matching $\textsc{Triple}(A\cdot 4^{i+2-F^i(1)}, F^i(1), \mathsf{a}, \mathsf{b}', \mathsf{c}', \mathsf{X})$ for some $0\leq i\leq B-1$. In particular, the following equation holds for π':

$$(\mathsf{a}+\mathsf{c}+\mathsf{t}+\Sigma\mathsf{Z})\cdot 4^{\mathsf{b}'} = A\cdot 4^{i+2} = \mathsf{a}+\mathsf{c}+\mathsf{t}+\Sigma\mathsf{Z}+\mathsf{c}'$$

  However, since π did *not* maximally iterate line 8, b' will be smaller in π compared to π' (and b will be larger - but this has no impact on the following argument since $\mathsf{b}\notin\mathsf{Z}$). Consequently, π will call the program $\mathcal{P}$ with a counter valuation satisfying:

$$(\mathsf{a}+\mathsf{c}+\mathsf{t}+\Sigma\mathsf{Z})\cdot 4^{\mathsf{b}'} < \mathsf{a}+\mathsf{c}+\mathsf{t}+\Sigma\mathsf{Z}+\mathsf{c}'.$$

  Since $\mathcal{P}$ is a strong amplifier, this equation still holds at the exit of $\mathcal{P}$. In particular, this implies that c' is not 0, thus the call to $\mathcal{P}$ is not Z-zeroing. ◀

C Proof of Lemma 10

▶ **Lemma 10.** *Let $\mathcal{Q}$ be a program simulating a strong F-amplifier $(\mathcal{P}, \mathsf{X}, (\mathsf{a},\mathsf{b},\mathsf{c}), (\mathsf{a},\mathsf{b}',\mathsf{c}'), \mathsf{t}, \mathsf{Z})$ without delegating the counters $\mathsf{a},\mathsf{b},\mathsf{c}$ and t.*

- *If $\mathcal{Q}$ delegates b' but not c', then $\widetilde{\mathcal{Q}}$ simulates $(\widetilde{\mathcal{P}}, \mathsf{X}\cup\{\mathsf{b}'',\mathsf{c}''\}, (\mathsf{a},\mathsf{b},\mathsf{c}), (\mathsf{a},\mathsf{b}'',\mathsf{c}''), \mathsf{Z}\cup\{\mathsf{c}''\})$ while delegating two input counters b'' and c'' in addition to the counters delegated by $\mathcal{Q}$.*
- *If $\mathcal{Q}$ delegates b' and c', then $\overline{\mathcal{Q}}$ simulates $(\widetilde{\mathcal{P}}, \mathsf{X}\cup\{\mathsf{b}'',\mathsf{c}''\}, (\mathsf{a},\mathsf{b},\mathsf{c}), (\mathsf{a},\mathsf{b}'',\mathsf{c}''), \mathsf{Z}\cup\{\mathsf{c}''\})$ while delegating only one input counter b'' in addition to the counters delegated by $\mathcal{Q}$.*

Proof. Let X and S denote the set of counters of $\mathcal{Q}$, respectively its stack alphabet. Let $\widetilde{h}$ denote the function $h_{\mathsf{X},\mathsf{S}\cup\{\mathsf{b}'',\mathsf{c}''\}}$ that transforms configurations of $\widetilde{\mathcal{Q}}$ into configurations of $\widetilde{\mathcal{P}}$. Similarly, let $\overline{h}$ denote the function $h_{\mathsf{X}\cup\{\mathsf{c}''\},\mathsf{S}\cup\{\mathsf{b}''\}}$ that transforms configurations of $\overline{\mathcal{Q}}$ into configurations of $\widetilde{\mathcal{P}}$. The proof is done in three steps. First, we show that we can easily translate the runs of $\widetilde{\mathcal{Q}}$ and $\overline{\mathcal{Q}}$ into runs of $\widetilde{\mathcal{P}}$ with matching source and target. The harder part of the proof is to show the reciprocal statement: we consider the good runs of $\widetilde{\mathcal{P}}$ described in Lemma 10.2 and we show how to translate them, first into runs of $\widetilde{\mathcal{Q}}$, and finally into runs of $\overline{\mathcal{Q}}$.

Transforming runs of $\widetilde{\mathcal{Q}}$ and $\overline{\mathcal{Q}}$ into runs of $\widetilde{\mathcal{P}}$. The program $\widetilde{\mathcal{Q}}$ is a constrained version of $\widetilde{\mathcal{P}}$: every line is identical except for the lines with a popping instruction instead of a decrement, which are more restrictive since the correct symbol needs to be at the top of the stack. As a consequence, for every run $\widetilde{\pi}$ of $\widetilde{\mathcal{Q}}$ between two configurations x and y we immediately get a run π of $\widetilde{\mathcal{P}}$ between $\widetilde{h}(x)$ and $\widetilde{h}(y)$ which uses the same lines as $\widetilde{\pi}$.

Now given a run of $\overline{\pi}$ of $\overline{\mathcal{Q}}$ between two configurations x and y, translating $\overline{\pi}$ into a run of $\widetilde{\mathcal{P}}$ is not as direct since the lines 7–16 of $\overline{\mathcal{Q}}$ are not exactly analogous to the lines 7–8 of $\widetilde{\mathcal{P}}$. However, as we explained in the paper, using some local reshuffling $\widetilde{\mathcal{P}}$ can reproduce any counter update corresponding to the lines 7–16 of $\overline{\mathcal{Q}}$. This allows us to transform the run $\overline{\pi}$ into a run of $\widetilde{\mathcal{P}}$ between $\overline{h}(x)$ and $\overline{h}(y)$.

Transforming runs of $\widetilde{\mathcal{P}}$ into runs of $\widetilde{\mathcal{Q}}$. Let us suppose that $\mathcal{Q}$ delegates the counter b' but not c', and let π be a *good* run of $\widetilde{\mathcal{P}}$ as described in the proof of Lemma 10.2. We denote by x and y the starting and ending configuration of π. In order to transfer π to $\widetilde{\mathcal{Q}}$, we begin by creating an appropriate initial configuration x' as in the proof of Lemma 9. Formally, let $u_\pi \in \{\mathsf{b}',\mathsf{c}'\}^*$ be the word listing, in order, the occurrences of the decrements of b'' and c'' along π. We define the configuration x' of $\widetilde{\mathcal{Q}}$ by setting the values of the counters of X to the values they have in the starting configuration x of π, and setting the stack content to the reverse of the word u_π (so that the first letter of u_π is at the top of the stack). This definition guarantees that $h(x') = x$. To conclude, we need to argue that $\widetilde{\mathcal{Q}}$ can simulate π starting from x'. First, remark that the initialisation step is easily simulated since the definition of the initial stack content guarantees that the appropriate symbol is at the top of the stack whenever needed. We now explain, step by step, how $\widetilde{\mathcal{Q}}$ simulates the iteration steps of π. First, thanks to the definition of the initial stack content the loops on lines 5–7 can be iterated as in π. Then, we also iterate line 8 as in π. Remark that this disrupts the stack by adding a sequence of b' on top of it. Next comes the call to $\mathcal{P}$, and since π is a good run we know that this call is *correct*, in the sense that it starts in $\textsc{Triple}(A, B, \mathsf{a}, \mathsf{b}', \mathsf{c}', \mathsf{X})$ (Equation (6)) and ends in $\textsc{Triple}(A \cdot 4^{B-\widetilde{F}(B)}, \widetilde{F}(B), \mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{X})$ (Equation (5)) for some $A, B \in \mathbb{N}_+$. Therefore in $\widetilde{\mathcal{Q}}$ we can simulate this call to $\mathcal{P}$ by a call to $\mathcal{Q}$, and since the value of b' is 0 in the ending configuration this implies that the call to $\mathcal{Q}$ will automatically pop all of the b' that were added on the stack. Therefore we are back with a stack content that matches a prefix of our initial stack content, and we can conclude the simulation of the iteration step by popping a single b'' from the stack.

Transforming runs of $\widetilde{\mathcal{P}}$ into runs of $\overline{\mathcal{Q}}$. Let us suppose that $\mathcal{Q}$ delegates both b' and c', and let π be a *good* run of $\widetilde{\mathcal{P}}$, as described in the proof of Lemma 10.2. We denote by x and y the starting and ending configuration of π. We show how to construct a run $\overline{\pi}$ of $\overline{\mathcal{Q}}$ that simulates π. First, remark that we have a single possibility for the starting configuration of $\overline{\pi}$: In the starting configuration of π only the values of a, b'' and c'' are nonzero (Equation (3)), and $\overline{\mathcal{Q}}$ only delegates b'' among these three counters. Therefore the initial stack content will just be a sequence of b'' of the appropriate length. Then, simulating the initialisation step of π is easy: one b'' is popped from the stack and the other counters are updated as in π.

To conclude, we show how to simulate the iteration steps visited by π. Let $\pi_1\pi_2\pi_3$ be a subrun of π corresponding to an iteration step of $\widetilde{\mathcal{P}}$, where π_2 stands for the call to the program $\mathcal{P}$. As we showed in the proof of Lemma 10.2, every call to $\mathcal{P}$ along π is *correct*, in the sense that it starts in some configuration $\textsc{Triple}(A, B, \mathsf{a}, \mathsf{b}', \mathsf{c}', \mathsf{X})$ (Equation (6)) and ends in $\textsc{Triple}(A \cdot 4^{B - \widetilde{F}(B)}, \widetilde{F}(B), \mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{X})$ for some $A, B \in \mathbb{N}_+$ (Equation (5)). As a consequence, since $\mathcal{Q}$ simulates $\mathcal{P}$, there exists a run $\overline{\pi}_2$ of $\mathcal{Q}$ that simulates π_2, but this run requires a starting stack content corresponding to some specific shuffle u_{π_2} of the word $(\mathsf{b}')^{\mathsf{B}}(\mathsf{c}')^{\mathsf{A}\cdot(4^{\mathsf{B}}-1)}$. Fortunately, as we explained in the paper, the lines 7–16 of $\overline{\mathcal{Q}}$ allow to push any shuffle of b' and c' on the stack. In particular, there exists a subrun $\overline{\pi}_1$ of $\overline{\mathcal{Q}}$ that simulates π_1 and pushes the word u_{π_2} on the stack. As a consequence, $\overline{\pi}_1\overline{\pi}_2$ simulates truthfully the subrun $\pi_1\pi_2$ with no impact on the stack: $\overline{\pi}_1$ pushes u_{π_2}, which is then popped by $\overline{\pi}_2$. Therefore, we can simulate the iteration step $\pi_1\pi_2\pi_3$ by starting with $\overline{\pi}_1\overline{\pi}_2$, and then adding $\overline{\pi}_3$ which pops a single b from the stack to simulate the decrement of b occurring in π_3. ◀