What Makes Petri Nets Harder to Verify: Stack or Data?[⋆]

Ranko Lazić[1] and Patrick Totzke[2]

[1] DIMAP, Department of Computer Science, University of Warwick, UK
[2] LFCS, School of Informatics, University of Edinburgh, UK

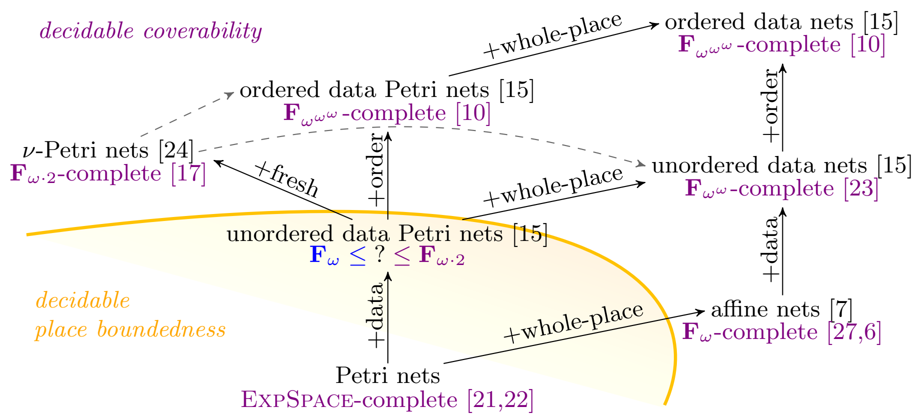
Abstract. We show how the yardstick construction of Stockmeyer, also developed as counter bootstrapping by Lipton, can be adapted and extended to obtain new lower bounds for the coverability problem for two prominent classes of systems based on Petri nets: Ackermann-hardness for unordered data Petri nets, and Tower-hardness for pushdown vector addition systems.

1 Introduction

Unordered Data Petri Nets (UDPN [15]) extend Petri nets by decorating tokens with data values taken from some countable data domain $\mathbb{D}$, broadly in the vein of coloured Petri nets [13]. These values act as pure names: they can only be compared for equality or non-equality upon firing transitions. Such systems can model for instance distributed protocols where process identities need to be taken into account [24]. UDPNs also coincide with the natural generalisation of Petri nets in the framework of sets with atoms [3]. In spite of their high expressiveness, UDPNs fit in the large family of Petri net extensions among the *well-structured* ones [1,8]. As such, they still enjoy decision procedures for several verification problems, prominently safety through the *coverability* problem.

UDPNs have an interesting position in the taxonomy of well-structured Petri net extensions (see Figure 1). Indeed, all their extensions forgo the decidability of the *reachability* problem (whether a target configuration is reachable) and of the *place boundedness* problem (whether the number of tokens in a given place can be bounded along all runs): this is the case of *ν-Petri nets* [24] that allow to create fresh data values, of *ordered data Petri nets* [15] that posit a dense linear ordering on $\mathbb{D}$, and of *unordered data nets* [15] that allow to perform 'whole-place' operations, which move and/or duplicate all the tokens from a place to another. By contrast, it is currently open whether reachability is decidable for UDPNs, but recent results on computing their coverability trees [11] and on linear combinations of unordered data vectors [12] suggest to conjecture decidability.

⋆ Supported by the EPSRC, grants EP/M011801/1 and EP/M027651/1, and by the Royal Society, grant IE150122.



decidable coverability
ordered data nets [15]
$\mathbf{F}_{\omega^{\omega^\omega}}$-complete [10]
+whole-place
ordered data Petri nets [15]
$\mathbf{F}_{\omega^{\omega^\omega}}$-complete [10]
+order
ν-Petri nets [24]
$\mathbf{F}_{\omega\cdot 2}$-complete [17]
+fresh
+order
unordered data nets [15]
$\mathbf{F}_{\omega^\omega}$-complete [23]
+whole-place
unordered data Petri nets [15]
$\mathbf{F}_\omega \leq ? \leq \mathbf{F}_{\omega\cdot 2}$
+data
decidable
place boundedness
+data
+whole-place
affine nets [7]
$\mathbf{F}_\omega$-complete [27,6]
Petri nets
ExpSpace-complete [21,22]

Fig. 1. A short taxonomy of some well-structured extensions of Petri nets. The complexities refer to the coverability problems, and can be taken as proxies for expressiveness; the new lower bound in this paper is displayed in blue, and the exact complexity for UDPNs remains open. Place boundedness is decidable below the yellow line and undecidable above. As indicated by the dashed arrows, freshness can be enforced using a dense linear order or whole-place operations.

The Power of Well-Structured Systems. This work is part of a general programme that aims to understand the expressive power and algorithmic complexity of well-structured transition systems (WSTS), for which the complexity of the coverability problem is a natural proxy. Besides the intellectual satisfaction one might find in classifying the worst-case complexity of this problem, we hope indeed to gain new insights into the algorithmics of the systems at hand, and into their relative 'power'. A difficulty is that the generic *backward* algorithm [1,8] developed to solve coverability in WSTS relies on well-quasi-orders (wqos), for which complexity analysis techniques are not so widely known.

Nevertheless, in a series of recent papers, the exact complexity of coverability for several classes of WSTSs has been established. These complexities are expressed using ordinal-indexed *fast-growing* complexity classes $(\mathbf{F}_\alpha)_\alpha$ [25], e.g. Tower complexity corresponds to the class $\mathbf{F}_3$ and is the first non elementary complexity class in this hierarchy, Ackermann corresponds to $\mathbf{F}_\omega$ and is the first non primitive-recursive class, hyper-Ackermann to $\mathbf{F}_{\omega^\omega}$ and is the first non multiply-recursive class, etc.; see Figure 2. To cite a few of these complexity results, coverability is $\mathbf{F}_\omega$-complete for reset Petri nets and affine nets [27,6], $\mathbf{F}_{\omega\cdot 2}$-complete for ν-Petri nets [17], $\mathbf{F}_{\omega^\omega}$-complete for lossy channel systems [4,26] and unordered data nets [23], and even higher complexities appear for timed-arc Petri nets and ordered data Petri nets ($\mathbf{F}_{\omega^{\omega^\omega}}$-complete [10]) and priority channel systems and nested counter systems ($\mathbf{F}_{\varepsilon_0}$-complete [9,5]); see the complexities in violet in Figure 1 for the Petri net extensions related to UDPNs.

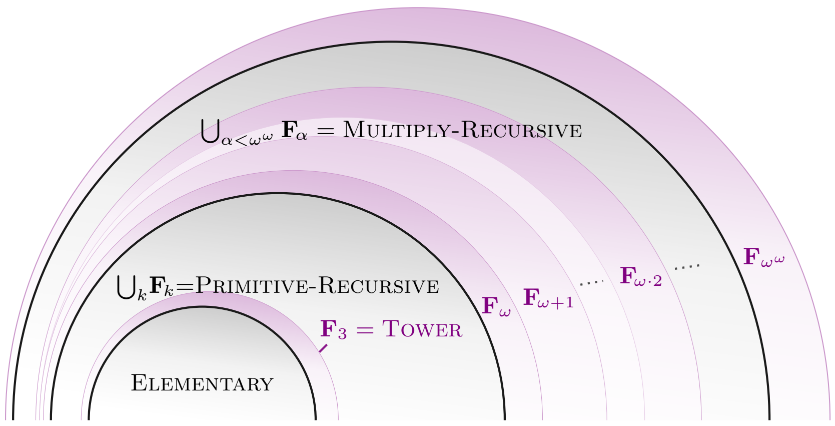

$\bigcup_{\alpha<\omega^\omega} \mathbf{F}_\alpha$ = Multiply-Recursive
$\bigcup_k \mathbf{F}_k$=Primitive-Recursive
$\mathbf{F}_3$ = Tower
Elementary
$\mathbf{F}_\omega$
$\mathbf{F}_{\omega+1}$
$\mathbf{F}_{\omega\cdot 2}$
$\mathbf{F}_{\omega^\omega}$

Fig. 2. Some complexity classes beyond Elementary. The two new lower bounds in this paper are $\mathbf{F}_3$ (Section 3) and $\mathbf{F}_\omega$ (Section 7), whereas the best known upper bound for UDPN coverability is $\mathbf{F}_{\omega\cdot 2}$ [17].

New Lower Bound for UDPNs. In this paper, we tackle the huge gap in what is known about the worst-case complexity of the coverability problem for UDPNs: between the Tower, i.e. $\mathbf{F}_3$, lower bound established in [15] and the $\mathbf{F}_{\omega\cdot 2}$ upper bound that holds even for the more general class of ν-Petri nets [17]. Our main result is an increased $\mathbf{F}_\omega$ lower bound, making it known that coverability for UDPNs cannot be decided in primitive recursive time or space.

For this Ackermann lower bound, we follow the pattern of Stockmeyer's yardstick construction [28] and Lipton's classical proof of ExpSpace-hardness for Petri nets [21], in that we design an 'object-oriented' implementation of the Ackermann function in UDPNs. By this, we mean that the implementation provides an interface with increment, decrement, zero, and max operations on larger and larger *counters* up to an Ackermannian value. This allows then the simulation of a Minsky machine working in Ackermann space.

The difficulty is that the bootstrapping implementation in UDPNs of the hierarchy of counters requires an iteration operator on 'counter libraries'—of the kind employed recently in the context of channel systems with insertion errors to obtain $\mathbf{F}_\omega$-hardness [16] and in the context of ν-Petri nets to obtain $\mathbf{F}_{\omega\cdot 2}$-hardness [17]—but UDPNs have fundamentally unordered configurations as well as no basic mechanism for creating fresh data values. To overcome that obstacle—and this is the key technical idea in the paper—we enrich the interfaces of the counter implementations by a *local freshness test*: verifying that a given data value is distinct from all data values that the implementation (and, recursively, the implementations of all lesser counters in the hierarchy) currently employs internally; see Sections 5 and 6.

Pushdown Vector Addition Systems. Motivations for considering extensions of Petri nets by a pushdown stack include verifying procedural programs with unbounded integer variables [2] and optimising two-variable queries on XML documents with unbounded data at leaf nodes [14]. The boundedness problem, as well as the coverability and counter-boundedness problems in the restricted setting of pushdown vector addition systems (PVAS) of dimension 1, have recently been shown decidable [18,20,19]. However, the coverability and reachability problems are interreducible for PVASs in general [14,20], and the whether they are decidable remains a challenging open question.

Partly in order to introduce the bootstrapping technique that is the basis of our $\mathbf{F}_\omega$ lower bound for UDPN coverability, we present a proof that the reachability problem for PVASs (also the coverability and boundedness problems) is $\mathbf{F}_3$-hard; this TOWER lower bound means that the latter problems cannot be decided in elementary time or space.

Outline. In the two sections, we introduce the bootstrapping technique using pushdown vector addition systems and obtain TOWER-hardness of their reachability problem. The four sections that follow develop the more involved ACKERMANN-hardness of the coverability problem for unordered data Petri nets. The latter lower bound still leaves a gap to the best known $\mathbf{F}_{\omega\cdot 2}$ upper bound, and we finish with some remarks about that in the concluding section.

2 Pushdown Vector Addition Systems

It is convenient for our purposes to formalise PVASs as programs that operate on non-negative counters and a finite-alphabet stack. More precisely, we define them as finite sequences of commands which may be labelled, where a command is one of:

– an increment of a counter ($x := x + 1$),
– a decrement of a counter ($x := x - 1$),
– a push (push a),
– a pop (pop a),
– a nondeterministic jump to one of two labelled commands (goto L or L'),
– or termination (halt).

Initially, all counters have value 0 and the stack is empty. Whenever a decrement of a counter with value 0 or an erroneous pop is attempted, the program aborts. In every program, halt occurs only as the last command.

Example 2.1. We display in Figure 3 a PVAS fragment, which will be useful in the next section. It is shown diagramatically, where multiple outgoing edges from a node are to be implemented by the nondeterministic jumps.

The reachability problem for PVASs can now be stated as follows:

Input: A PVAS $\mathcal{P}$.
Question: Does $\mathcal{P}$ have a computation which reaches the `halt` command with all counters being 0 and the stack being empty?

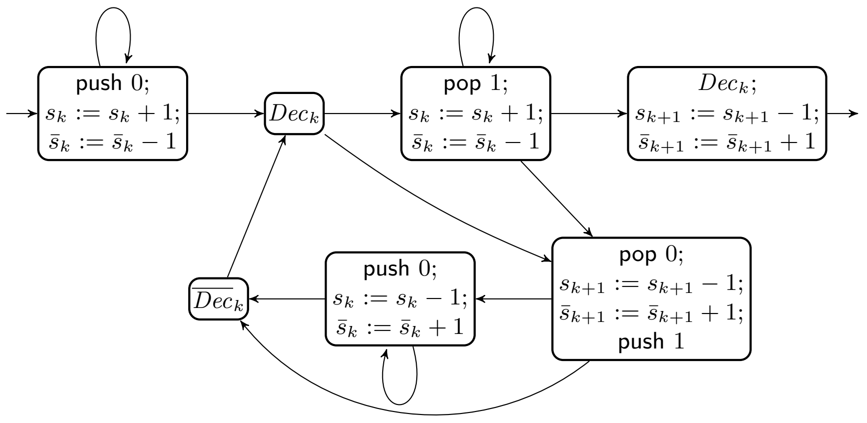

push 0; $s_k := s_k + 1$; $\bar{s}_k := \bar{s}_k - 1$
Dec_k
pop 1; $s_k := s_k + 1$; $\bar{s}_k := \bar{s}_k - 1$
Dec_k; $s_{k+1} := s_{k+1} - 1$; $\bar{s}_{k+1} := \bar{s}_{k+1} + 1$
pop 0; $s_{k+1} := s_{k+1} - 1$; $\bar{s}_{k+1} := \bar{s}_{k+1} + 1$; push 1
push 0; $s_k := s_k - 1$; $\bar{s}_k := \bar{s}_k + 1$
$\overline{Dec_k}$

Fig. 3. PVAS procedure Dec_{k+1}. The calls of procedures Dec_k and $\overline{Dec_k}$ use the stack in the standard manner. The latter is the variant of Dec_k that decrements $\bar{s}_k$ exactly tower(k) times, i.e. with s_k and $\bar{s}_k$ swapped.

3 Tower-Hardness

Theorem 3.1. *The reachability problem for PVASs is* TOWER*-hard.*

Proof. We reduce from the tower(n)-bounded halting problem for *Minsky programs* with n commands, where:

– for $k \in \mathbb{N}$, the 'tetration' operation is defined by

$$\text{tower}(0) = 1 \text{ and } \text{tower}(k+1) = 2^{\text{tower}(k)} \ ;$$

– the Minsky programs are defined like PVASs, except that they have no stack, have only deterministic jumps (goto L), but can test counters for zero (if $x = 0$ then L else L').

The following problem is TOWER-hard [25, Section 2.3.2 and Theorem 4.1]:

Input: A Minsky program $\mathcal{M}$ with n commands.
Question: Can $\mathcal{M}$ reach the halt command by a computation during which all counter values are at most tower(n)?

Given such a Minsky program $\mathcal{M}$, we construct in time polynomial in n a PVAS $\mathcal{P}(\mathcal{M})$ that simulates $\mathcal{M}$ as long as its counters do not exceed tower(n). Similarly to Stockmeyer's yardstick construction [28] and Lipton's proof of EXPSPACE-hardness for Petri nets [21], the idea is to bootstrap the ability to simulate zero tests of counters that are bounded by tower(1), tower(2), …, tower(n).

More precisely, for each counter x of $\mathcal{M}$, $\mathcal{P}(\mathcal{M})$ have a pair of counters x and $\bar{x}$, on which it maintains the invariant $x + \bar{x} = \text{tower}(n)$. Thus, every increment of x in $\mathcal{M}$ is translated to $x := x + 1;\ \bar{x} := \bar{x} - 1$ in $\mathcal{P}(\mathcal{M})$, and similarly for decrements.

For every zero test of x in $\mathcal{M}$, $\mathcal{P}(\mathcal{M})$ uses auxiliary counters s_n and $\bar{s}_n$, for which it also maintains $s_n + \bar{s}_n = \text{tower}(n)$. Moreover, we assume that $s_n = 0$ at the start of each zero-test simulation. The simulation begins by $\mathcal{P}(\mathcal{M})$ transferring some part of $\bar{x}$ to s_n (while preserving the invariants). It then calls a procedure Dec_n that decrements s_n exactly $\text{tower}(n)$ times. For the latter to be possible, x must have been 0. Otherwise, or in case not all of $\bar{x}$ was transferred to s_n, the procedure can only abort. When Dec_n succeeds, the initial values of x and $\bar{x}$ are reversed, so to finish the simulation, everything is repeated with x and $\bar{x}$ swapped.

The main part of the construction is implementing Dec_k for $k = 1, 2, \ldots, n$. Assuming that Dec_k which decrements s_k exactly $\text{tower}(k)$ times and maintains $s_k + \bar{s}_k = \text{tower}(k)$ has been implemented for some $k < n$, Dec_{k+1} consists of performing the following by means of s_k, $\bar{s}_k$ and Dec_k, cf. Figure 3:

– push exactly $\text{tower}(k)$ zeros onto the stack;
– keep incrementing the $\text{tower}(k)$-digit binary number that is on top of the stack until no longer possible, and decrement s_{k+1} for each such increment;
– pop $\text{tower}(k)$ ones that are on top of the stack, and decrement s_{k+1} once more.

Following the same pattern: starting with all counters having value 0, $\mathcal{P}(\mathcal{M})$ can initialise each auxiliary counter $\bar{s}_k$ to $\text{tower}(k)$, and each $\bar{x}$ to $\text{tower}(n)$; also provided $\mathcal{M}$ reaches its halt command, $\mathcal{P}(\mathcal{M})$ can empty all its counters, as required. □

4 Unordered Data Petri Nets

This extension of classical Petri nets is by decorating tokens with data values taken from some countably infinite data domain $\mathbb{D}$. These values act as pure names: they can only be compared for equality or non-equality upon firing transitions. We recall the definition from [24,23].

A multiset over set X is a function $M : X \to \mathbb{N}$. The set $X^{\oplus}$ of all multisets over X is ordered pointwise, and the union of $M, M' \in X^{\oplus}$ is $(M \oplus M') \in X^{\oplus}$ with $(M \oplus M')(\alpha) \overset{\text{def}}{=} M(\alpha) + M'(\alpha)$ for all $\alpha \in X$. If $M \geq M'$ holds then the difference $(M \ominus M')$ is defined as the unique $M'' \in X^{\oplus}$ with $M = M' \oplus M''$.

Definition 4.1. *An unordered data Petri net (UDPN) over domain $\mathbb{D}$ consists of finite sets P, T, Var of* places, transitions *and* variables, *respectively, and a* flow *function $F : (P \times T) \cup (T \times P) \to \mathit{Var}^{\oplus}$ that assigns each place $p \in P$ and transition $t \in T$ a finite multiset of variables.*

A marking *is a function $M : P \to \mathbb{D}^{\oplus}$. Intuitively, $M(p)(\alpha)$ denotes the number of tokens of type α in place p. A transition $t \in T$ is* enabled *in marking*

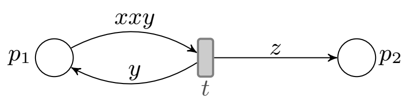


Fig. 4. An UDPN with places p_1, p_2, variables x, y, z and a single transition t. The transition t takes $\langle 2, 0\rangle$ tokens of type x and $\langle 1, 0\rangle$ tokens of type y in places $\langle p_1, p_2\rangle$. It puts 1 token of type z onto p_2 and 1 token of type y onto p_1.

M with mode σ *if $\sigma : Var \to \mathbb{D}$ is an injection such that $\sigma(F(p,t)) \leq M(p)$ for all $p \in P$. There is a step $M \to M'$ between markings M and M' if there exists t and σ such that t is enabled in M with mode σ, and for all $p \in P$,*

$$M'(p) = M(p) \ominus \sigma(F(p,t)) \oplus \sigma(F(t,p)) \ .$$

For notational convenience we will sometimes write that a marking M has tokens $\langle n_1, n_2, \ldots, n_k\rangle$ of type α in places $\langle p_1, p_2, \ldots, p_k\rangle$ *if $M(p_i)(\alpha) = n_i$ holds for all $1 \leq i \leq k$. Similarly, we write that a transition t takes (resp. puts) $\langle n_1, n_2, \ldots, n_k\rangle$ tokens of type α in places $\langle p_1, p_2, \ldots, p_k\rangle$ if for all $1 \leq i \leq k$ it holds that $F(p_i, t)(\alpha) = n_i$ (resp. $F(t, p_i)(\alpha) = n_i$).*

Notice that UDPN are a generalization of ordinary P/T nets, which have only one type of token, i.e. $\mathbb{D} = \{\bullet\}$. See Figure 4 for a depiction of an UDPN in the usual Petri net notation.

The *Coverability Problem* for UDPN is the following decision problem where $\xrightarrow{*}$ denotes the transitive and reflexive closure of the step relation.

Input: An UDPN (P, T, Var, F) and two markings $I, F : P \to \mathbb{D}^{\oplus}$.
Question: Does there exist a marking $F' \geq F$ such that $I \xrightarrow{*} F'$?

The following example shows that three places and a simple addressing mechanism are enough to simulate ordinary Petri nets with an arbitrary number of places. This suggests that UDPN are more succinct than Petri nets and indeed, as we shall see in Section 6, UDPN can be used to design more involved addressing mechanisms. This will allow us to push the classical approach of [21] to simulate bounded counter machines from a double exponential bound to an Ackermannian bound.

Example 4.2. Given a Petri net with places $P = \{p_0, \ldots, p_{n-1}\}$, we build a UDPN with three places a, $\bar{a}$, and v and variables $Var = \{x_0, \ldots, x_{n-1}\}$.

The intuition is for a and $\bar{a}$ to maintain an *addressing* mechanism for the original places in P, while v maintains the actual token counts of the original net. The places a and $\bar{a}$ hold $n-1$ different data values such that all reachable configurations are of the form $\bigoplus_{i=0}^{n-1} M_i$ where $M_i(a) = i$ and $M_i(\bar{a}) = n-1-i$ for all $0 \leq i < n$.

Each partial marking M_i represents a marking of the place p_i in the original net by holding in $M_i(v)$ the number of tokens in place p_i. Each transition of the original net translates into a UDPN transition where the flows of the variables

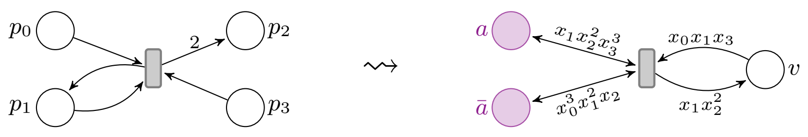


Fig. 5. Simulation of a Petri net transition (left) by a UDPN (right).

with places a and $\bar{a}$ identify uniquely the places of the original net, while the flows with place v update the token counts accordingly.

Figure 5 shows how a transition of a Petri net with 4 places (on the left) is simulated with this construction (on the right).

5 Counter Libraries in UDPNs

To present our lower bound construction, we indirectly describe UDPNs in terms of sequential programs. For this purpose we will now develop a simple but convenient language for programming with UDPNs.

Routines, Libraries, and Programs. Let a *library* mean a sequence of named routines

$$\ell_1 : R_1, \; \ldots \;, \ell_K : R_K$$

where $\ell_1, \ldots, \ell_K$ are pairwise distinct labels. In turn, a *routine* is a sequence of commands $c_1, \ldots, c_{K'}$, where the last command $c_{K'}$ is return and each c_i for $i < K'$ is one of the following:

- a UDPN transition,
- a nondeterministic jump goto G for a nonempty subset G of $\{1, \ldots, K'\}$, or
- a subroutine invocation call ℓ'.

When a library contains no subroutine calls, we say it is a *program*. The denotation of a program L is a UDPN $\mathcal{N}(L)$ constructed so that:

- The places of $\mathcal{N}(L)$ are all the places that occur in transition commands of L, and four special places p, $\overline{p}$, p', $\overline{p}'$. Places $\langle p, \overline{p}\rangle$ are used to store the pair of numbers $\langle i, K - i\rangle$ where $\ell_i : R_i$ is the routine being executed, and then places $\langle p', \overline{p}'\rangle$ to store the pair of numbers $\langle i', K' - i'\rangle$ where i' is the current line number in routine R_i and K' is the maximum number of lines in any $R_1, \ldots, R_K$.
- Each transition of $\mathcal{N}(L)$ either executes a transition command $c_{i'}$ inside some R_i ensuring that $\langle p, \overline{p}\rangle$ contains $\langle i, K - i\rangle$ and modifying the contents of $\langle p', \overline{p}'\rangle$ from $\langle i', K' - i'\rangle$ to $\langle i' + 1, K' - (i' + 1)\rangle$, or similarly executes a nondeterministic jump command.

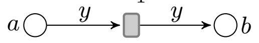
We shall refer to the special places p, $\overline{p}$, p', $\overline{p}'$ as *control places*, to the rest as *tape places*, and to markings of the latter places as *tape contents*.

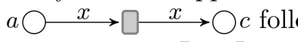
For two tape contents M and M', we say that a routine $\ell_i : R_i$ in a program L *can compute* M' from M if and only if $\mathcal{N}(L)$ can reach in finitely many steps M' with the control at the last line of R_i from M with the control at the first line of R_i. When $\ell_i : R_i$ cannot compute any M' from M, we say that it *cannot terminate* from initial tape content M.

Note that there are two sources of nondeterminism in routine computations: choosing how to instantiate the variables in the commands that are UDPN transitions, and resolving the destinations of the jump commands. The computations can also become blocked, which happens if they reach a UDPN transition that is disabled due to insufficient tokens being available in the current tape content.

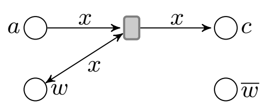
Interfaces and Compositions of Libraries. For a library L, let us write $\Lambda_{\text{in}}(L)$ and $\Lambda_{\text{out}}(L)$ for the set of all routine labels that are invoked in L and provided by L, respectively. We say that libraries L_0 and L_1 are *compatible* if and only if $\Lambda_{\text{in}}(L_0)$ is contained in $\Lambda_{\text{out}}(L_1)$. In that case, we can compose them to produce a library $L_0 \circ L_1$ in which tape contents of L_1 persist between successive invocations of its routines, as follows:

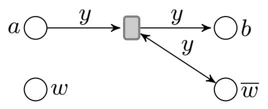
- $\Lambda_{\text{in}}(L_0 \circ L_1) = \Lambda_{\text{in}}(L_1)$ and $\Lambda_{\text{out}}(L_0 \circ L_1) = \Lambda_{\text{out}}(L_0)$.
- $L_0 \circ L_1$ has an additional place w used to store the name space of L_0 (i.e., for each name manipulated by L_0, one token labelled by it) and an additional place $\overline{w}$ for the same purpose for L_1.
- For each routine $\ell : R$ of L_0, the corresponding routine $\ell : R \circ L_1$ of $L_0 \circ L_1$ is obtained by ensuring that the transition commands in R (resp., L_1) maintain the name space stored on place w (resp., $\overline{w}$), and then inlining the subroutine calls in R.

Example 5.1. Suppose that in L_1 the routine with label ℓ_1 consists only of the transition command 

followed by return. Suppose further that L_0 has a routine with label ℓ_0 and commands 

followed by call ℓ. Then the corresponding routine ℓ_0 in the composition $L_0 \circ L_1$ is



followed by


.

Notice that, in the above definition of $\mathcal{N}(L_0 \circ L_1)$, the places of $\mathcal{N}(L_0)$ and $\mathcal{N}(L_1)$ are not duplicated: a transion command in L_0 may operate on some place which is also used in L_1. The name space places w and $\overline{w}$ and the way transition commands translate into actual UDPN transitions in $\mathcal{N}(L_0 \circ L_1)$ ensure that the commands of the two libraries do not interfere. However, this relies on an additional mechanism for preventing the same name to be used by both L_0 and L_1—unless disjointness of corresponding places is guaranteed—and that is what the local freshness checks developed in the sequel provide.

Counter Libraries. We aim to write programs that simulate bounded two-counter Minsky machines. For this purpose we will now focus on libraries that provide the necessary operations to manipulate a pair of counters. Letting Γ denote the set of labels of operations

$$\Gamma \stackrel{\text{def}}{=} \{\mathit{init}, \mathit{fresh}, \mathit{eq}, i.\mathit{inc}, i.\mathit{dec}, i.\mathit{iszero}, i.\mathit{ismax} : i \in \{1,2\}\} \ ,$$

we regard L to be a *counter* library if and only if $\Lambda_{\text{out}}(L) = \Gamma$ and $\Lambda_{\text{in}}(L) \subseteq \Gamma$.

When L is also a program, and N is a positive integer, we say that L is N-*correct* if and only if the routines behave as expected with respect to the bound N and correctly implement a freshness test on a special tape place ν. Namely, for every tape content M that can be computed from the empty tape content by a sequence σ of operations from Γ, provided *init* occurs only as the first element of σ, every routine in $\Gamma \setminus \{\mathit{init}\}$ either does not terminate or computes a unique tape content from M. If n_i is the difference between the numbers of occurrences in σ of $i.\mathit{inc}$ and $i.\mathit{dec}$, we must have for both $i \in \{1,2\}$:

- $i.\mathit{inc}$ can terminate from M if and only if $n_i < N-1$;
- $i.\mathit{dec}$ can terminate from M if and only if $n_i > 0$;
- eq can terminate from M (and compute M) if and only if $n_1 = n_2$;
- $i.\mathit{iszero}$ can terminate from M (and compute M) if and only if $n_i = 0$;
- $i.\mathit{ismax}$ can terminate from M (and compute M) if and only if $n_i = N-1$.

Moreover, N-correctness requires that L behaves with respect to *fresh* and ν so that:

- only transition commands in the routines *init* and *fresh* use the place ν;
- if M is computed from the empty tape content by *init*, then M has no tokens on place ν;
- for every tape content A that has one token of type α on place ν and is otherwise empty, and for every tape content M computed by a sequence σ as above, we have that *fresh* can terminate from $M \oplus A$ (and compute $M \oplus A$) if and only if α is not in the support of $M(p)$ for all places $p \neq \nu$.

We also need a notion of correctness for counter libraries that may not be programs, i.e. may invoke operations on another pair of counters (which we call *auxiliary*). Given a counter library L, and given a function $F : \mathbb{N}^+ \to \mathbb{N}^+$, we say that L is F-*correct* if and only if, for all N-correct counter programs C, the program $L \circ C$ is $F(N)$-correct.

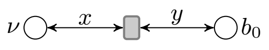
We now present two example counter libraries, which will be used in our later constructions.

Example 5.2 (An Enumerated Counter Program). For every positive integer N, one can implement a pair of N-bounded counters by manipulating the values and their complements directly as follows. Let $\mathit{Enum}(N)$ be the counter program which uses four places e_1, $\overline{e}_1$, e_2, $\overline{e}_2$ and such that for both $i \in \{1,2\}$:

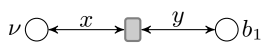
- routine *init* chooses a datum β, and puts $N-1$ tokens onto $\overline{e}_1$ and $N-1$ tokens onto $\overline{e}_2$, all carrying β;

1: call *fresh*

2: goto $\{3, 5\}$

3: 

4: goto $\{6\}$

5: 

6: return

Fig. 6. The routine *fresh* of the counter library *Double*. Here, the numbers on the left of the commands are line numbers to be referenced in goto commands.

- routine *fresh* takes one token from e_1 or $\overline{e}_1$, and checks it for inequality with the token on ν;
- routine *eq* guesses $n \in \{0, \ldots, N-1\}$, takes $\langle n, N-1-n, n, N-1-n\rangle$ tokens from places $\langle e_1, \overline{e}_1, e_2, \overline{e}_2\rangle$, and then puts them back;
- routine $i.inc$ moves a token from $\overline{e}_i$ to e_i;
- routine $i.dec$ moves a token from e_i to $\overline{e}_i$;
- routine $i.iszero$ takes $N-1$ tokens from place $\overline{e}_i$ and then puts them back;
- routine $i.ismax$ takes $N-1$ tokens from place e_i and then puts them back.

It is simple to verify that $Enum(N)$ is computable in space logarithmic in N, and that:

Lemma 5.3. *For every N, the counter program $Enum(N)$ is N-correct.*

Example 5.4 (A Counter Library for Doubling). Let *Double* be a counter library which uses four places b_1, $\overline{b}_1$, b_2, $\overline{b}_2$, is such that:

- routine *init* first initialises the auxiliary counters (call *init*), then chooses a datum β and checks that it is fresh with respect to the auxiliary counters (call *fresh*), and finally puts one token carrying β onto both $\overline{b}_1$ and $\overline{b}_2$;
- routine *fresh* checks that the given datum (on the special place ν) is both fresh with respect to the auxiliary counters, and distinct from the datum on b_1 or $\overline{b}_1$ (equivalently, b_2 or $\overline{b}_2$), see Figure 6 for the code that implements this;
- routine *eq* first calls *eq* on the auxiliary counters, then either takes $\langle 1, 0, 1, 0\rangle$ or $\langle 0, 1, 0, 1\rangle$ tokens from places $\langle b_1, \overline{b}_1, b_2, \overline{b}_2\rangle$, and finally puts them back;

and for both $i \in \{1, 2\}$:

- routine $i.inc$ calls $i.inc$, or calls $i.ismax$ and moves a token from $\overline{b}_i$ to b_i;
- routine $i.dec$ calls $i.dec$, or calls $i.iszero$ and moves a token from b_i to $\overline{b}_i$;
- routine $i.iszero$ calls $i.iszero$, takes a token from $\overline{b}_i$ and puts it back;
- routine $i.ismax$ calls $i.ismax$, takes a token from b_i and puts it back.

Given a correct program C that provides counters bounded by N, the library *Double* essentially uses two extra bits (each represented by a pair $\langle b_i, \overline{b}_i\rangle$ of places) to implement a program $Double \circ C$, where the bound on the provided counters is $2N$.

Lemma 5.5. *The counter library Double is $\lambda x.2x$-correct.*

Proof. When the control reaches the end of the *init* routine, $\mathit{Double} \circ C$ has as tape content a marking $M \oplus M_C$, where M_C is a marking of $\mathcal{N}(C)$ representing two 0-valued counters with bound N, and M has $\langle 0,1,0,1\rangle$ tokens of some type β on places $\langle b_1, \bar{b}_1, b_2, \bar{b}_2\rangle$. So by adding two new most significant bits, this tape content represents two 0-counters with bound $2N$. Notice that all routines apart from *init* preserve the invariant that tape contents have exaxtly two tokens of type β on the places $b_1, \bar{b}_1, b_2, \bar{b}_2$. We can easily check that those routines satisfy the respective correctness criteria.

For example, take the routine *eq* and let $n_1, n_2 \in \mathbb{N}$ denote the values of the counters represented by the current tape content M. If $n_1 = n_2$, then $M \oplus M_C$ has $\langle 1,0,1,0\rangle$ or $\langle 0,1,0,1\rangle$ tokens on places $\langle b_1, \bar{b}_1, b_2, \bar{b}_2\rangle$ and the numbers n_1', n_2' represented by M_C are the same. Since C is N-correct, the command call *eq* terminates. Moreover, one of the two operations to take $\langle 1,0,1,0\rangle$ or $\langle 0,1,0,1\rangle$ from $\langle b_1, \bar{b}_1, b_2, \bar{b}_2\rangle$ is possible. So *eq* terminates. Conversely, if $n_1 \neq n_2$, then either the content of places $\langle b_1, \bar{b}_1, b_2, \bar{b}_2\rangle$ is $\langle 0,1,1,0\rangle$ or $\langle 1,0,0,1\rangle$, or the values represented by the auxiliary counters are not equal. In the first case, the commands to take $\langle 1,0,1,0\rangle$ or $\langle 0,1,0,1\rangle$ are disabled; in the latter case, the command call *eq* does not terminate, by the correctness assumption on C.

In a similar fashion we can see that the routine *fresh* (see Figure 6) is correct: suppose the current tape content is $M \oplus M_C$ and A is the tape content that has one token α on place ν and is otherwise empty. If α is different from β (used on places $b_i, \bar{b}_i$) and also different from all data values in the configuration of C, then the routine must terminate without changing the tape content. If $\alpha = \beta$ then then both commands in lines 3 and 5 will block. If α appears in M_C then the command call *fresh* in line 1 must block. □

6 Bootstrapping Counter Libraries

The most complex part of our construction is an operator $-^*$ whose input is a counter library L. Its output L^* is also a counter library which behaves like an arbitrary number of copies of L composed in sequence. Namely, for every N-correct counter program C, the counter operations provided by $L^* \circ C$ behave in the same way as those provided by

$$\overbrace{L \circ \cdots \circ L}^{N} \circ \mathit{Enum}(1).$$

Hence, when L is F-correct, we have that L^* is F'-correct, where $F'(x) = F^x(1)$.

The main idea for the definition of L^* is to combine a distinguishing of name spaces as in the composition of libraries with an arbitrarily wide indexing mechanism like the one employed in Example 4.2. The key insight here is that a whole collection of 'addressing places' $\langle a_i, \bar{a}_i\rangle_i$ as used in Example 4.2 can be simulated by adding one layer of addressing. We will use the routine *fresh* to set up this addressing mechanism during initialisation, recursively.

$w, \bar{w}$	contain the addressing mechanism for recording the current control information of the L_i
$w', \bar{w}'$	contain the name spaces of the L_i, where the multiplicities of tokens identify the indices i
$p, \bar{p}$	identify the currently active routines of the L_i
$p', \bar{p}'$	identify the currently active commands of the L_i
f	temporarily stores a guessed datum for comparison
ν	stores the datum to be checked for freshness

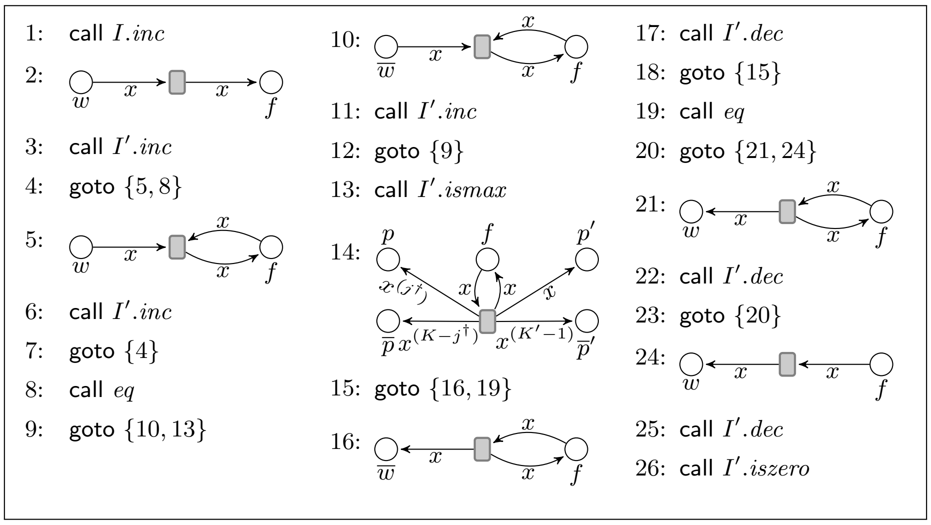
Table 1. A glossary of tape places in L^*. Not listed are places that are internally used in transition commands of L, nor the control places of $\mathcal{N}(L^* \circ C)$.

Let us write here I, I' for the two N-bounded auxiliary counters and number the copies of L by $0, \ldots, N-1$, writing $\ell_1 : R_1$ up to $\ell_K : R_K$ for the routines of L and K' for the maximum number of commands in a routine. Since L is a counter library, it has $K = |\Gamma'| = 11$ routines, and we assume without loss of generality that $\ell_1 = \mathit{init}$ and $\ell_2 = \mathit{fresh}$. The net for L^* can maintain the control and the tape of each copy of L in the implicit composition as follows.

- To record that the program counter of the ith copy of L is currently in routine $\ell_j : R_j$ at line j', $\langle i, N-1-i, j, K-j, j', K'-j' \rangle$ tokens carrying a separate name α_i are kept on special places $\langle w, \overline{w}, p, \overline{p}, p', \overline{p}' \rangle$.
- The current height i of the stack of subroutine calls is kept in one of the auxiliary counters, and we have that:
  - for all $i' < i$, the program counter of the i'th copy of L is at some subroutine invocation call ℓ' such that the program counter of the $(i'+1)$th copy of L is in the routine named ℓ';
  - for all $i' > i$, there are $\langle i', N-1-i', 0, 0, 0, 0 \rangle$ tokens carrying $\alpha_{i'}$ on places $\langle w, \overline{w}, p, \overline{p}, p', \overline{p}' \rangle$.
- For every name manipulated by the ith copy of L, $\langle i, N-1-i \rangle$ tokens carrying it are kept on special places $\langle w', \overline{w}' \rangle$.

To define L^*, its places are all the places that occur in L, plus nine special places w, $\overline{w}$, w', $\overline{w}'$, p, $\overline{p}$, p', $\overline{p}'$ and f. All routines of the library L^* end in the same sequence of commands, which we will just call *the simulation loop.* This uses I' and place f repeatedly to identify numbers j' and j'' such that there are exactly $\langle I, N-1-I, j', K-j', j'', K'-j'' \rangle$ tokens carrying α_I on $\langle w, \overline{w}, p, \overline{p}, p', \overline{p}' \rangle$, and then advance the Ith copy of L by performing its command c at line j'' in routine $\ell_{j'} : R_{j'}$ of L as follows.

- If c is a UDPN transition, use I' and place f to maintain the Ith name space, i.e. to ensure that all names manipulated by c have $\langle I, N-1-I \rangle$ tokens on places $\langle w', \overline{w}' \rangle$.
- If c has put a datum β on place ν, invoke routine *fresh* of the auxiliary counters.
- If c is a nondeterministic jump goto G, choose $j^{\ddagger} \in G$ and ensure that there are $\langle j^{\ddagger}, K' - j^{\ddagger} \rangle$ tokens carrying α_I on places $\langle p', \overline{p}' \rangle$.



```
1:  call $I.inc$
2:
3:  call $I'.inc$
4:  goto $\{5, 8\}$
5:
6:  call $I'.inc$
7:  goto $\{4\}$
8:  call $eq$
9:  goto $\{10, 13\}$
10:
11: call $I'.inc$
12: goto $\{9\}$
13: call $I'.ismax$
14:
15: goto $\{16, 19\}$
16:
17: call $I'.dec$
18: goto $\{15\}$
19: call $eq$
20: goto $\{21, 24\}$
21:
22: call $I'.dec$
23: goto $\{20\}$
24:
25: call $I'.dec$
26: call $I'.iszero$
```

Fig. 7. Performing a call $\ell_{j^\dagger}$ provided $I < N-1$. At the beginning, I' is assumed to be zero, and the same is guaranteed at the end.

- If c is a subroutine invocation call $\ell_{j^\dagger}$ and $I < N-1$, put $\langle j^\dagger, K-j^\dagger, 1, K'-1\rangle$ tokens carrying α_{I+1} on places $\langle p, \overline{p}, p', \overline{p}'\rangle$, and increment I. Example code that implements this can be found in Figure 7.
- If c is a subroutine invocation call ℓ', $I = N-1$ and ℓ' is not an increment or a decrement (of the trivial counter program $Enum(1)$), simply increment the program counter by moving a token carrying α_I from place $\overline{p}'$ to place p'. When ℓ' is an increment or a decrement, L^* blocks.
- In the remaining case, c is return. Remove the tokens carrying α_I from places $\langle p, \overline{p}, p', \overline{p}'\rangle$. If $I > 0$, move a token carrying α_{I-1} from $\overline{p}'$ to place p' and decrement I. Otherwise, exit the loop and return.

The code of this simulation loop is used (inlined) in the actual code for the routines R_j^* of L^*, which simulate the routines of the outmost copy L_0 as follows.

Initialization ($\ell_j = \ell_1 = init$)**:**
- call *init* to initialise the auxiliary counters;
- for each $i \in \{0, \ldots, N-1\}$, put $\langle i, N-1-i\rangle$ tokens carrying a fresh name α_i onto places $\langle w, \overline{w}\rangle$ (this uses the auxiliary counters, their *fresh* routine, and place f);
- put $\langle 1, K-1, 1, K'-1\rangle$ tokens carrying name α_0 onto places $\langle p, \overline{p}, p', \overline{p}'\rangle$ to record that the first routine (*init*) of L_0 should be simulated from line 1;
- enter the simulation loop.

Freshness test ($\ell_j = \ell_2 = fresh$)**:**
- call *fresh* to check that the datum β on place ν is distinct from all data used in the auxiliary counters;
- verify that $\beta \neq \alpha_i$ for all $i \in \{0, \ldots, N-1\}$;

- put $\langle 2, K-2, 1, K'-1\rangle$ tokens carrying name α_0 onto places $\langle p, \overline{p}, p', \overline{p}'\rangle$ to record that the 2nd routine (*fresh*) of L_0 should be simulated from line 1;
- enter the simulation loop.

Routines $\ell_j : R_j^*$ for $j > 2$: – put $\langle 2, K-j, 1, K'-j\rangle$ tokens carrying name α_0 onto places $\langle p, \overline{p}, p', \overline{p}'\rangle$ to record that the jth routine of L_0 should be simulated from line 1;
- enter the simulation loop.

Notice that these routines do not actually call routines of L but simulate them internally and terminate only after the whole simulation loop terminates.

Observe that L^* is computable from L in logarithmic space.

Lemma 6.1. *For every function $F : \mathbb{N} \to \mathbb{N}$ and F-corrent counter library L, the couner library L^* is $\lambda x.F^x(1)$-correct.*

Proof. Recall that for any $N \in \mathbb{N}$, the program $L^N \circ Enum(1)$, the N-fold composition of L with itself and the trivial 1-bounded counter program, is $F^k(1)$-correct by our assumption on L and Lemma 5.3.

We need to show that $L^* \circ C$ is $F^N(1)$-correct for every N-correct counter program C. We argue that, after initialisation and with respect to termination/nontermination of the counter routines $\varGamma \setminus \{init\}$, the program $L^* \circ C$ behaves just as $L^N \circ Enum(1)$.

Fix $k \leq \{1, \dots, N\}$. We say a tape content M of $L^k \circ Enum(1)$ is *represented* by a tape content $M^\dagger$ of $L^* \circ C$ if, for all $i \in \{0, \dots, k-1\}$,

1. there is a unique name α_{N-k+i} that labels $\langle N-k+i, k-i-1\rangle$ tokens on places $\langle w, \bar{w}\rangle$ in $M^\dagger$, and
2. the restriction M_i of M to the names in the name space to the ith copy of L equals the restriction $M^\dagger_{N-k+i}$ of $M^\dagger$ to the names that label $\langle N-k+i, k-i-1\rangle$ tokens on places $\langle w', \overline{w}'\rangle$ and to the places of L.

Let us now look at how the code of the simulation loop in L^* acts on representations of tape contents of $L^k \circ Enum(1)$.

For two tape contents M and M' of $L^k \circ Enum(1)$, we say that the simulation loop $\langle j, j'\rangle$*-computes* M' from M if from a tape content (of $L^* \circ C$) that represents M, where the stack height stored in the first auxiliary counter is $N-k$ and there are $\langle j, K-j, j', K'-j'\rangle$ tokens carrying α_{N-k} on places $\langle p, \bar{p}, p', \bar{p}'\rangle$, the net $\mathcal{N}(L^* \circ C)$ can reach, by simulating a single command and without reducing the stack height below $N-k$, a tape content that represents M'.

The following claim can be shown by induction on $k \leq N$.

Claim. For two tape contents M, M' of $L^k \circ Enum(1)$, command j' of routine $\ell_j{:}R_j$ computes M' from M in $L^k \circ Enum(1)$, if and only if, the simulation loop in $L^* \circ C$ $\langle j, j'\rangle$-computes M' from M.

This in particular (for $k = N$) implies that, after correct initialisation and with respect to termination of routines other than *init*, $L^* \circ C$ bahaves just like

$L^N \circ \mathit{Enum}(1)$. Notice that if $L^* \circ C$ computes a *fresh* command within the *init* routine of $L^N \circ \mathit{Enum}(1)$, the simulation loop ensures that the new datum is also distinct from all values used in the auxiliary counters of L^*. It remains to show that, for any tape content M that is computed from the empty tape content by the *init* routine of $L^N \circ \mathit{Enum}(1)$, there is a tape content of $L^* \circ C$ that represents M and that is computed from the empty tape content by the *init* routine of $L^* \circ C$.

To see this, observe that the first command of *init* in $L^* \circ C$ calls the initialisation routine of C, providing the auxiliary counters. By the assumption that C is N-correct, this allows to place exactly $\langle i, N-1-i\rangle$ tokens carrying a fresh name α_i onto places $\langle w, \overline{w}\rangle$ for each $i \in \{0, \ldots, N-1\}$. Thus, after these commands, the tape content of $L^* \circ C$ represents the empty tape content of $L^N \circ \mathit{Enum}(1)$. The rest of the initialisation routine contains the code of the simulation loop, so the conclusion follows from the claim above, where $k = N$. □

7 Ackermann-Hardness

We work with a hierarchy of functions A_i, defined as follows for all k and x in $\mathbb{N}$:

$$A_1(x) \stackrel{\text{def}}{=} 2x \qquad \text{and} \qquad A_{k+2}(x) \stackrel{\text{def}}{=} A_{k+1}^x(1) \ .$$

The *Ackermann* function is then defined as $A_\omega(x) \stackrel{\text{def}}{=} A_{x+1}(x)$, and by [25, Section 2.3.2 and Theorem 4.1], we have that the next problem is Ackermann-complete (cf. Section 3) and that the class Ackermann is closed under primitive recursive reductions:

Input: A 2-counter Minsky program $\mathcal{M}$ with n commands.
Question: Can $\mathcal{M}$ reach the `halt` command by a computation during which both counter values are less than $A_\omega(n)$?

Theorem 7.1. *The coverability problem for UDPNs is* Ackermann*-hard.*

Proof. Suppose $\mathcal{M}$ is a 2-counter Minsky program with n commands.

By Lemmas 6.1, 5.3 and 5.5, we have that the counter program

$$\mathit{Acker}(n) \stackrel{\text{def}}{=} (\cdots(\mathit{Double}\overbrace{{}^*)^* \cdots)^*}^{n} \circ \mathit{Enum}(n)$$

is $A_\omega(n)$-correct.

Since the star operator is computable in logarithmic space and increases the number of places by adding a constant, we have that $\mathit{Acker}(n)$ is computable in time elementary in n, and that its number of places is linear in n.

It remains to simulate $\mathcal{M}$ by a one-routine library that uses the two $A_\omega(n)$-bounded counters provided by the counter program $\mathit{Acker}(n)$. The resulting one-routine program can terminate if and only if its UDPN can cover the marking in which the two line-number places point to the last command. □

8 Conclusion

We have shown that the reachability, coverability and boundedness problems for pushdown vector addition systems are $\mathbf{F}_3$-hard. Whether they are decidable remains unknown, in the case of reachability even with only one counter, i.e. in dimension 1. The best known lower bound for the latter problem is NP [20].

For unordered data Petri nets, we have advanced the state-of-the-art lower bound of the coverability (and thus also reachability) problem from $\mathbf{F}_3$ [15] to $\mathbf{F}_\omega$. A gap therefore remains to the best known $\mathbf{F}_{\omega\cdot 2}$ upper bound [17]. We conjecture $\mathbf{F}_\omega$-completeness, which would complement nicely the $\mathbf{F}_{\omega\cdot 2}$-completeness [17] and $\mathbf{F}_{\omega^\omega}$-completeness [23] results for the extensions of UDPN by fresh name generation and whole-place operations, respectively. However, the tightening from $\mathbf{F}_{\omega\cdot 2}$ to $\mathbf{F}_\omega$ membership seems a considerable challenge for the following reason: by providing UDPNs with an initial supply of N fresh names on some auxiliary place, they can operate for N steps indistinguishably from ν-Petri nets, and so the classical backward coverability algorithm [1,8] cannot terminate for UDPNs in only Ackermann many iterations.

References

1. Parosh Aziz Abdulla, Karlis Čerāns, Bengt Jonsson, and Yih-Kuen Tsay. Algorithmic analysis of programs with well quasi-ordered domains. *Inform. and Comput.*, 160(1–2):109–127, 2000.
2. Mohamed Faouzi Atig and Pierre Ganty. Approximating Petri net reachability along context-free traces. In *FSTTCS*, volume 13 of *LIPIcs*, pages 152–163. LZI, 2011.
3. Mikołaj Bojańczyk, Bartek Klin, and Sławomir Lasota. Automata theory in nominal sets. *Logic. Meth. in Comput. Sci.*, 10(3:4):1–44, 2014.
4. P. Chambart and Ph. Schnoebelen. The ordinal recursive complexity of lossy channel systems. In *LICS*, pages 205–216. IEEE Press, 2008.
5. Normann Decker and Daniel Thoma. On freeze LTL with ordered attributes. In *FoSSaCS*, volume 9634 of *LNCS*, pages 269–284. Springer, 2016.
6. Diego Figueira, Santiago Figueira, Sylvain Schmitz, and Philippe Schnoebelen. Ackermannian and primitive-recursive bounds with Dickson's Lemma. In *LICS*, pages 269–278. IEEE Press, 2011.
7. Alain Finkel, Pierre McKenzie, and Claudine Picaronny. A well-structured framework for analysing Petri net extensions. *Inform. and Comput.*, 195(1–2):1–29, 2004.
8. Alain Finkel and Philippe Schnoebelen. Well-structured transition systems everywhere! *Theor. Comput. Sci.*, 256(1–2):63–92, 2001.
9. Christoph Haase, Sylvain Schmitz, and Philippe Schnoebelen. The power of priority channel systems. *Logic. Meth. in Comput. Sci.*, 10(4:4):1–39, 2014.
10. Serge Haddad, Sylvain Schmitz, and Philippe Schnoebelen. The ordinal recursive complexity of timed-arc Petri nets, data nets, and other enriched nets. In *LICS*, pages 355–364. IEEE Press, 2012.
11. Piotr Hofman, Sławomir Lasota, Ranko Lazić, Jérôme Leroux, Sylvain Schmitz, and Patrick Totzke. Coverability trees for Petri nets with unordered data. In *FoSSaCS*, volume 9634 of *LNCS*, pages 445–461. Springer, 2016.

12. Piotr Hofman, Jérôme Leroux, and Patrick Totzke. Linear combinations of unordered data vectors. arXiv:1610.01470 [cs.LO], 2016.
13. Kurt Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use - Volume 1, Second Edition.* Monographs in Theoretical Computer Science. An EATCS Series. Springer, 1996.
14. Ranko Lazić. The reachability problem for vector addition systems with a stack is not elementary. arXiv:1310.1767 [cs.FL], 2013.
15. Ranko Lazić, Tom Newcomb, Joël Ouaknine, A.W. Roscoe, and James Worrell. Nets with tokens which carry data. *Fund. Inform.*, 88(3):251–274, 2008.
16. Ranko Lazić, Joël Ouaknine, and James Worrell. Zeno, Hercules, and the Hydra: Safety metric temporal logic is Ackermann-complete. *ACM Trans. Comput. Logic*, 17(3), 2016.
17. Ranko Lazić and Sylvain Schmitz. The complexity of coverability in ν-Petri nets. In *LICS*, pages 467–476. ACM, 2016.
18. Jérôme Leroux, M. Praveen, and Grégoire Sutre. Hyper-Ackermannian bounds for pushdown vector addition systems. In *CSL-LICS*, pages 63:1–63:10. ACM, 2014.
19. Jérôme Leroux, Grégoire Sutre, and Patrick Totzke. On boundedness problems for pushdown vector addition systems. In *RP*, volume 9328 of *LNCS*, pages 101–113. Springer, 2015.
20. Jérôme Leroux, Grégoire Sutre, and Patrick Totzke. On the coverability problem for pushdown vector addition systems in one dimension. In *ICALP*, volume 9135 of *LNCS*, pages 324–336. Springer, 2015.
21. Richard Lipton. The reachability problem requires exponential space. Technical Report 62, Yale University, 1976.
22. Charles Rackoff. The covering and boundedness problems for vector addition systems. *Theor. Comput. Sci.*, 6(2):223–231, 1978.
23. Fernando Rosa-Velardo. Ordinal recursive complexity of unordered data nets. Technical Report TR-4-14, Universidad Complutense de Madrid, 2014.
24. Fernando Rosa-Velardo and David de Frutos-Escrig. Decidability and complexity of Petri nets with unordered data. *Theor. Comput. Sci.*, 412(34):4439–4451, 2011.
25. Sylvain Schmitz. Complexity hierarchies beyond Elementary. *ACM Trans. Comput. Theory*, 8(1):1–36, 2016.
26. Sylvain Schmitz and Philippe Schnoebelen. Multiply-recursive upper bounds with Higman's Lemma. In *ICALP*, volume 6756 of *LNCS*, pages 441–452. Springer, 2011.
27. Philippe Schnoebelen. Revisiting Ackermann-hardness for lossy counter machines and reset Petri nets. In *MFCS*, volume 6281 of *LNCS*, pages 616–628. Springer, 2010.
28. Larry J. Stockmeyer. *The complexity of decision procedures in Automata Theory and Logic.* PhD thesis, MIT, 1974. Project MAC TR-133.