Products of Recursive Programs for Hypersafety Verification

(Extended Version)

RUOTONG CHENG, University of Toronto, Canada
AZADEH FARZAN, University of Toronto, Canada

We study the problem of *automated hypersafety verification* of *infinite-state recursive programs*. We propose an *infinite* class of *product programs*, specifically designed with *recursion* in mind, that reduce the hypersafety verification of a recursive program to standard safety verification. For this, we combine insights from *language theory* and *concurrency theory* to propose an algorithmic solution for constructing an infinite class of recursive product programs. One key insight is that, using the simple theory of *visibly pushdown languages*, one can maintain the *recursive structure* of syntactic program alignments which is vital to constructing a new product program that can be viewed as a classic recursive program — that is, one that can be executed on a single stack. Another key insight is that techniques from *concurrency* theory can be generalized to help define product programs based on the view that the parallel composition of individual recursive programs includes all possible *alignments* from which a *sound* set of alignments that faithfully preserve the satisfaction of the hypersafety property can be selected. On the practical side, we formulate a family of parametric canonical product constructions that are intuitive to programmers and can be used as building blocks to specify recursive product programs for the purpose of relational and hypersafety verification, with the idea that the right product program can be verified automatically using existing techniques. We demonstrate the effectiveness of these techniques through an implementation and highly promising experimental results.

CCS Concepts: • **Software and its engineering → Software verification**; • **Theory of computation → Logic and verification**; *Concurrency*; *Grammars and context-free languages*.

Additional Key Words and Phrases: hyperproperties,product programs,reductions,visibly pushdown languages

This article is an extended version of:
Ruotong Cheng and Azadeh Farzan. 2025. Products of Recursive Programs for Hypersafety Verification. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 382 (October 2025), 29 pages. https://doi.org/10.1145/3763160

1 Introduction

Hyperproperties [16] are properties expressible over multiple runs of a program. A k-safety property is a property whose violation is witnessed by k finite-length program runs. Determinism is an example of such a property: non-determinism can only be witnessed by two runs of the program on the same input that produce two different outputs. This makes determinism an instance of a 2-safety property. A hypersafety property is a k-safety property for $k > 1$.

Verification of hypersafety properties was first achieved by reducing the problem to a standard safety verification by means of *self-composition* [8]. For example, in order to verify that P is deterministic, one can verify that $P; P'$ under the assumption of equivalent inputs for P and P' guarantee the equivalence of their outputs at the end, where P' is a fresh copy of P with a disjoint memory. It was argued in [8] that any other *independent composition* operator can be used in place of the sequential composition operator. Later, *cross products* [58], which combine P and P' synchronously in lockstep, were used in a more limited setup of compiler validation. *Program products* [7] were introduced as combining the two notions where one can *fuse* the two programs both synchronously and asynchronously where appropriate.

It is now well-established that product programs are vital to verification of hypersafety properties and relational reasoning in general [14, 27, 38, 52, 53]. In algorithmic verification, they permit the use of existing tools and techniques for classic safety verification of sequential programs to

Authors' Contact Information: Ruotong Cheng, University of Toronto, Toronto, Canada, chengrt@cs.toronto.edu; Azadeh Farzan, University of Toronto, Toronto, Canada, azadeh@cs.toronto.edu.

be applied to hypersafety verification. This is mainly due to the fact that they accommodate the creation of instances that admit simpler proofs. Approaches to hypersafety or relational verification typically *implicitly* or *explicitly* define a state space of product programs that may be used to simplify verification. For instance in [7, 53], a successful derivation of the underlying logical rules *implicitly* defines the verified product program, while in [26], the product programs are syntactically defined and look like classic programs. With the implicit construction, since the choice of application of a proof rule and the next alignment step are combined and one typically has control over the entire verification stack, one can opt for an ergonomic design that works more effectively; e.g., heuristics for proofs search can be tuned to the specific problem. On the other hand, since the explicit construction decouples the construction of the program from the construction of the proof, it can benefit from improvements in standard verification tools. More importantly, having access to a concrete (product) program opens up possibilities for using other standard tools for program analyses such as symbolic/concolic execution, fuzzing, monitoring and/or repair, and static analyses. The two sets of techniques are therefore complementary and serve different use cases. In this paper, we are interested in the *explicit* construction of product programs.

This problem is somewhat well-studied for iterative programs, but has only been lightly studied in the context of recursive programs, and the majority of the proposed techniques fit in the *implicit* construction category. Most even bypass the program view entirely and considers the question of how a similar concept can be incorporated in the CHC (Constrained Horn Clauses) encoding of them, in the form of CHCProduct [47] or *fold/unfold* [4, 20] techniques to implicitly explore the space of the product programs and proofs simultaneously (more details in Section 9). The very few [22] that construct explicit product programs use specialized constructions that help solve a limited set of verification instances.

Consider the recursive code illustrated in Figure 1, which computes the integer result of dividing the numerator `n` by the denominator `d`. Assume that we want to check if this *implementation* satisfies the following property of the original mathematical function:

```
def div(n: nat, d: nat) -> nat
    if n < d: return 0
    else: q = div(n - d, d) + 1
          return q
```

Fig. 1. Recursive integer division.

$$n \leq n' \implies \texttt{div}(n, d) \leq \texttt{div}(n', d) \qquad \text{(MONOTONICITY)}$$

First, observe that MONOTONICITY is a 2-safety property. The most straightforward encoding for solving the MONOTONICITY property uses self-composition, that is, it reduces the problem to the safety verification of $\texttt{div}(\texttt{n}, \texttt{d}); \texttt{div}(\texttt{n}', \texttt{d}')$. However, this instance cannot be verified by an existing automated solver due to the nonlinearity of the arithmetic reasoning involved in guessing the required inductive function summaries [30]. In particular, the summary *must* precisely capture that $\texttt{div}(\texttt{n}, \texttt{d})$ is the *integer division* function.

Product programs are conceptually straightforward when it comes to non-recursive code. However, recursion introduces significant additional complexity. If P and P' are recursive, then $P; P'$ is a recursive program in the standard sense. However, the same cannot be said for an arbitrary product program made from P and P' (even if P' is a copy of P). For instance, it is well-known that the parallel product of two recursive programs is a 2-stack machine, whose syntactic traces form a *context sensitive* language [37], which does not correspond to a recursive program. This problem does not exist in the iterative setting.

For MONOTONICITY, the programmer's intuition points to a *lockstep* (or synchronous) product program, namely one in which the two recursive codes are executed *in tandem*, when possible. An appropriate product program then is one that (1) looks like a classic (single stack) recursive program, and (2) simulates the two copies of `div` in tandem. This product program can be defined using four recursive functions: `div1` and `div2` as two distinct copies of `div` from Figure 1, and two

distinct copies of the variation illustrated in Figure 2. In a lockstep simulation we need to execute in tandem while both copies last, and then a given copy alone when the other copy has terminated. `div12` assumes that the second copy is still running and as such calls it in tandem, while `div1` assumes that the second copy has terminated and only locally recurses.

```
def div12(n1, d1, n2, d2) -> (nat,nat)
  if n1 < d1: q2 = div2(n2, d2)
    return 0, q2
  else: q2, q1 = div21(n2, d2, n1 - d1, d1)
    return q1 + 1, q2
```

```
def div21(n2, d2, n1, d1) -> (nat,nat)
  if n2 < d2:  q1 = div1(n1, d1)
    return 0, q1
  else: q1, q2 = div12(n1, d1, n2 - d2, d2)
    return q2 + 1, q1
```

Fig. 2. With two disjoints copies of `div`, these functions simulate a synchronous product of two `div`'s

This product program does not syntactically match the one that we formally define and algorithmically construct in this paper, but it is more human-readable and conveys the two main ideas: (1) our product programs are *syntactic* in the sense that any syntactic run of it is identical to a syntactic run of the full (parallel) product of two distinct copies of `div` (not strictly true for this simplified example), and (2) even in the simplest case, they are far from trivial. How can one systematically construct such a program?

In [26–28], this is achieved for iterative (non-recursive) programs, by taking an *operational* view of the program [25]. The set of all possible alignments are defined by the parallel composition of the components, which include arbitrary interleavings of syntactic runs from individual components. Then, a particular choice of product programs becomes about selecting a *sound* subset of these interleavings. As long as the subset has a finite representation (in a program or program-like structure), the product program and this set of behaviours are interchangeable.

Our first important observation is that, for the same operational style to work for recursive programs, one has to be explicitly conscious of the *structure* of the recursive runs, in terms of the matching of calls and returns in each component. To visualize this, consider runs τ, τ' illustrated on the right, as being runs of the two disjoint copies $\mathtt{div}(\mathtt{n}, \mathtt{d})$ and $\mathtt{div}(\mathtt{n}', \mathtt{d}')$ respectively. Open and closed brackets respectively denote calls and returns, and all internal actions are represented as the ellipses, since their specifics are unimportant. Note that each illustrated run is *well-nested*, in the sense that we have a *balanced* sequence of calls and returns. These are also called *interprocedurally realizable paths* [50, 51].

Below we can see two (of the many) ways that one can *interleave* (or align) τ and τ':

$$[\ldots\ [\ldots\ (\ldots\ (\ldots\ [\ldots\ldots]\ \ldots)\ \ldots)\ \ldots]\ \ldots] \qquad \text{(well-nested)}$$

$$[\ldots\ [\ldots\ (\ldots\ (\ldots\ [\ldots\ldots)\ \ldots)\ \ldots]\ \ldots]\ \ldots] \qquad \text{(ill-nested)}$$

and we contend that the well-nested run belongs to a product program and the ill-nested one does not. In the well-nested one, the matched calls and returns belong to the same component from which they originated. If we ignore the types/colours, we can pretend that the former is a well-matched run of a classic recursive program, whereas in the latter, we obtain an *interprocedurally unrealizable* path: a call and a non-matching return. Therefore, to construct a product program in this operational style, one needs (1) representations for syntactic program runs that maintain the recursive structure, and (2) mechanisms to isolate sound appropriate subsets of the well-nested alignments as product programs.

For (1), we lean on the well-established theory of *visibly pushdown languages* [2] to formally represent the syntactic runs of recursive programs, while retaining the recursive structure in the runs. Unlike a *call graph*, a classic model for recursive program analysis, a visibly pushdown

grammar or *automaton* captures the *well-matched* syntactic runs of a recursive program. We then give a construction of the product program in the same formalism, which guarantees that it can be treated as a classic recursive program for all intents and purposes.

To understand the complications of devising (2), let us further expand our simple `div` example with a slightly more complicated property:

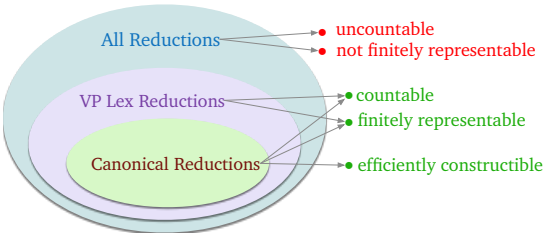
$$n = 2 \times n' \implies \texttt{div}(n, d) \geq 2 \times \texttt{div}(n', d) \qquad \text{(SCALING)}$$

where SCALING is a 2-safety property. Observe that a *lockstep* product is no longer appropriate. For SCALING, we expect the execution of the first copy to be twice as long as the second copy. A *suitable* product program, therefore, must synchronize each recursive call of $\texttt{div}(\texttt{n}, \texttt{d})$ against two recursive calls of $\texttt{div}(\texttt{n}', \texttt{d}')$. Consider a variation of this property where the scale factor changes from 2 to a different constant, like 5, which immediately changes the product program needed for its verification to a new one. This motivates the following research question: how can one systematically define a state space of useful product programs for recursive programs for the purpose of relational and hypersafety verification? Any specific choice of a product program can be viewed as a *reduction* of the set of all alignments (or the behaviours of the parallel product). For a group of k syntactic runs, one from each individual component, a (minimal) *reduction* includes precisely one alignment of them, and discards all others. For recursive programs, with syntactic runs of unbounded length, there are infinitely many choices that can be made and as such infinitely many reductions that can be defined.

Our key (theoretical) contribution in this paper is to define an infinite space of product programs (reductions). Each reduction is representable as standard recursive programs and therefore can be proved correct using standard verification techniques. Each member of this space is then formally definable and algorithmically constructible, and the set is *countable* and (recursively) *enumerable*, in the same style as [27, 28].

To define the state space of these programs, we rely on the observation that the task of choosing one alignment from a group is like *scheduling*. Intuitively, one can think of the resolution of the nondeterminism to be done through the choices made by a *scheduler*. Most importantly, we observe that these *schedulers* can also be modelled in the same formalism as recursive programs. Given a set of statements, coming from distinct components, one can formally define a set of *schedules* (which are basically sequences over these statements), which can then serve as a restriction of the set of all alignments. For instance, a lockstep scheduler of two (distinct) components can be viewed as one that admits any schedule *alternating* statements from the two components. Construction of a reduction then becomes a standard *intersection* of the set of schedules defined by the scheduler and the well-nested shuffle of the components; a construction that is supported by the theory of visibly pushdown languages for both automata and grammars. In Section 3, we formalize these schedulers as *contextual lexicographical orders* and the yielded reductions as *contextual lexicographical reductions*, *lex reductions* for short. Here, *context* refers to the fact that the schedule is *stateful* and adjusts its future choices based on what it has done in the past. These can be viewed as generalization of similar concepts from [26–28] to recursive programs.

We use ideas from *trace theory* [46] (a simple theory for concurrency) to argue that our reductions are constructible. However, in Section 4, we propose a new optimized algorithm for constructing a representation of our contextual lexicographic reductions as *visibly pushdown automata and grammars*. The new construction leverages the fact that, in the context of hypersafety (and relational) verification, we have the *full commutativity* of the actions from different components, to propose a *simpler* and *more compact* construction of visibly pushdown (VP) lex reductions.



The specific feature of our proposed space of reductions is that, unlike the set of all reductions, it is recursively enumerable. This means that assuming the existence of a verifiable reduction in the space, one can set up a trivial decision procedure to find it. The underlying verification problem remains undecidable for generic infinite-state programs, but at least, we know that the component of searching for a reduction does not add any new divergence opportunities. This is in sharp contrast to, for example, considering the state space of *all possible* reductions, where one has to either bound the space to a finite subset [7] or use ad-hoc heuristics [47, 53] to curb this intrinsic divergent behaviour. Moreover, the VP lex reductions are finitely representable, whereas an arbitrary reduction may not be.

Rather than setting up a search algorithm for a reduction in the space of VP lex reductions, we opted for investigating a different more practical hypothesis in this paper. This is motivated by the fact that the search, even in the simpler setup of [27, 28], is very expensive and unscalable and the problem only gets worse with the increased complexity of the same operations for pushdown systems [23]. So, there is a natural question about what to do when one hits that scalability barrier. There is a well-established tradition of user-provided annotations to help with the automation of harder instances. For instance, in [52], when the mined predicates are insufficient for the combined search for an alignment and a proof, the user can input predicates in the proof (of the unknown alignment) to help; which is in line with this tradition. In this paper, we explore the alternative possibility of user-provided annotations for the specification of reductions. we devise a smaller (yet infinite) subfamily of lex reductions, we call *canonical reductions*, that have three properties:

(1) They are intuitive to the programmer. For instance, looking at `div` and SCALING, the programmer can easily guess and specify that they are interested in an asynchronous product of two `div` instances where one copy moves twice as fast as the other copy.
(2) They are simply and succinctly specifiable (like a correctness property is in a logical language) as part of the verification command, using *composable building blocks*.
(3) They are expressive enough to provide the means of solving a broad range of instances.

To understand the obstacles in achieving this, let us expand our simple `div` example one more time with a substantially more sophisticated property:

$$n'' = n + n' \implies \mathtt{div}(n'', d) \geq \mathtt{div}(n, d) + \mathtt{div}(n', d) \qquad \text{(DISTRIBUTIVITY)}$$

where DISTRIBUTIVITY is a 3-safety property, and clearly, *lockstep* product is not the right choice for this and varying the rate is also insufficient. Consider the product program

$$(\mathtt{div}(\mathtt{n}, \mathtt{d}) \oplus \mathtt{div}(\mathtt{n}', \mathtt{d}')) \otimes \mathtt{div}(\mathtt{n}'', \mathtt{d}'')$$

where we intentionally use the vague $\oplus$ and $\otimes$ operators for the choice of product operators for now. Intuitively, since $n'' = n + n'$, we expect the execution length of the $\mathtt{div}(\mathtt{n}'', \mathtt{d}'')$ copy to be roughly the same as the sum of the execution lengths of the $\mathtt{div}(\mathtt{n}, \mathtt{d})$ and $\mathtt{div}(\mathtt{n}', \mathtt{d}')$ copies. Therefore, in a suitable product program, we want to run a kind of *sequential composition* of $\mathtt{div}(\mathtt{n}, \mathtt{d})$ and $\mathtt{div}(\mathtt{n}', \mathtt{d}')$ in tandem with $\mathtt{div}(\mathtt{n}'', \mathtt{d}'')$; that is, align each of the first n recursive calls of $\mathtt{div}(\mathtt{n}'', \mathtt{d}'')$ with a call from $\mathtt{div}(\mathtt{n}, \mathtt{d})$ and the last n' recursive calls of $\mathtt{div}(\mathtt{n}'', \mathtt{d}'')$ with a call from $\mathtt{div}(\mathtt{n}', \mathtt{d}')$. This is precisely how looped computations are split [7] to accommodate the construction of an effective product program. However, a recursive computation cannot be split like a loop through *loop fission*, by breaking a loop into two sequentially composed loops. The solution, at the high level, is in fact the complete opposite. One first *merges* $\mathtt{div}(\mathtt{n}, \mathtt{d})$ and $\mathtt{div}(\mathtt{n}', \mathtt{d}')$ so that it looks like

one recursive sequence of calls, and then synchronizes this merged program with $\texttt{div}(\texttt{n}'', \texttt{d}'')$ in a lockstep manner. This *merge* can be viewed as a recursive analog of *loop fusion*.

To visualize this, recall the runs τ and τ' illustrated earlier, and consider τ'' illustrated on the right, as being respectively runs of the three disjoint copies $\texttt{div}(\texttt{n}, \texttt{d})$, $\texttt{div}(\texttt{n}', \texttt{d}')$, and $\texttt{div}(\texttt{n}'', \texttt{d}'')$. First, τ and τ' are *merged* together, by *fully nesting* τ' inside τ to get ρ. This operation (standing in for $\oplus$) can be viewed as the recursive analog of *concatenation* in the looped (iterative) setting.

We introduce *nested concatenation*, the recursive dual of concatenation as a *novel building block*. Note that, for a *tail recursion*, nested concatenation behaves similar to concatenation, but otherwise, it is an entirely different and new way of composing computations. ρ can then be composed with τ'' through a standard lockstep composition, where the green and blue/pink calls are executed in tandem. Effectively, the innermost two calls of τ'' are synched with the two calls of τ', and its outermost three calls are synched with the calls of τ.

Looking at the MONOTONICITY and SCALING properties of div, a *parametric-by-speed* version of lockstep is also a suitable candidate for another building block. Note that the parametric lockstep, by itself, yields an unbounded family of reductions. Combined with nested concatenation, they give rise to an infinite number of possibilities for product programs that can be very succinctly specified by the user. These building blocks are highly intuitive, but very nontrivial to define formally and construct correctly. Our second (theoretical and practical) key contribution is that we propose formal definitions for these building block reductions (Section 6.1), and also show that these reductions all belong to the state space of lex reductions. Moreover, we propose an alternative direct way of constructing product programs made from the two building block reductions of *nested concatenation* and *parametrized lockstep* (Section 6.3) that yields much more compact representations for these canonical reductions, compared to the generic (but optimized) construction from Section 4 for all lex reductions. Remarkably, the subfamily demonstrates a large degree of expressivity with nearly all of our 103 benchmarks becoming solvable using a a reduction from this family.

As a bonus (third) contribution, we make a new observation for the verification of concurrent recursive programs: The well-nested shuffle is an under-approximation of a *concurrent recursive program*, which itself is a standard (nondeterministic) recursive program. As such, any existing technique for recursive program analysis and bug finding can be used to find errors in it which are guaranteed to be errors in the original program. Moreover, in Section 5, we formulate conditions under which this under-approximation becomes a *sound* one; that is, any property proven for the well-nested shuffle (using standard techniques) is guaranteed to hold for the original program.

In Section 7, we outline how the language-theoretic (automata or grammar) description of a product program can be turned into a classic recursive program and encoded for standard safety verification, which serves as the last step of our reduction of the problem of hypersafety verification of recursive programs to standard safety verification.

We have a tool called HyRec, that reduces an input hypersafety instance to a classic safety instance by constructing the appropriate product programs. We experiment with the a large set (103) of benchmarks to show that the product programs are absolutely vital for hypersafety verification of recursive programs. Specifically, our experiments with 103 benchmarks confirm that the high level idea of the product program is intuitively known to the programmer and can be succinctly specified and facilitates the verification of the hypersafety instance.

2 Background

2.1 Languages, Automata, and Grammars

Visibly pushdown langauges [2] are a well-established formal model for capturing the set of *interprocedurally realizable paths* [50, 51] of a program: procedure calls always return to the correct location, but the (data) feasibility of the paths is not considered.

We review some key concepts for visibly pushdown languages, and refer the reader to [2] for a full exposition. A *visibly pushdown alphabet* (VP alphabet) $\widetilde{\Sigma}$ is the union of three disjoint finite alphabets, Σ^{call} (*calls*), Σ^{ret} (*returns*), and Σ^{int} (*internals*). We use $\widetilde{\Sigma}$ when we want to emphasize the alphabet is visibly pushdown and use Σ for a general alphabet. A word $w_1 \cdots w_l \in \widetilde{\Sigma}^*$ induces a unique *matching relation* [3] $\rightsquigarrow$ over $\{-\infty, 1, 2, \ldots, l\} \times \{1, 2, \ldots, l, +\infty\}$ so that $i \rightsquigarrow j$ $(-\infty < i < j < +\infty)$ means w_i, w_j are a pair of call and return that match in the usual sense, and $i \rightsquigarrow +\infty$ (resp. $-\infty \rightsquigarrow i$) means i is a *pending* call (resp. return). A word without pending letter is *well-matched*.

A *visibly pushdown automaton* (VPA) [2] over a VP alphabet is a pushdown automaton whose stack operations are entirely controlled by the type of the letter read: it pushes when reading a call, pops when reading a return, and neither pushes nor pops when reading an internal.

Definition 2.1 (Visibly Pushdown Automaton (VPA)). A visibly pushdown automaton over a VP alphabet $\widetilde{\Sigma}$ is a tuple $M = (Q, Q^{\mathsf{in}}, \Gamma, \delta, Q^{\mathsf{F}})$ where Q is a finite set of states, $Q^{\mathsf{in}} \subset Q$ is a set of initial states, Γ is a finite stack alphabet that contains a special bottom-of-stack symbol $\perp$, $\delta = \delta^{\mathsf{call}} \cup \delta^{\mathsf{ret}} \cup \delta^{\mathsf{int}}$ is a set of transitions, where $\delta^{\mathsf{call}} \subset Q \times \Sigma^{\mathsf{call}} \times Q \times (\Gamma \setminus \{\perp\})$, $\delta^{\mathsf{ret}} \subset Q \times \Sigma^{\mathsf{ret}} \times \Gamma \times Q$, and $\delta^{\mathsf{int}} \subset Q \times \Sigma^{\mathsf{int}} \times Q$, and $Q^{\mathsf{F}} \subset Q$ is a set of final states.

We say a VPA is complete if at any state, reading any letter leads to a next state. A *visibly pushdown language* (VPL) is a language recognized by some VPA. VPLs can also be characterized by *visibly pushdown grammars* (VPGs) [2], a subclass of context-free grammars. We are mainly concerned with languages of well-matched words, which are generated by *well-matched VPGs* [39]. We additionally permit a grammar to have more than one start symbols.

Definition 2.2 (Well-matched VPG). A context-free grammar $G = (V, \widetilde{\Sigma}, P, S)$ is a *well-matched VPG* if every production rule in P has the form $X \to \epsilon$, $X \to aY$, or $X \to cYrZ$ where $a \in \Sigma^{\mathsf{int}}$, $c \in \Sigma^{\mathsf{call}}$, and $r \in \Sigma^{\mathsf{ret}}$.

We recall the *shuffle* operator $\|$ over languages. For any alphabet Σ and $\Sigma' \subset \Sigma$, let the projection function $\Pi_{\Sigma'} : \Sigma^* \to \Sigma'^*$ be the string homomorphism that erases letters that are not in Σ'.

Definition 2.3 (Shuffle). Let Σ be an alphabet that can be partitioned into a disjoint union $\Sigma_1 \uplus \cdots \uplus \Sigma_n$. Let L_i be a language over alphabet Σ_i $(i = 1, \ldots, n)$. The *shuffle* $L_1 \| \cdots \| L_n$ is the set of words $w \in \Sigma^*$ such that $\Pi_{\Sigma_i}(w) \in L_i$ for each i.

Regular languages are closed under shuffle, but this is not true for VPLs, even in the case where L_i's are copies of the same language. For instance, the shuffle of two Dyck languages [19] over disjoint VP alphabets (opening/closing delimiters are calls/returns) is not context-free [37].

2.2 Recursive Programs

Let V be the set of all valuations, i.e., maps from program variables to values, and $\mathrm{Stack}(V)$ be the set of all nonempty stacks whose frames are elements of V. We use $S.\nu$ to denote a stack whose top frame is ν and the rest is S, which is a smaller stack.

Following [35], we model a recursive program as a VPL P over a VP alphabet $\widetilde{\Sigma}$ and a semantics $\mathcal{S} : \widetilde{\Sigma} \to \mathcal{P}(\mathrm{Stack}(V) \times \mathrm{Stack}(V))$ which respects the VP alphabet $\widetilde{\Sigma}$: the type of stack operation performed by $\mathcal{S}[\![a]\!]$ is determined by the type of a. For our purpose, we assume P is well-matched. Formally, there is a map $\mathcal{F} \subset (\Sigma^{\mathsf{call}} \times V \times V) \cup (\Sigma^{\mathsf{ret}} \times V \times V \times V) \cup (\Sigma^{\mathsf{int}} \times V \times V)$ such that

- If $c \in \Sigma^{\mathsf{call}}$, then $\mathcal{S}[\![c]\!] := \{(S.v,\ S.v.v') : v' \in \mathcal{F}[\![c]\!](v)\}$.
- If $r \in \Sigma^{\mathsf{ret}}$, then $\mathcal{S}[\![r]\!] := \{(S.v_<.v,\ S.v') : v' \in \mathcal{F}[\![r]\!](v, v_<)\}$.
- If $a \in \Sigma^{\mathsf{int}}$, then $\mathcal{S}[\![a]\!] := \{(S.v,\ S.v') : v' \in \mathcal{F}[\![a]\!](v)\}$.

Extend $\mathcal{S}$ from letters to syntactic runs (words over $\widetilde{\Sigma}$ without pending return) via relation composition. For a syntactic run ρ, the hoare triple $\{\mathsf{pre}\}\rho\{\mathsf{post}\}$ holds if for all $(S.v,\ S'.v') \in \mathcal{S}[\![\rho]\!]$, if $v \models \mathsf{pre}$, then $v' \models \mathsf{post}$. We say ρ is *feasible* if $\mathcal{S}[\![\rho]\!] \neq \varnothing$. If ρ is infeasible, any triple $\{\mathsf{pre}\}\rho\{\mathsf{post}\}$ is vacuously true. Given a recursive program $(P, \mathcal{S})$, define $\{\mathsf{pre}\}P\{\mathsf{post}\}$ as $\forall \rho \in P.\ \{\mathsf{pre}\}\rho\{\mathsf{post}\}$. When $\mathcal{S}$ is clear from the context, we sometimes use just P to refer to the recursive program.

2.3 Commutativity and Reductions

Definition 2.4 (Commutativity-based Equivalence [46]). Given an irreflexive symmetric *commutativity relation* $I \subset \Sigma \times \Sigma$, define the *commutativity-based equivalence* $\equiv_I$ on Σ^* as the least transitive reflexive relation such that for all $u, v \in \Sigma^*$ and $a, b \in \Sigma$, if $(a, b) \in I$, then $uabv \equiv_I ubav$.

It is straightforward to verify that $\equiv_I$ is indeed an equivalence relation. Intuitively, two words u, v are equivalent under $\equiv_I$ if v can be obtained from u by repeatedly swapping adjacent commuting letters. The equivalence relation $\equiv_I$ partitions a language into equivalence classes. If a subset of the language contains representatives from all equivalence classes, we call it a *reduction*.

Definition 2.5 (Reduction). Given a commutativity relation I on Σ, for a language L over Σ, a subset $\widehat{L}$ of L is said to be a *reduction* if for all $\rho \in L$, there exists $\sigma \in \widehat{L}$ such that $\sigma \equiv_I \rho$. A reduction is *minimal* if there is precisely one such σ for every ρ.

In our context, letters are statements and words are runs of a program. Thus, we are interested in not an arbitrary commutativity relation but one that is *sound*: if $(a, b) \in I$, then ab and ba have the same semantics. If $\widehat{P}$ is a reduction of P under a sound commutativity relation, then $\widehat{P}$ covers all semantics of P, which results in the following theorem:

Theorem 2.6 (Soundness of Reductions). *If $\widehat{P}$ is a reduction of P under a sound commutativity relation, then* $\{\mathsf{pre}\}\ \widehat{P}\ \{\mathsf{post}\}$ *if and only if* $\{\mathsf{pre}\}\ P\ \{\mathsf{post}\}$.

3 Reductions Space for Hypersafety Verification

The classic way [8] to verify a hypersafety property for a recursive procedure is to produce n disjoint copies $P_1, \ldots, P_n$ from it and state the pre/postconditions pre and post as a property of the sequential composition of these disjoint copies: $\{\mathsf{pre}\}\ P_1; \ldots; P_n\ \{\mathsf{post}\}$. Observe that the same disjointness assumption implies that the sequential composition can be replaced by a parallel composition (or the shuffle). Hence we can state our goal as constructing a single (product) recursive program $\widehat{P}$ such that for any hypersafety property expressed as a pair of pre/postconditions pre and post:

$$\{\mathsf{pre}\}\ P_1 \parallel \ldots \parallel P_n\ \{\mathsf{post}\} \iff \{\mathsf{pre}\}\ \widehat{P}\ \{\mathsf{post}\} \qquad \text{(Soundness)}$$

which makes it sound to verify $\widehat{P}$ instead. The parallel product is more useful for our purpose, because it already includes every possible imaginable alignment of individual program runs inside it. We start by assuming that the programs P_i's are entirely disjoint, and later (in Section 5) address the concurrency case where some may share variables.

Each P_i is a classic recursive program over a disjoint memory and, in the classic sense, can be interpreted using a stack semantics in the style outlined in 2.2. In order to track the semantic meaning of an arbitrary run of a *product* of these programs, which may align the executions from the programs in an arbitrary way, the semantic function has to maintain n stacks, one for each component, to keep track of each of the n disjoint recursive computations. In other words, the

global state is now a *tuple* of n stacks, and a statement from the i-th component operates on the i-th stack from this tuple. To get a classic recursive program, we need this tuple to collapse into a single stack while preserving the ability to keep track of the computation of any arbitrary component P_i. This can be guaranteed precisely when the alignments are *well-nested*.

Definition 3.1 (Well-nested Words). Let $\widetilde{\Sigma} = \widetilde{\Sigma}_1 \uplus \cdots \uplus \widetilde{\Sigma}_n$, with disjoint $\widetilde{\Sigma}_i$'s. A word $w \in \widetilde{\Sigma}^*$ is *well-nested* if $i \rightsquigarrow j$ (where $\rightsquigarrow$ is the matching relation on w) implies that w_i and w_j both belong to the same $\widetilde{\Sigma}_k$. For any language $L \subset \widetilde{\Sigma}^*$, $\mathsf{wn}(L)$ denotes the set of all well-nested words in L.

If a language over $\widetilde{\Sigma}$ contains only well-nested words, we say the language is well-nested. Note that being well-nested is different from being well-matched (see Section 2.1).

We need a suitable representation for individual P_i's and $\widehat{P}$ that facilitates this goal. The control flow graph (or equivalent) representations lose track of the recursive structure of the runs. Context free grammars (CFG) or a pushdown automaton (PA) are the classic models that maintain this structure. However, *visibly pushdown languages* (VPL) [2], or alternatively the *regular nested word languages* (RNWL) [3], are a strict subset of context free languages that are expressive enough to model recursive programs and yet have the same desirable boolean properties as regular languages.

It is straightforward to take as input the code of a recursive component P_i and construct a *visibly pushdown grammar* (VPG) or *visibly pushdown automaton* (VPA) that precisely captures the set of well-matched syntactic runs of P_i. We abuse the notation P_i to refer both to the program and the set of syntactic runs, which form a well-matched VPL, since the distinction is always clear from context. We can always ensure that each P_i is over a distinct visibly pushdown alphabet $\widetilde{\Sigma}_i$, and introduce $\widetilde{\Sigma} = \widetilde{\Sigma}_1 \uplus \cdots \uplus \widetilde{\Sigma}_n$.

The language $P_{\|} = P_1 \parallel \ldots P_n$ is not necessary visibly pushdown, even if all P_i's are; nor does it exclude ill-nested alignments. In this section, we formulate how we can use simple observations from language theory and concurrency theory to define and algorithmically construct individual product programs (reductions) from $P_{\|}$, which (1) satisfy the Soundness condition, (2) only contain well-nested alignments, and (3) are visibly pushdown and therefore representable through VPAs and VPGs. It is very important to understand that we manipulate the language (i.e. the set of syntactic runs) to achieve this, and *do not* formulate this as a manipulation of any specific representation of it. So, even though we operate on sets of syntactic runs, we do not manipulate (bounded) syntactic structures (grammars, automata, control flow graphs, etc) to achieve this. Syntactic structures are bounded and assume bounded amount of manipulation; that is, unless one resorts to unfolding and therefore changing the structures. Languages are unbounded and as such present an unbounded amount of potential for selection of product programs.

3.1 Contextual Lexicographic Reductions

For every run $\rho \in P_{\|}$, it suffices that the reduction $\widehat{P}$ contains *at least one* permutation of statements of ρ while maintaining the order of the statements within each P_i. All other permutations are *equivalent* and it suffices that the entire *equivalence class* (see Definition 2.4) has at least one of its elements. This is a *commutativity* type argument, where pairs of statements from different copies can be considered to soundly commute, since they operate on disjoint memories. If the goal is to opt for programs with simpler proofs then we select *precisely one* representative, since it is generally easier to prove a program with fewer behaviours correct. Such a language $\widehat{P}$ is by definition a *minimal reduction* (see Section 2) of $P_{\|}$ under the commutativity relation $\mathbb{I} = \{(a, b) \mid a \in \widetilde{\Sigma}_i, b \in \widetilde{\Sigma}_j, i \neq j\}$, the largest sound commutativity relation.

Note that for any program that is not bounded, there are infinitely many equivalence classes of $\mathbb{I}$, and as long as these equivalence classes have more than one member, then *uncountably* many

minimal reductions (see Definition 2.5), each uniquely determined by a *selector* of representatives from the equivalence classes under I.

Proposition 3.2. *In general, the set of* minimal reductions *of $P_{\|}$ under $\mathbb{I}$ is not* countable.

Our goal is to introduce an infinite yet finitely representable (and thus countable) subset of this. To this end, we are interested in a selector (from equivalence classes) that can be finitely represented. Inspired by [26, 28], we focus on selectors of the form $E \mapsto \min_{\preceq}(E)$, where E is an equivalence class, and $\preceq$ is a *contextual lexicographic order*, a generalization of the standard lexicographic orders. We define a *contextual order* on $\widetilde{\Sigma}$ as a map from $\widetilde{\Sigma}^*$ to strict partial orders on $\widetilde{\Sigma}$ (denoted $\mathrm{PO}(\widetilde{\Sigma})$); this allows the order on letters to vary according to the left context, which is a word over $\widetilde{\Sigma}$. Each contextual order on $\widetilde{\Sigma}$ determines a contextual lexicographic order on $\widetilde{\Sigma}^*$.

Definition 3.3 (Contextual Lexicographic Order). Given a contextual order $\prec\colon \widetilde{\Sigma}^* \to \mathrm{PO}(\widetilde{\Sigma})$, we lift $\prec$ to a *contextual lexicographic order* (CLO) on $\widetilde{\Sigma}^*$ by having $\sigma \preceq \rho$ if and only if

- σ is a prefix of ρ, or
- $\sigma = \alpha a \beta$ and $\rho = \alpha b \gamma$ for some $\alpha, \beta, \gamma \in \widetilde{\Sigma}^*$ and $a, b \in \widetilde{\Sigma}$ such that $a \prec_\alpha b$.

The standard lexicographic orders are the special case where the map $\prec$ is a constant function and its image is a total order.

Proposition 3.4. *If $\prec$ is a contextual order on $\widetilde{\Sigma}$, then the CLO induced by $\prec$ is a partial order on $\widetilde{\Sigma}^*$. Furthermore, if $\prec_x$ is a total order on $\widetilde{\Sigma}$ for every $x \in \widetilde{\Sigma}^*$, then the CLO induced by $\prec$ is total.*

Example 3.5 (A Round-Robin[1] CLO). Let $\widetilde{\Sigma}_1 = \{(,)\}$ and $\widetilde{\Sigma}_2 = \{[,]\}$ be the VP alphabet of parentheses and brackets respectively, and let $\widetilde{\Sigma} = \widetilde{\Sigma}_1 \uplus \widetilde{\Sigma}_2$. Define the contextual order $\prec$ as follows: for any words $x, w \in \widetilde{\Sigma}^*$ such that w is nonempty and well-matched,

$$\begin{aligned} x(&\mapsto [< (<) <] \\ x(w &\mapsto) < (< [<] \\ x[w &\mapsto] < [< (<) \\ \text{otherwise} &\mapsto (< [<] <) \end{aligned}$$

Intuitively, $\prec$ simulates round-robin scheduling and it enforces well-nestedness. Let $\preceq$ be the CLO induced by the contextual order $\prec$. Consider $\alpha = \texttt{([()])}$, $\beta = \texttt{(([]))}$, and $\gamma = \texttt{([()])}$. They are in the same equivalence class under $\mathbb{I}$, and we have $\alpha \preceq \beta$ and $\alpha \preceq \gamma$. Note that β does not follow a round-robin scheduling and γ is ill-nested, and α is selected over both by this CLO. □

If $\preceq$ is a CLO, the set of $\preceq$-minimal representatives of the equivalence classes, induced by a commutativity relation I, of a language L form a *lex reduction* of L.

Definition 3.6 (Lex Reduction). Let L be a language over $\widetilde{\Sigma}$ and $I \subseteq \widetilde{\Sigma} \times \widetilde{\Sigma}$ a commutativity relation. The reduction of L induced by contextual order $\prec$ on $\widetilde{\Sigma}$ is defined as

$$\mathrm{red}_{I,\prec}(L) := \{w \in L \mid \forall u \in L :\ u \equiv_I w \wedge u \preceq w \implies u = w\},$$

where $\preceq$ is the CLO induced by $\prec$.

If $\preceq$ is total, then $\mathrm{red}_{I,\prec}(L)$ is a minimal reduction. For simplicity, we assume from now on that the contextual orders in consideration map words over $\widetilde{\Sigma}$ to strict total orders on $\widetilde{\Sigma}$ (denoted $\mathrm{TO}(\widetilde{\Sigma})$), which is a sufficient (but not necessary) condition for the order yielding a minimal reduction. Recall that $\mathbb{I}$ is the relation that declares any letters from distinct alphabets as commuting. Since we are

[1] Round-robin here is an intuitive simplification of (1,1)-lockstep, whose formal definition is presented in Section 6.1.

strictly interested in this commutativity relation, we use the shorthand notation $\mathrm{red}_{\prec}$ in place of $\mathrm{red}_{\mathbb{I},\prec}$, since the commutativity relation will always be $\mathbb{I}$.

Example 3.7. We continue in the setting of Example 3.5. Observe that for *any* $P_1 \subset \{(^n)^n : n > 0\}$ and $P_2 \subset \{[^n]^n : n > 0\}$, the reduction $\mathrm{red}_{\prec}(P_1 \parallel P_2)$ is precisely the round-robin like scheduling of P_1 and P_2 that also enforces well-nestedness.

If I is a sound commutativity relation (such as $\mathbb{I}$ in the hypersafety context), then any reduction under I can be verified in place of the original program (Theorem 2.6). This naturally applies to lex reductions, which we restate as follows:

Theorem 3.8 (Soundess of Lex Reductions). *Let I be a sound commutativity relation over $\widetilde{\Sigma}^*$ and $\prec$ a contextual order on $\widetilde{\Sigma}$. The lex reduction $\mathrm{red}_{I,\prec}(P_1 \parallel \ldots \parallel P_n)$ satisfies the Soundness condition.*

We conclude our discussion around lex reductions with the following observations, which motivate the content of the next section.

Proposition 3.9. *There exists a contextual order $\prec$ such that $\mathrm{red}_{\prec}(P_1 \parallel P_2)$ is not context-free. There also exists a contextual order $\prec$ such that $\mathrm{red}_{\prec}(P_1 \parallel P_2)$ contains ill-nested words.*

3.2 Visibly Pushdown Well-Nested Lex Reductions

The conclusion from the results presented in the previous section is that we need to further restrict the set of lex reductions so that we can get reductions that are well-nested and visibly pushdown. We achieve this by restricting the set of contextual orders in consideration.

Definition 3.10 (Visibly Pushdown Contextual Order (VPO)). A contextual order $\prec\colon \widetilde{\Sigma}^* \to \mathrm{TO}(\widetilde{\Sigma})$ is *represented by a deterministic VPA* $A = (Q_A, Q_A^{\mathsf{in}}, \Gamma_A, \delta_A, Q_A^{\mathsf{F}})$ if A is complete and there is a map $\mathrm{ord} : Q_A \to \mathrm{TO}(\widetilde{\Sigma})$ such that for any $w \in \widetilde{\Sigma}^*$, if the run of A on w ends at state q, then $\prec_w$ is $\mathrm{ord}(q)$. We call such orders *visibly pushdown contextual orders*.

An alternative way of interpreting this definition, for the reader unfamiliar with language theory, is to think about the contextual order to be definable through a scheduler that has the same computation power as a visibly pushdown automaton.

Example 3.11. The contextual order $\prec$ in Example 3.5 is visibly pushdown, which can be seen as follows. Under the map $\prec$, the preimage of each order on $\widetilde{\Sigma}$ is a VPL: the four different types of prefixes (i.e., context) that would lead to the four different choices of orders are all recognizable by deterministic VPA. Then, the product of the four VPAs (as in the proof of closure under intersection [2]) can represent the contextual order $\prec$.

The set of all minimal runs defined by any visibly pushdown order is visibly pushdown. In other words, if the order is VP, then all schedules (independent of the choice of language) that are accepted by the order form a VP language.

Proposition 3.12. *If $\prec$ is a VPO, then $\mathrm{red}_{\prec}(\widetilde{\Sigma}^*)$ is visibly pushdown.*

Yet, this result is not strong enough to imply that a minimal reduction of $P_1 \parallel \cdots \parallel P_n$ induced by a VPO is visibly pushdown or well-nested.

Proposition 3.13. *There exists a VPO $\prec$ such that $\mathrm{red}_{\prec}(P_1 \parallel P_2)$ is not context-free. There also exists a VPO $\prec$ such that $\mathrm{red}_{\prec}(P_1 \parallel P_2)$ contains ill-nested words.*

What is the missing piece? Interestingly, there is a correlation between the reduction being well-nested and it being visibly pushdown. Namely, if we enforce well-nestedness, we get visibly pushdown for free.

Theorem 3.14 (Visibly Pushdown Lex Reduction). *If $\prec$ is visibly pushdown and $\mathrm{red}_{\prec}(P_1 \parallel P_2) \subset \mathsf{wn}(P_1 \parallel P_2)$, then $\mathrm{red}_{\prec}(P_1 \parallel P_2)$ is visibly pushdown.*[2]

The problem is that the above condition is dependent on the specific choices of P_i's and is rather non-constructive (or equivalently hard to check). To obtain a condition on the order alone, independent of P_i's, we introduce the concept of *coherent contextual orders*. Intuitively, a coherent contextual order deprioritizes any returns that do not match the last open/pending call, because if the non-matching return is prioritized, then one ends up violating well-nestedness. Formally:

Definition 3.15 (Coherent Contextual Order). Let $\widetilde{\Sigma}_i$ be VP alphabets ($i = 1, 2$) and $\widetilde{\Sigma} = \widetilde{\Sigma}_1 \uplus \widetilde{\Sigma}_2$. A contextual order $\prec$ on $\widetilde{\Sigma}$ is said to be *coherent* if for any word $u \in \widetilde{\Sigma}^*$ and $i \in \{1, 2\}$, if u has pending calls and the last one is from Σ_i^{call}, then for any letter $a \in \widetilde{\Sigma}_i$ and $r \in \bigcup_{j \neq i} \Sigma_j^{\mathsf{ret}}$, we have $a \prec_u r$.

Example 3.16. The contextual order $\prec$ in Example 3.5 is coherent. For instance, if $u = x($w, then the last pending call in u is $(\in \Sigma_1^{\mathsf{call}}$, and we can verify that $a \prec_u]$ for all $a \in \widetilde{\Sigma}_1$.

For specific languages P_1, P_2, coherency is not a necessary condition for the well-nestedness of $\mathrm{red}_{\prec}(P_1 \parallel P_2)$. For instance, the word u that witnesses the violation of coherency may not occur as a prefix in the shuffle $P_1 \parallel P_2$ at all. However, coherency is always a sufficient condition.

Proposition 3.17. *Let $\prec$ be a coherent contexual order on $\widetilde{\Sigma} = \widetilde{\Sigma}_1 \uplus \widetilde{\Sigma}_2$ and P_1, P_2 be well-matched VPLs over $\widetilde{\Sigma}_1, \widetilde{\Sigma}_2$ respectively. Then $\mathrm{red}_{\prec}(P_1 \parallel P_2) \subset \mathsf{wn}(P_1 \parallel P_2)$.*

Note that a coherent contextual order is not necessarily visibly pushdown, and vice versa. As a consequence of Theorem 3.14 and Proposition 3.17, we can conclude that the combination of the two properties in an order is sufficient to guarantee that it induces a reduction that is both visibly pushdown and well-nested.

Theorem 3.18 (VP and Coherent Contextual Order). *If $\prec$ is a visibly pushdown* **and** *coherent contextual order, then $\mathrm{red}_{\prec}(P_1 \parallel P_2)$ is visibly pushdown and well-nested.*

Once we know a reduction is visibly pushdown, we know that it is theoretically constructible. However, in the next section, we take a closer look at what is the best way of algorithmically constructing a visibly pushdown and well-nested $\mathrm{red}_{\prec}(P_1 \parallel P_2)$.

4 Constructing Reductions

A language L over $\widetilde{\Sigma}$ is called *closed* under a commutativity relation I if each equivalence class of I is either entirely included in L or none of the members appear in L. For example, $P_1 \parallel \ldots \parallel P_n$ is by definition closed under any commutativity relation that does not reorder statements from the same P_i. It is straightforward to see that if L is closed under I then

$$\mathrm{red}_{I,\prec}(L) = \mathrm{red}_{I,\prec}(\widetilde{\Sigma}^*) \cap L. \tag{1}$$

This simple observation puts forward an algorithmic path for the construction of the reductions in the iterative case [26–28]. The generic language $\mathrm{red}_{I,\prec}(\widetilde{\Sigma}^*)$, which is somewhat independent of any given verification instance and is purely defined based on an alphabet of statements, can be constructed by an adaptation of the idea of *sleep sets* [29]. Then, any reduction can be constructed through a simple construction of the intersection of this generic language and the baseline parallel product $P_1 \parallel \ldots \parallel P_n$. The proof of Theorem 3.18 puts forward a similar type of construction for the visibly pushdown case. However, this is unsatisfactory in two orthogonal ways:

[2]For notational simplicity, the statement is made only for two components, but this theorem and the ones below also hold for $n > 2$ components.

(1) *Complexity:* the construction is generic and does not exploit properties of the commutativity relation. We are primarily interested in the commutativity relation $\mathbb{I}$, and as we demonstrate in Section 4.3, there is a construction that is strictly simpler under the assumption that all copies fully commute and the stack alphabet is small.
(2) *Burden on user:* Theorem 3.18 demands two independent conditions on the order. The visibly pushdown condition on the order is rather trivial; if the order can be expressed in a specific template (grammar or automata), then it is visibly pushdown. However, the coherence condition is an extra burden to check/verify, which should ideally be avoidable.

Here, we give an alternative construction of VP lex reductions that overcomes these restrictions.

4.1 Well-nested Shuffle

Recall that the role of the *coherence* condition is to ensure that the reduction only includes well-nested alignments, or in other words, it is a subset of $\mathsf{wn}(P_1 \parallel \ldots \parallel P_n)$. It turns out that the latter is an object that is interesting on its own, and we can define it in a more elegant way using a new binary composition operator:

Definition 4.1 (Well-Nested Shuffle). For languages $L_1, \ldots, L_n$ respectively over disjoint alphabets $\widetilde{\Sigma}_1, \ldots, \widetilde{\Sigma}_n$, define the *well-nested shuffle* $L_1 \between \cdots \between L_n$ as the set of well-nested words in $L_1 \parallel \cdots \parallel L_n$.

We observe that well-nested shuffle is associative, so the well-nested shuffle of n languages for any $n > 1$ is well-defined. For well-matched languages L_1, L_2, we have $L_1; L_2 \subset L_1 \between L_2 \subset L_1 \parallel L_2$.

Theorem 4.2 (Closure). *Visibly pushdown languages are closed under the well-nested shuffle.*

This is significant, because the same is not true for the standard shuffle operator. If we view programs $P_1, \ldots, P_n$ as languages of their behaviours, then the program $P_1 \between \cdots \between P_n$ is the maximal, in terms of the set of its behaviours, program consisting of arbitrary alignments of behaviours of $P_1, \ldots, P_n$ that can be executed using a single stack.

The well-nested shuffle is a construction of interest, independent of the specific problem of constructing lex reductions. We further remark on this in Section 5.

4.2 Visibly Pushdown Lex Reductions as Reductions of The Well-Nested Shuffle

When a contextual order $\prec$ defines a reduction of $P_1 \parallel \cdots \parallel P_n$, coherency of $\prec$ guarantees that only well-nested words are selected from it. If we can alternatively define a reduction as a restriction of $P_1 \between \cdots \between P_n$, it will by definition include only well-nested words. There is, however, a major technical challenge in achieving this goal: the construction path of Theorem 3.18 relies on the validity of Equation 1, which in turn relies on the *closedness* of the language. $P_1 \between \cdots \between P_n$ is not closed (w.r.t. $\mathbb{I}$), because swapping a pair of adjacent calls or returns may break well-nestedness.

Our alternative construction relies on a new insight: any VPO $\prec$ can be repaired to a *coherent* (Definition 3.15) VPO $\prec'$ such that for any well-matched $\rho, \tau \in \mathsf{wn}(\widetilde{\Sigma}^*)$ such that $\rho \equiv_{\mathbb{I}} \tau$, we have $\rho \preceq \tau \Rightarrow \rho \preceq' \tau$. We can construct $\prec'$ as follows: for any $a, b \in \widetilde{\Sigma}$ and $u \in \widetilde{\Sigma}^*$, with $R_k := \bigcup_{j \neq k} \Sigma_j^{\mathsf{ret}}$,

$$a \prec'_u b \equiv \begin{cases} ((a \in R_k \leftrightarrow b \in R_k) \wedge a \prec_u b) \vee (a \notin R_k \wedge b \in R_k), & \text{if } u \text{ has a last pending call from } \widetilde{\Sigma}_k \\ a \prec_u b. & \text{otherwise} \end{cases}$$

Lemma 4.3. *For any VPO $\prec$, there exists a coherent VPO $\prec'$ such that* $\mathrm{red}_{\prec}(P_1 \between \cdots \between P_n) = \mathrm{red}_{\prec'}(P_1 \parallel \cdots \parallel P_n)$ *for any well-matched languages $P_1, \ldots, P_n$ over disjoint VP alphabets.*

This leads to the key theorem of this section that links a family of VPL reductions to the family of visibly pushdown contextual orders by construction:

Theorem 4.4 (Visibly Pushdown Lex Reduction). *If $\prec$ is a visibly pushdown contextual order and $P_1, \ldots, P_n$ are VPLs, then so is* $\mathrm{red}_{\prec}(P_1 \between \cdots \between P_n)$.

The theorem is a consequence of Theorem 3.18 (a generic construction via sleep sets) and Lemma 4.3 and yields an automaton of $O(q^n q_A n\, 2^{|\widetilde{\Sigma}|})$ states, assuming the VPA for each P_i has $O(q)$ states and the VPA representing the contextual order has $O(q_A)$ states. Next, we argue that under the assumption of full commutativity, one can do better than this.

4.3 Optimized Construction Specific to Hypersafety

Remarkably, it turns out that one can exploit the assumption of a full commutativity relation, applicable in all hypersafety verification instances, to devise a *simple* construction for the VPA, resulting in a more efficient algorithm. Consider two singleton well-matched languages $\{\rho_1\}$ and $\{\rho_2\}$. Under the assumption of full commutativity between the two alphabets, the lex reduction $\mathrm{red}_{\prec}(\{\rho_1\} \between \{\rho_2\})$, which consists of exactly one word, can be computed in a greedy manner: If we have correctly interleaved the prefixes ρ_1', ρ_2' of ρ_1, ρ_2 as σ, then the next letter is the smaller one of the letters following ρ_1', ρ_2' that do not break well-nestedness: if σ contains a pending call and the last one is from $\widetilde{\Sigma}_i$, then the next letter is not from any Σ_j^{ret} such that $j \neq i$.

With modest assumptions on the order, this idea can be applied to the construction of $\mathrm{red}_{\prec}(P_1 \between P_2)$ as well. An order $\prec$ is *uniform* w.r.t. VPA P_1, P_2 if for any pair of states q_1, q_2 from P_1, P_2, for any $w \in \widetilde{\Sigma}^*$, $\prec_w$ ranks outgoing letters of q_1, q_2 consistently, i.e., if some outgoing letter from q_1 is less than some outgoing letter from q_2, then all outgoing letters from q_1 are less than all outgoing letters from q_2. If the VPAs are directly obtained from code, the only operational restriction is that the `then` and `else` branches of a component have to be ordered the same way against actions of another component. The uniformity condition is not practically limiting, as one can always adjust P_1 and P_2 (while preserving the language) to accommodate the order. For instance, any VPA can be converted in linear time to one such that for every state, the outgoing letters are all internals, all calls, or all returns. For a uniform VP contextual order, we can construct a compact VPA for $\mathrm{red}_{\prec}(P_1 \between P_2)$ in the same greedy manner:

Theorem 4.5. *Let $\prec$ be a uniform VP order represented by VPA A. Then the lex reduction* $\mathrm{red}_{\prec}(P_1 \between P_2)$ *is recognized by a VPA with $O(n_1 n_2 n_A n_\Gamma)$ states, where n_1, n_2, and n_A are the number of states of P_1, P_2, and A respectively and n_Γ is the sum of the size of the stack alphabets of P_1 and P_2.*

As outlined in Section 8, one can convert an automaton to a grammar using a standard construction [3] or verify this automaton directly. Directly constructing a VPG for the same language is not straightforward, since the contextual information (used by the contextual order) needs to propagate not in a top-down manner but through the linear and hierarchical edges.

5 Sound Reductions for Concurrency

As an underapproximation of the shuffle, the well-nested shuffle and its lex reductions are classic recursive programs that can always be used for *bug finding* for recursive concurrent programs.

In the hypersafety context, it is sound to consider only behaviours of $P_1 \parallel \cdots \parallel P_n$ that are subsets of $P_1 \between \cdots \between P_n$, since under the assumption that P_i's are fully disjoint, we have

$$\{\mathrm{pre}\}\ P_1 \between \cdots \between P_n\ \{\mathrm{post}\} \implies \{\mathrm{pre}\}\ P_1 \parallel \ldots \parallel P_n\ \{\mathrm{post}\}. \tag{2}$$

In the presence of shared memory, there may be dependency between different components. It is then no longer the case that any run of $P_1 \parallel \cdots \parallel P_n$ can be *soundly* rearranged to be well-nested and condition 2 may no longer hold. We outline under what assumptions about P_i's, we can recover this property for shared memory concurrent programs, so the well-nested shuffle can also be soundly *verified* in place of the original recursive concurrent program.

Consider a simple case of tail-recursive programs first. Since a tail-recursion can be transformed to a standard iteration, one intuitively expects this case to be problem-free, and it is. Assume P_1 and P_2 each make a single call of a tail-recursive function, resulting in runs $\rho_1 \in P_1$ and $\rho_2 \in P_2$. Since a return statement is always a local operation and therefore soundly commutes with any statement from other components, one can commute the return statements to make any interleaving of ρ_1 and ρ_2 well-nested while preserving semantics. For example, below, the last (pink) return can be commuted to turn the run to a semantically equivalent well-nested one.

$$[\,.\,.\,.[\,.\,.\,.[\,.\,.\,.(\,.\,.\,.(\,.\,.\,.)\,]\,]\,]\,]\,) \longrightarrow [\,.\,.\,.[\,.\,.\,.[\,.\,.\,.(\,.\,.\,.(\,.\,.\,.)\,)\,]\,]\,]$$

One can further relax the tail-recursion condition to *tail-independence*. For any $\widetilde{\Sigma}_i$, let $\widetilde{\Sigma}_i^{\mathsf{indep}}$ be all letters in $\widetilde{\Sigma}_i$ that soundly commute with all letters in other components.

Definition 5.1 (Tail/Head-Independent). We call a component P_i *tail-independent* (resp. *head-independent*) if every run ρ of P_i can be written as uv $(u, v \in \widetilde{\Sigma}_i^*)$ such that

- all calls are in u and all returns are in v, except those at a position j such that all letters between j and its matching position are in $\widetilde{\Sigma}_i^{\mathsf{indep}}$; and
- all letters in v (resp. u) are in $\widetilde{\Sigma}_i^{\mathsf{indep}}$.

This definition is effective, since given alphabets $\widetilde{\Sigma}_i$ and $\widetilde{\Sigma}_i^{\mathsf{indep}}$, one can construct a VPL P_i^{max} such that a component P_i is tail-independent iff $P_i \subset P_i^{\mathsf{max}}$. The same holds for head-independence. Hence, for any concurrent recursive program, one can algorithmically check if Defintion 5.1 applies.

Proposition 5.2. *Given a component P_i as a VPA and the alphabet $\widetilde{\Sigma}_i^{\mathsf{indep}}$, it is decidable in polynomial time whether P_i is tail-independent (or head-independent).*

We can then conclude conditions under which the well-nested shuffle can be soundly used for verification of a concurrent program:

Theorem 5.3 (Soundness of Well-Nested Shuffle Under Concurrency). *If every component P_i is tail-independent (or respectively head-independent) then condition (2) is satisfied.*

The well-nested shuffle can be verified as a classic recursive program using classic verification techniques, without concurrency posing an additional complication. Moreover, as we discuss in Section 4.2, any sound reduction of it can also be used for the purpose of verification.

6 Canonical Reductions and Their Construction

In Section 3, we introduce lex reductions as an infinite family of recursive product programs. Here, we introduce a simpler (still infinite) subfamily of lex reductions that are defined based on a set of building blocks, called *canonical reductions*, which are intuitive for the programmer to understand, compose, and use as annotations in hypersafety verification of recursive programs.

In Section 6.1, we give the formal definitions, and show that this new family is a subset of the lex reductions. As such, the results from the previous section support the construction of these in the style presented in Section 4.3. We observe, however, that for this sub-family, one can follow the inductive definitions and provide a direct VPG construction of a reduction from the family. This construction yields smaller grammars compared to that of Section 4.3 as we illustrate in Section 8.

6.1 Canonical Reductions for Recursive Programs

We start by giving a formal definition for the nested concatenation, motivated by the distributivity property of `div` from Section 1, and then one for the *parametric lockstep*. We define them on words for simplicity, since one can lift the definition from words to languages (i.e., sets of words) in the

standard way. We formally argue that these are instances of lex reductions, and then use an example to discuss the expressive power of the simple subfamily.

Nested Concatenation. The rules in Fig. 3 inductively define a ternary relation $w_1 \oplus w_2 \to w$, where w_1 and w_2 are well-matched words. The *nested concatenation* $w_1 \oplus w_2$ is the word w such that $w_1 \oplus w_2 \to w$, which can be shown to be uniquely defined and well-matched. We generalize this from two to n words by letting $w_1 \oplus w_2 \cdots \oplus w_n = w_1 \oplus (w_2 \oplus \cdots \oplus w_n)$.

$$\frac{}{\epsilon \oplus w_2 \to w_2}\ \text{NC-}\epsilon \qquad \frac{w_1 \oplus w_2 \to w'}{a w_1 \oplus w_2 \to a w'}\ \text{NC-int} \qquad \frac{w_1 \oplus w_2 \to w'}{c w_1 r v \oplus w_2 \to c w' r v}\ \text{NC-call}$$

Fig. 3. Definition of nested concatenation: c, r, and a respectively stand for a call, a return, or an internal statement. In $cwrv$, the r is meant to be the return that matches c.

Parametric Lockstep. Next, we define a lockstep product that is parametric on a vector $\boldsymbol{s}$ of positive integers that provides the nominal speed of the movement of each component in the lockstep synchronization. We define a simple *modulo* decrement operation for such vectors to simplify our definition. For vectors of natural numbers $\boldsymbol{s}$ and $\boldsymbol{t}$ such that $|\boldsymbol{s}| = |\boldsymbol{t}| = n > 0$, $\mathbf{0} \le \boldsymbol{t} \le \boldsymbol{s}$ (componentwise), and $\boldsymbol{t} \neq \boldsymbol{s}$, define

$$\mathrm{dec}_{\boldsymbol{s}}(\boldsymbol{t}) := \begin{cases} (s_1 - 1, s_2, \ldots, s_n), & \text{if } \boldsymbol{t} = \mathbf{0} \\ (0, \ldots, 0, t_i - 1, t_{i+1}, \ldots, t_n). & \text{if } i = \min\{j : t_j > 0\} \end{cases}$$

$\mathrm{LS}_{\boldsymbol{s}}(w_1, \ldots, w_n)$ denotes the lockstep composition of the n well-matched words at the *speed* of $\boldsymbol{s}$. To inductively define it, we need an additional helper vector $\boldsymbol{t}$. The rules in Fig. 4 define a relation $\mathrm{LS}_{\boldsymbol{s},\boldsymbol{t}}(w_1, \ldots, w_n) \to w'$, where $\boldsymbol{s}$ and $\boldsymbol{t}$ are vectors satisfying the constraints above. The $\boldsymbol{s}$*-lockstep* of $w_1, \ldots, w_n$ is the unique word w' such that $\mathrm{LS}_{\boldsymbol{s},\mathbf{0}}(w_1, \ldots, w_n) \to w'$.

Note that the definition is inductive over the structure that exposes the *leftmost* letter of a word. This is intentional, for consistency with the definition of CLO (Definition 3.3), which looks at the *left context*. However, this is not the only valid choice, and we discuss the alternative in Section 6.2.

Subfamily of Lex Reductions. Crucially, we can show that the nested concatenation and $\boldsymbol{s}$-lockstep of well-matched VPLs over disjoint alphabets can be produced through a VP contextual order (VPO) as a lex reduction of the well-nested shuffle.

Proposition 6.1. *If $P_1, \ldots, P_n$ are well-matched VPLs over disjoint alphabets, then their concatenation, nested concatenation, and $\boldsymbol{s}$-lockstep are lex reductions of $P_1 \mathbin{⫴} \cdots \mathbin{⫴} P_n$ induced by VPOs.*

We point out that the VPOs used by the construction are uniform (see Section 4.3) w.r.t. VPA such that for every state, the outgoing letters are all internals, all calls, or all returns. Combining Proposition 6.1 and Theorem 4.4, we have the following theorem:

Theorem 6.2. *If $P_1, \ldots, P_n$ are well-matched VPLs over disjoint alphabets, then any reduction built from a combination of concatenation, nested concatenation, and $\boldsymbol{s}$-lockstep and one occurrence of each P_i is a VP reduction of $P_1 \mathbin{⫴} \cdots \mathbin{⫴} P_n$.*

Expressive Power. Superficially, nested concatenation and parametric lockstep may seem simple and trivial. As we illustrate empirically in Section 8, their compositions yields a powerful family that can solve nearly all our benchmarks. However, to discuss this in depth, let us look at an example borrowed from [14], which puts forward a solution for constructing *semantic* product programs. Our reductions are syntactic and as we remark in Section 8.4, this implies a strict limitation compared to semantic ones. However, as we demonstrate using the example below, the

$$\frac{}{\mathsf{LS}_{s,t}(w_1) \to w_1}\ \text{LS-single} \qquad \frac{\mathsf{LS}_{s',t'}(w_1,\dots,w_{j-1},w_{j+1},\dots w_n) \to w' \quad s',t' \leftarrow s_{-j},t_{-j}}{\mathsf{LS}_{s,t}(w_1,\dots,w_{j-1},\epsilon,w_{j+1},\dots w_n) \to w'}\ \text{LS-}\epsilon$$

$$\frac{\mathsf{LS}_{s,t}(w_1,\dots,w_n) \to w' \quad \text{none of } w_i \text{ where } i < m \text{ starts with an internal}}{\mathsf{LS}_{s,t}(w_1,\dots,w_{m-1},aw_m,w_{m+1},\dots,w_n) \to w'}\ \text{LS-int}$$

$$\frac{\mathsf{LS}_{s,\mathsf{dec}_s(t)}(w_1,c_2w_2r_2,\dots,c_nw_nr_n) \to u' \quad t = 0 \quad \mathsf{LS}_{s,t}(v_1,\dots,v_n) \to v'}{\mathsf{LS}_{s,t}(c_1w_1r_1v_1,\dots,c_nw_nr_nv_n) \to c_1u'r_1v'}\ \text{LS-call-1st}$$

$$\frac{\mathsf{LS}_{s,\mathsf{dec}_s(t)}(\dots,c_{m-1}w_{m-1}r_{m-1},w_m,c_{m+1}w_{m+1}r_{m+1},\dots) \to u' \quad t \neq 0 \quad m = \min\{i : t_i > 0\}}{\mathsf{LS}_{s,t}(c_1w_1r_1v_1,\dots,c_nw_nr_nv_n) \to c_mu'c_mv_m}\ \text{LS-call}$$

Fig. 4. Inductive defintion of parametric lockstep. s_{-j} denotes the vector obtained from s by dropping the j-th component, and similar for t_{-j}.

combination of canonical reductions with slight *customizations*, which operate on *unbounded* and *rich* syntax, can be surprisingly powerful.

Consider the functions f and g illustrated on the right and the goal of proving them equivalent. They both increment k by m for n times; the only difference is that in f, the number m is computed by an auxiliary function h as the sum of m 1's. They appear in Figure 12 from [14], where an iterative version of this is cited as a challenge for verification using product programs, even *semantically* constructed ones. To solve this in [14], the parameter m is bounded to a small constant.

```
def g(n: nat, m: nat, i: nat, k: nat) -> nat
   if i < n:
      k = k + m
      k = g(n, m, i + 1, k)
   return k
def f(n: nat, m: nat, i: nat, k: nat) -> nat
   if i < n:
      k = h(m, 0, k)
      k = f(n, m, i + 1, k)
   return k
def h(m: nat, j: nat, k: nat) -> nat
   if j < m:
      k = h(m, j + 1, k + 1)
   return k
```

Fig. 5. Challenging Example From [14]

In constrast, our methodology easily yields a solution. In each interleaving run, we intuitively want to match every call of f against one call of g. This is *almost* a $(1,1)$-lockstep, except that the formal definition in Fig. 4 dictates that *every* call and return participates in the scheduling, which undesirably forces the calls of h to also participate in the lockstep matching of the calls; as a result, the second recursive call of g are nested inside the first call of h. However, as we demonstrate in Section 6.2, this limitation in the formal definitions of the canonical reductions is superficial, and we can easily *customize* them to schedule only a selected subset of calls and returns (in this case, those of functions f and g) and treat the rest the same way as internal actions. Our tool HyRec can verify this instance when such a customized $(1,1)$-lockstep reduction is given as input.

6.2 Customizations of Canonical Reductions

In this section, we introduce two simple *customizations* of canonical reductions.

Scheduling Selected Calls/Returns Only. By default, every call and return participates in the scheduling of the canonical reductions, and our first customization is to allow only a selected subset to participate, and the rest stack operations $\Sigma' \subset \Sigma^{\text{call}} \cup \Sigma^{\text{ret}}$ are treated the same way as internal actions.

To achieve this, one cannot simply change the partition of the visibly pushdown alphabet, which breaks the mechanics of visibly pushdown languages; nor can we apply the NC-int rule or LS-int rule for runs that start with a call that does not participate in the scheduling, which breaks well-nestedness of the reduction. Instead, we modify the *contextual orders* $\prec$ for the canonical reductions (Proposition 6.1): Based on $\prec$, we can define another contextual order $\prec'$ that replaces letters in Σ' with internal letters, queries $\prec$, and returns the result. Then the reduction induced by $\prec'$ is exactly the customization that we want. Formally, let $f : \widetilde{\Sigma}^* \to \widetilde{\Sigma}^*$ be a letter-to-letter string homomorphism such that

$$f(a) \in \Sigma_i^{\mathsf{int}} \text{ if } a \in \Sigma' \cap \widetilde{\Sigma}_i \text{ for any } i, \text{ and } f(a) = a \text{ otherwise.} \tag{3}$$

Then $\prec'$ is given by the contextual order $f(\prec)$ defined by $\forall a, b \in \widetilde{\Sigma}.\ a\ f(\prec)_w\ b \Leftrightarrow f(a) \prec_{f(w)} f(b)$.

In general, $f(\prec)_w$ is not a linear order even when $\prec$ is a map $\widetilde{\Sigma}^* \to \mathrm{TO}(\widetilde{\Sigma})$, as we have been assuming; but in our use case, where f replaces excluded calls and returns with internals from the same component, such a definition is sufficient to induce a minimal reduction. Moreover, when $\prec$ is a VPO (as in Proposition 6.1), with a reasonable assumption on Σ', there is a contextual order *equivalent* to $f(\prec)$ that is a VPO:

Proposition 6.3. *Let Σ' be a subset of $\Sigma^{\mathsf{call}} \cup \Sigma^{\mathsf{ret}}$ such that for all $i \in \{1, \ldots, n\}$ and $c, r \in \widetilde{\Sigma}_i$ that match in some run of P_i, we have $c \in \Sigma' \Leftrightarrow r \in \Sigma'$. Let f be a letter-to-letter string homomorphism that satisfies (3). If $\prec$ is a VPO, then there is a VPO $\prec'$ such that $\mathrm{red}_{\prec'}(P_1 \,\|\!|\, \cdots \,\|\!|\, P_n) = \mathrm{red}_{f(\prec)}(P_1 \,\|\!|\, \cdots \,\|\!|\, P_n)$.*

The proof in Appendix C gives a construction for the VPO $\prec'$ given $\prec$ and f.

Right Alignment. The canonical reductions are defined over the structure that exposes the *leftmost* letter of a word, for consistency with the use of *left context* (Definition 3.3), which eventually leads to Proposition 6.1. However, this does not have to be the case: we can customize the *alignment* of canonical reductions. To have *right-aligned* canonical reductions, we consider well-matched words of the form ϵ, wa, and $vcwr$ and change the rules accordingly. Then Proposition 6.1 holds for the modified version of contextual lex reductions that look at the *right context*.

As an example, consider a benchmark from [48] about proving the monotonicity of the *Ackermann function* $A(m, n)$ over the second argument:

$$A(m,n) := \begin{cases} n+1, & \text{if } m = 0 \\ A(m-1, 1), & \text{if } m > 0 \wedge n = 0 \\ A(m-1, A(m, n-1)). & \text{if } m > 0 \wedge n > 0 \end{cases}$$

Intuitively, a $(1,1)$-lockstep reduction seems like the appropriate choice. However, let us consider a run ρ_1 that starts with calling $A(m, 0)$ and a run ρ_2 that starts with calling $A(m, n)$ $(m, n > 0)$. The run ρ_1 consists of one direct recursive call $A(m-1, 1)$, whereas the run ρ_2 consists of two direct recursive calls $a := A(m, n-1)$ and $A(m-1, a)$, in that order. We would want to match $A(m-1, 1)$ in ρ_1 against the *second* call $A(m-1, a)$ in ρ_2, since it is the second call that produces the final result and has the same argument $m-1$. A concise way to describe such a reduction is the *right-aligned* $(1,1)$-lockstep.

6.3 Direct Construction of Canonical Reductions

Given well-matched VPGs $G_1, \ldots, G_n$ describing the language of the syntactic runs of each component, the inductive definitions of the canonical reductions (Figures 3 and 4) inspire a way to directly construct them as VPGs. A syntactic assumption on the form of input VPGs facilitate this construction: for any productions $X \to \alpha$ and $X \to \beta$, we assume that the right-hand sides α, β

both start with a call, both start with an internal, or both are empty. Any VPG can be converted in linear time to an equivalent one in this form.

Nested Concatenation: One can view the construction as a type of *product* construction. The nonterminal symbols of the grammar G for the language $\mathcal{L}(G_1) \oplus \mathcal{L}(G_2)$ are of the form $[W_1, W_2]$, where W_1, W_2 are nonterminal symbols of G_1, G_2 respectively; $[W_1, W_2]$ is a start symbol if W_1, W_2 are start symbols of G_1, G_2 respectively. The productions of G are defined by the rules in Figure 6, where $(X \to \alpha) \in G$ denotes $(X \to \alpha)$ is a production in G, a slight abuse of notation. Observe the syntactic similarity between the rules of Figure 3 and those in Figure 6. The extra syntactic condition guarantees this simplicity of inference of a grammar from the inductive definition.

$$\frac{(Y_1 \to \epsilon) \in G_1 \quad (W_2 \to \alpha) \in G_2}{([Y_1, W_2] \to \alpha) \in G}\text{GN-}\epsilon \qquad \frac{(Y_1 \to aW_1) \in G_1 \quad (W_2 \to \alpha) \in G_2}{([Y_1, W_2] \to a[W_1, W_2]) \in G}\text{GN-int} \qquad \frac{(Y_1 \to cW_1rV) \in G_1 \quad (W_2 \to \alpha) \in G_2}{([Y_1, W_2] \to c[W_1, W_2]rV) \in G}\text{GN-call}$$

Fig. 6. VPG productions for nested concatenation

***s*-Lockstep:** The construction of $\boldsymbol{m}$-lockstep follows the same pattern. The nonterminal symbols of the grammar G of the $\boldsymbol{m}$-lockstep of $\mathcal{L}(G_1), \ldots, \mathcal{L}(G_n)$ have the form $[\boldsymbol{s}, \boldsymbol{t}, (W_1, \ldots, W_k)]$, where $\boldsymbol{s}, \boldsymbol{t}$ satisfy the constraints in Section 6.1, W_i is a nonterminal in G_i for each i, and the length and maximum element of $\boldsymbol{s}$ are bounded by those of $\boldsymbol{m}$. The rules for generating the grammar construction rules mimic those of the inductive definition in Figure 4.

There is, however, a small complication with an easy fix. In the rules for nested concatenation (in Figure 3), for every pattern in the conclusion, the parts that appear in the premises have a nonterminal symbol dedicated to it. For instance, aw_1 is a pattern in the inductive definition, and w_1 corresponds to nonterminal symbol W_1 in the grammar construction rules. This makes things straightforward. Now consider the rule "LS-call" (of figure 4).

There is a pattern $c_i w_i r_i v_i$ in the conclusion, yet the subword $c_i w_i r_i$ is isolated in the premise, for which there does not exist a nonterminal. We can fix this if, for every rule of the form $Y \to cWrV$ in G_i, we add a rule $Y' \to cWrE$ (where $E \to \epsilon$), so subwords generated by cWr can be referred to as Y'. The full set of production rules are listed in Figure 8 in Appendix D.2.

Algorithmic Construction: In an algorithmic view of this construction, we can maintain a queue of reachable symbols, so only the productions that are truly needed are added to the grammar, making the construction efficient. Otherwise, the grammar would end up having a lot of unreachable non-terminals. These VPG constructions exploit properties of the canonical reductions: the contextual information is propagated in a top-down manner. For an arbitrary VP contextual order, the contextual information propagates through the linear and hierarchical edges of the nested words, so the reduction does not necessarily admit a direct VPG construction in the style demonstrated in this section.

7 The Semantics of The Recursive Product Program

Recall that our full notation for a recursive program is a pair $(P, \mathcal{S})$ with P representing its syntax as a visibly pushdown language and $\mathcal{S}$ representing its semantics. Let $(P_1, \mathcal{S}_1), \ldots, (P_n, \mathcal{S}_n)$ be recursive programs over disjoint VP alphabets $\widetilde{\Sigma}_1, \ldots, \widetilde{\Sigma}_n$ respectively. Assume that we have constructed a product program $\widehat{P}$ that is sound, well-nested, and visibly pushdown. To complete its description, we need to discuss its semantics. Here, we discuss how to compose $\mathcal{S}_1, \ldots, \mathcal{S}_n$ into a single $\mathcal{S}$.

We assume the variable names from the individual programs are disjoint. Suppose $\mathcal{S}_k$ is induced by $\mathcal{F}_k$ (see Section 2.2) and $V = V_1 \uplus \cdots \uplus V_n$ where V_k is the set of all valuations for P_k. The trivial

solution would be to use n stacks, which corresponds to a semantics $\mathcal{M} : \widetilde{\Sigma} \to \mathcal{P}(\Pi_{j=1}^{n} \mathrm{Stack}(V_j) \times \Pi_{j=1}^{n} \mathrm{Stack}(V_j))$. Each component uses the corresponding stack from the tuple, and its semantic is defined in the standard way. For an interleaving run $\rho \in P_1 \parallel \cdots \parallel P_n$, the Hoare triple $\{\mathrm{pre}\}\rho\{\mathrm{post}\}$ is *valid* w.r.t. $\mathcal{M}$ if

$$\forall((S_1.v_1, \ldots, S_n.v_n), (S_1'.v_1', \ldots, S_n'.v_n')) \in \mathcal{M}[\![\rho]\!] : \bigcup_{j=1}^{n} v_j \models \mathrm{pre} \implies \bigcup_{j=1}^{n} v_j' \models \mathrm{post}.$$

$\mathcal{M}$ is the semantics where all commutativity reasoning happens: the commutativity relation $\mathbb{I}$ (see Section 3.1) is sound w.r.t. $\mathcal{M}$.

By construction, any product program $\widehat{P}$ is well-nested, so it conceptually runs on a single stack. However, if we just naively push and pop on the same stack, we run into a problem with scoping of variables from different components. For instance, suppose P_1 pushes and writes to variable x_1; if P_2 pushes next, then P_1 loses track of x_1, which is no longer on the top frame.

There is a simple solution to this problem. We interpret the letters slightly differently: let each frame in the single stack store the variables from *all* n recursive programs (recall that the variable names are assumed to be disjoint); when P_k pushes or pops the stack, we also copy over the irrelevant variables, i.e., those from P_j where $j \neq k$. Formally, we define a semantics $\mathcal{S} : \widetilde{\Sigma} \to \mathcal{P}(\mathrm{Stack}(V) \times \mathrm{Stack}(V))$ induced by $\mathcal{F} \subset (\Sigma^{\mathrm{call}} \times V \times V) \cup (\Sigma^{\mathrm{ret}} \times V \times V \times V) \cup (\Sigma^{\mathrm{int}} \times V \times V)$:

- If $c \in \Sigma_k^{\mathrm{call}}$, then $\mathcal{F}[\![c]\!] := \{(v, v) : v \in V_j,\ j \neq k\} \cup \mathcal{F}_k[\![c]\!]$.
- If $r \in \Sigma_k^{\mathrm{ret}}$, then $\mathcal{F}[\![r]\!] := \{(v, v_<, v) : v, v_< \in V_j,\ j \neq k\} \cup \mathcal{F}_k[\![r]\!]$.
- If $a \in \Sigma_k^{\mathrm{int}}$, then $\mathcal{F}[\![a]\!] := \{(v, v) : v \in V_j,\ j \neq k\} \cup \mathcal{F}_k[\![a]\!]$.

$\mathcal{S}$ can be easily adjusted to account for shared memory: the shared global variables are also copied over when when the stack is pushed and popped.

Recall that $\mathcal{M}$ is the n-stack semantics. Well-nestedness guarantees that the shift from $\mathcal{M}$ to $\mathcal{S}$ preserves the validity of Hoare triples, so any run in the well-nested shuffle has no distinction from a run of a standard recursive program.

Theorem 7.1. *For any syntactic run $\rho \in P_1 \between \cdots \between P_n$ and assertions A, B, the validity of $\{A\}\rho\{B\}$ is the same with respect to the semantics $\mathcal{M}$ and $\mathcal{S}$.*

8 Experimental Results

We have implemented our reduction-based scheme as a tool, called HyRec, that verifies hypersafety properties of sequential recursive programs.

8.1 Implementation

HyRec verifies an input program written in a simple procedural programming language against a given pair of pre/postconditions. The programming language supports the integer, boolean, and array data types. The user specifies the property and one of the built-in (parametric) canonical reductions, or a combination of canonical reductions. For instance, to verify the DISTRIBUTIVITY property of `div`, one would use the command[3]:

```
HyRec div -property DISTRIBUTIVITY -reduction "(1,1)-lockstep(P3, nested_concatenation(P1,P2))"   (C)
```

HyRec also accepts an arbitrary uniform visibly pushdown order, but none was required to solve any of the benchmarks used for the evaluation.

HyRec implements the following algorithms:

[3] DISTRIBUTIVITY is a shorthand for the pre/postcondition that expresses the property.

AUT: the optimized construction of Section 4.3 as a nested word automaton (NWA) [3], a model that is equivalent to a VPA and supported by existing libraries [33];
VPG: the same optimized construction but additionally converting the NWA into a VPG;
DIRECT: the direct construction of Section 6.3.

In each case, the resulting product program (in NWA or VPG format) and the corresponding hypersafety property are encoded by HYREC as system of constrained Horn clauses (CHCs) in the SMT-LIB format [6], which is satisfiable if and only if the pre/postcondition holds. We did not find a classic recursive verification tool that does well and is not already using a similar CHC encoding in the backend. In particular, [34] cannot handle these instances well. We use Z3/Spacer [21, 40], Eldarica [36], and Golem [11] as the backend CHC solvers.

CHC Encodings. We interpret the product program under the semantics $\mathcal{S}$ in Section 7. An NWA can be directly encoded as CHCs in linear time (see Appendix D.1): one predicate for each state, and one constraint for each transition that captures the *inductivity* [35] of the predicates (assertions). In [35], the assertions annotate a nested trace, whereas in our setting, the assertions annotate the NWA, and the naive encoding would be unsound with respect to unsat since it does not distinguish context-sensitive and insensitive paths. Our solution is to introduce a ghost variable that represents the "return address", so only assertions along context-sensitive paths are required to be inductive.

In the conversion of the NWA to a VPG, we use the standard quadratic time [3] algorithm and then eliminate the useless symbols and productions [37]. A VPG is encoded as CHCs in linear time (see Appendix D.1): one predicate for each nonterminal that *overapproximates* the semantics of the language generating by this nonterminal. The semantics $\mathcal{S}$ involves a stack, but for a well-matched language, if we are only concerned with partial correctness, then we may use a *stack-free* semantics defined inductively over the nesting structure, relating the variables before and after the execution of a trace. VPGs have this nesting structure built-in, and each production is translated into one constraint. This encoding is similar to "transition summaries" from [10].

8.2 Benchmarks

We built a benchmark set by including the safe relational benchmarks from [48] (28 in total, which are in turn adapted from [20, 47]), taking the iterative hypersafety (non-concurrent) benchmarks from [27] (17 in total) and rewriting them as recursive code, and adding 58 new ones. In some cases, the new additional instances are in the form of additional hypersafety properties of benchmarks from [27, 48]. Our newly added benchmarks consist of two groups, one for arithmetic functions (33), and one for functions on arrays (25). In total, we have 62 arithmetic benchmarks, and 41 array benchmarks. (See Appendix E for details.)

The arithmetic benchmarks are hypersafety properties of primitives, in the spirit of the `div` example from Section 1. They include programs with linear or non-linear recursion, and then several hypersafety properties per program. Recall that the MONOTONICITY property of `div` is relatively straightforward, but the DISTRIBUTIVITY property of it is rather challenging to solve. So, the hardness of an instance can depend on the benchmark or the property or both. The array benchmarks are hypersafety properties of, or relations between, aggregate functions (min, max, and sum) and comparators (pointwise comparison and lexicographic order) on arrays; we have also included benchmarks that manipulate trees that are encoded as heaps in arrays; this is due to restrictions in the CHC solvers' support for theories and has nothing to do with our methodology.

Despite the fact that hyperproperties were originally motivated by their applications in security, in terms of diversity and quantity, there are many more instances of *primitives* (like division or lexicographical orders on vectors) implementing known operations where k-safety properties are

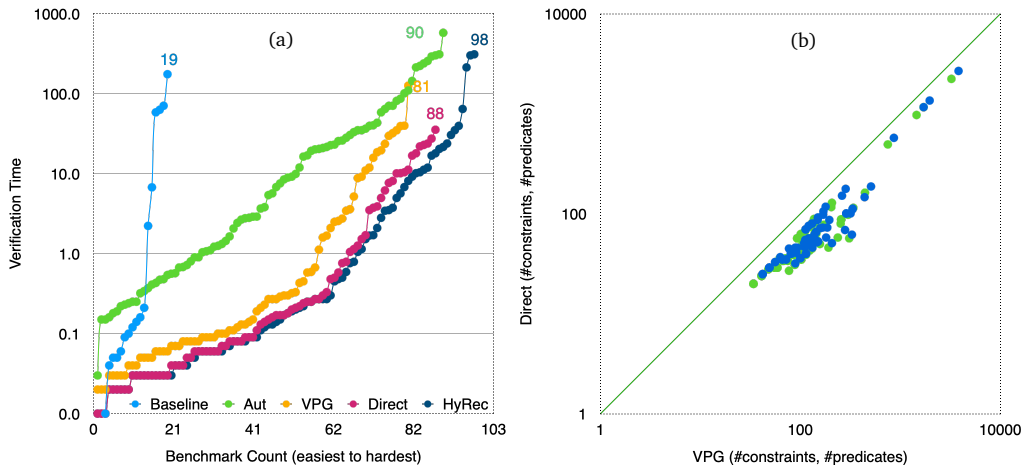


Fig. 7. Experimental Results: logscale plots. (a): quantile plot of performance comparison HyRec's algorithms against the baseline. (b): scatter plot of comparing CHC encoding sizes for the two grammar constructions.

essential to encode and check the basic properties of these primitives. The arities of the hypersafety instances for these can vary. All but four benchmarks are 2- or 3-safety properties, and the rest has $k = 7$ at maximum. These properties are specified in the style of (C).

8.3 Experiments

Our experiments were designed to answer the following research questions:

- RQ1: Is our reduction approach essential and effective in proving hypersafety properties of recursive programs?
- RQ2: Are parameterized lockstep and nested concatenation, as building blocks, expressive enough to support the construction of effective product programs for a broad range of examples?
- RQ3: Does the representation of reduction (automata v.s. two grammar styles) have an impact on the effectiveness of the process?

With respect to concurrency, this paper mainly proposes the theoretical insight that our reduction methodology acts as a complete sequentialization scheme for certain classes of recursive programs. We do not view HyRec as a tool for verification or bug finding of recursive concurrent programs.

Results. As a **baseline** comparison, we use three standard scenarios, with the *VPG encoding* in Appendix D.1: (1) sequential composition as the product program, (2) a direct encoding of the problem without reductions and without making k disjoint copies for the k-safety property, and (3) the variation of (2) in which k disjoint copies are made. We refer to Baseline as the best of these three algorithms, including the best performing CHC solver. We tried RelRecMc (the tool from [48]) on all baseline encodings, but it times out or throws a `Segmentation fault` on every instance. We also tested the `div` examples from Section 1 with its own style of encoding, and scaling and distributivity cannot be solved.

On average, the production of the CHC encoding takes very little time (around 1 second) for all instances. As expected, the direct construction method is the fastest, followed by the generation of the automaton and the VPG based on the automaton. Our reported runtimes, therefore, are only for the time to verify the product program, and excludes this negligible construction time. We use a timeout of 600 seconds for the combination of both steps.

Figure 7(a) illustrates the *log-scale* quantile plot that compares our three algorithms (Aut, VPG, and Direct) against the Baseline. HyRec is the best of the three algorithms. The arithmetic benchmarks solved by Baseline are exactly those with a function summary in Linear Integer Arithmetic. (RQ1) Direct, VPG, and Aut contribute 73, 17, and 13 instances to HyRec respectively (accounting for the ties). The number for each curve lists the total number of benchmarks solved by that algorithm. It is noteworthy that CHC encodings based on grammars are on average much more efficiently solvable than the one based on automata; yet, Aut is better at solving harder instances: out of the 13 winners contributed by Aut, 9 of them are only solvable by Aut, whereas all 17 benchmarks where VPG wins can be solved by others albeit more slowly. (RQ3)

Our direct construction does offer a significant improvement. This is while the resulting product programs are identical when it comes to the set of their syntactic runs. The difference may stem from the compactness of the resulting grammar. Figure 7(b) compares the sizes of the resulting CHC encodings: blue dots compare the number constraints and green dots that of predicates. (RQ3)

HyRec fails to solve 5 of the 103 benchmarks. One instance is a shortcoming of the CHC solvers and is not related to the product program aspect. It is the simplest of a group of 4 benchmarks from [27] that increasingly unroll loops. HyRec successfully proves unrolling the loop by 3, 4, and 5 times, but fails at the 2 unrolling. The rest of the failures are worthy of a deeper discussion, and are explored in the next section. (RQ2)

Separately, we also evaluate the scalability of our approach for higher arity hypersafety instances. With increasing k until the first timeout or error, we run the tool with the `div` program (Fig. 1), $(1, \ldots, 1)$-lockstep reduction, and the variation of (monotonicity) as a k-property:

$$n_1 \leq \cdots \leq n_k \Rightarrow \texttt{div}(n_k, d) \leq \cdots \leq \texttt{div}(n_k, d).$$

As the arity of the hypersafety instance goes higher, the time for generating the CHC encoding is increasing. The three algorithms Direct, Aut, and VPG scale to $k = 12$, 7, and 6 respectively; for the largest instance (Direct, $k = 12$, and 10^5 predicates in the CHC encoding), CHC generation takes up 33% of the total time. In practice, however, these arities for hypersafety are very rare. As expected, the size of the CHC encodings (measured by the number of predicates or constraints) grows multiplicatively for all algorithms.

8.4 Discussion and Limitations

Here, we summarize some key observations from the experiments and discuss some unique features and limitations of our proposed solution. Most importantly, we discuss the reasons behind why 4 benchmarks from the set cannot be solved by HyRec.

Representation matters. It is somewhat surprising that although the same exact set of alignments are represented by the two different grammar constructions and the automata construction, the results significantly differ in some cases. The difference between the grammar constructions can be attributed to the size, as the scattered plot in Figure 7(b) highlights. This advantage, of having 3 chances at solving an instance, is a strict consequence of our *operational* framework.

Alignments do not always work. It is undeniable that reductions are essential to the success of these hypersafety verification instances. It is also important to note that not all hypersafety verification instances can be solved using these reductions. For instance, we use a primitive $\texttt{mult}(a, b)$ in our benchmark set that multiplies a by b through repetitive addition. Consider the following simple theorem about the commutativity of multiplication: $\texttt{mult}(n, m) = \texttt{mult}(m, n)$. This is a 2-safety property, where none of the lex reductions in this paper can simplify the proof down to a decidable theory (linear integer arithmetic). The same is true for the iterative version of this code as well. Two benchmarks from the set cannot be solved for this reason.

Explicit Product Programs v.s. Integrated Verification Technique. There is a benchmark from [48] related to Fibonacci that HyRec cannot solve, which stems from the fact that the conceptual product program for this must reorder the two recursive calls to succeed. In code (and standard semantics), this reordering is forbidden, but at the level of CHC encoding, once one is looking at a symbolic encoding, it can be done semantically and soundly. One can weaken program semantics to permit such sound reorderings. However, when the manipulations are done at the level of the symbolic encoding, it comes for free. As discussed in Section 1, having an explicit product program can have other advantages: one can get testing, symbolic execution, program analysis, and other off-the-shelf applications for free.

Syntactic v.s. Semantic alignments. Our reduction are syntactic, and HyRec fails at solving the *sum of squares* benchmark [52] (modeled by an array in our benchmark set), which truly requires a semantic alignment in the sense that one needs access to the program data to determine the right synchronization; the tool from [38], for iterative programs, can handle the iterative version of this benchmark automatically. We believe semantic alignments can be incorporated in our framework, but this requires a nontrivial shift in representation, for instance to *symbolic automata* [17, 18], and this is left for future work. It is noteworthy, however, that in some instances in the literature, *semantic* alignments are used to account for limitations in *syntactic* alignments that are superficial. For instance, our nested concatenation is an instance of a syntactic benchmark that may seem superficially like a semantic benchmark: one needs to wait for one recursive call to be finished before starting the second one (which is meant to be nested inside it). However, we handle this in a purely syntactic way, and lex reductions in general include reductions in which the two copies can go out of synch for arbitrarily long durations, without the need to rely on program data to make this happen.

Programmer's intuition goes a long way. We found it very easy and straightforward to guess and specify the right choice of reduction for each benchmark. Considering that 45% of our benchmarks were not designed by us, but acquired from other sources (including those that search for reductions), this is remarkable.

9 Related Work

The most closely related work to ours is a line of work that identifies the gap in *relational verification* with the aid of CHC solving [4, 10, 20, 24, 47, 48] where the models are expressive enough to encode arbitrary recursion schemes like ours. While we operate at the recursive program level, these techniques directly target the CHC encodings [10] of these programs and employ transformations such as fold/unfold [4, 20]. In [24], asynchronous fold/unfold transformation is proposed for fixpoint logic; the transformed logic formula is encoded and passed to a CHC solver. Our product programs go beyond a bounded number of unfoldings. Recall the DISTRIBUTIVITY property of `div`. In the spirit of fold/unfold, one may think that unfolding once each of the $\texttt{div}(n, d)$ and $\texttt{div}(n', d)$ instances and twice the $\texttt{div}(n'', d)$ instance may yield a product program, but this would not work due to a problem with the base case. Our product program conceptually unfolds $\texttt{div}(n + n', d)$ for n' times, which is essential in handling cases like this.

CHCProduct [47] is a family of transformations that resemble synchronous lockstep. The net effect is not theoretically equivalent to our Figure 4 with $\boldsymbol{s} = (1, \ldots, 1)$, since CHCProduct operates on the symbolic encoding of the programs, which allows the function calls to be "rearranged" and "duplicated". This flexibility also means the CHCProduct is not uniquely defined, and the extension of *synchronous* CHCProduct [47] employs a heuristic to select a unique product. In [48], an alternative technique based on Property Directed Reachability is proposed, which maintains over-

and under-approximations of groups of predicates and automatically infers synchronous lockstep strategies, also on the symbolic encoding of the programs.

Putting CHC aside, in [22] for programs with procedures, *modular product programs* are constructed via a source-to-source transformation. Such product programs are intuitively a $(1, \ldots, 1)$-lockstep reduction of copies of the same program. In this special case, modular product programs introduce Boolean *activation variables* to encode control-flow symbolically, thereby avoiding taking the product of control locations.

Multi-Stack Models and Sequentialization. The languages of runs of a multi-stack pushdown automata are equivalent to those of Turing machines, even with two stacks. There are a number of solutions in the literature that deal with this problem by under-approximating these multi-stack languages. The most well-known ones are based on bounding a measure in the set of runs; these include bounding the number of context switches [49, 55], phase bounding [41, 55], and scope bounding [42–44]. Our reductions are not an under-approximation, but rather a faithful representation. The languages are also theoretically incomparable against all these models. For example, in a bounded context switching scheme, one is permitted to break the respective order of calls and returns a bounded number of times, whereas it cannot be done in our model. In contrast, we include runs that are not bounded in up to any of the measures: number of context switches, phases, or scopes. Our focus in this paper is not on developing new classes of languages or giving new decidability results. There is a loose connection between reductions and notion of *sequentialization* [45] in the literature based on the same ideas of bounding the number of context switches, phases, or scopes. The well-nested shuffle can be considered as sequentialization that is *complete* under the conditions given in Section 5. It would be interesting to explore in future how well *incomplete* reductions of concurrent programs perform as means of under-approximations for bug finding.

Hyperproperty Verification. There is a large body of work studying verification of hyperproperties, where earlier work mostly focused on verification of hypersafety properties [5, 7, 8, 22, 27, 31, 38, 52–54, 56, 57]. Since our focus is on verification of infinite-state hypersafety properties, we forgo surveying the verification of hyperliveness properties for finite-state programs, unless there is an overlap in techniques that makes the comparison worthwhile. There is a cluster of work [1, 12, 32] on verification of hyperproperties [15] based on HyperLTL (a temporal logic for hyperproperties) [15]. The work [32] on the verification of finite-state recursive programs, in the context of asynchronous HyperLTL, is the only one from the cluster that addresses recursion in the hyperproperty space. The focus of [32] is on the decidability results of the relaxations of the logic for finite state recursive programs. We are in the scope of infinite-state programs, where verification is undecidable even in the absence of recursion. The state space of alignments (i.e., the degree of asynchrony) in [32] is not as general as an arbitrary reduction we define in Section 3.

Alignments for Iterative Programs. Putting recursion aside, the closest to our work in terms of aim and scope are those that focus on verification of infinite-state programs and take advantages of the state space of alignments to solve the verification task. Specifically, in [27, 52, 53], automated hypersafety verification techniques for sequential and concurrent programs [27] are proposed. In [56], a generalization of CHCs with a specialized solving algorithm is introduced, which is able to encode hyperproperties beyond hypersafety. In [9], an automated game-based verification technique is used for a subset of hyperproperties that includes hypersafety properties but go beyond to include some classes of hyperliveness. Finally, in [38], a solution based on CHC encoding of a similar class of hyperproperties is proposed, which searches for a semantic alignment and a proof simultaneously. The tool from [38] can handle some infinite-state programs, but for others, it requires the right *abstraction*, given by the user, that would translate the program into a finite-state

one; consequently, the search space for alignments also become finite. As an instance, none of the three properties of the iterative version of the `div` example (from Section 1) can be verified by the tool over the concrete state. In some sense, recursive programs inherit all the complications of the iterative framework and add an extra layer of complexity. We discuss this extra complexity in Section 1 and also note that the existence of a stack means even a program with finite data domain may have infinite state space.

In the paper, we mostly target the additional layer of complexity, and do not offer a new insight for iterative programs, with one exception: *canonical reductions as building blocks* for the user to communicate reduction ideas to a verification tool. More precisely, the analogous set of lex reductions for iterative programs were already introduced in [26], but the analog of canonical reductions (as a simple and effective way to specify a reduction) does not exist in the iterative program literature, to the best of our knowledge. It is noteworthy that the iterative counterparts of the canonical reductions are far easier to define formally and implement. Therefore, they may offer a practical solution to program alignments that sits between one fixed reduction (which has limited expressivity) and automated alignment search (which has limited scalability [27] or requires user-provided abstractions in practice [38]).

Data-Availability Statement

Our tool HyRec and the benchmarks used in the evaluation (Section 8) can be found at [13]. A description of all benchmarks is provided in Appendix E.

References

[1] Shreya Agrawal and Borzoo Bonakdarpour. 2016. Runtime Verification of K-Safety Hyperproperties in HyperLTL. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*. 239–252. doi:10.1109/CSF.2016.24

[2] Rajeev Alur and P. Madhusudan. 2004. Visibly Pushdown Languages. In *Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing*. ACM, Chicago IL USA, 202–211. doi:10.1145/1007352.1007390

[3] Rajeev Alur and P. Madhusudan. 2009. Adding Nesting Structure to Words. *J. ACM* 56, 3 (May 2009), 16:1–16:43. doi:10.1145/1516512.1516518

[4] Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. 2018. Predicate Pairing for Program Verification. *Theory and Practice of Logic Programming* 18, 2 (March 2018), 126–166. doi:10.1017/S1471068417000497

[5] Timos Antonopoulos, Eric Koskinen, Ton Chanh Le, Ramana Nagasamudram, David A. Naumann, and Minh Ngo. 2023. An Algebra of Alignment for Relational Verification. *Proc. ACM Program. Lang.* 7, POPL (Jan. 2023), 20:573–20:603. doi:10.1145/3571213

[6] Clark Barrett, Aaron Stump, and Cesare Tinelli. 2010. The SMT-LIB Standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability modulo Theories (Edinburgh, UK)*, A. Gupta and D. Kroening (Eds.).

[7] Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2011. Relational Verification Using Product Programs. In *FM 2011: Formal Methods*, Michael Butler and Wolfram Schulte (Eds.). Springer, Berlin, Heidelberg, 200–214. doi:10.1007/978-3-642-21437-0_17

[8] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. 2004. Secure Information Flow by Self-Composition. In *Proceedings. 17th IEEE Computer Security Foundations Workshop*. IEEE Computer Society, 100–100. doi:10.1109/CSFW.2004.1310735

[9] Raven Beutner and Bernd Finkbeiner. 2022. Software Verification of Hyperproperties Beyond K-Safety. In *Computer Aided Verification*, Sharon Shoham and Yakir Vizel (Eds.). Vol. 13371. Springer International Publishing, Cham, 341–362. doi:10.1007/978-3-031-13185-1_17

[10] Nikolaj Bjørner, Arie Gurfinkel, Ken McMillan, and Andrey Rybalchenko. 2015. Horn Clause Solvers for Program Verification. In *Fields of Logic and Computation II*, Lev D. Beklemishev, Andreas Blass, Nachum Dershowitz, Bernd Finkbeiner, and Wolfram Schulte (Eds.). Vol. 9300. Springer International Publishing, Cham, 24–51. doi:10.1007/978-3-319-23534-9_2

[11] Martin Blicha, Konstantin Britikov, and Natasha Sharygina. 2023. The Golem Horn Solver. In *Computer Aided Verification*, Constantin Enea and Akash Lal (Eds.). Springer Nature Switzerland, Cham, 209–223. doi:10.1007/978-3-031-37703-7_10

[12] Laura Bozzelli, Adriano Peron, and César Sánchez. 2021. Asynchronous Extensions of HyperLTL. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 1–13. doi:10.1109/LICS52264.2021.9470583

[13] Ruotong Cheng and Azadeh Farzan. 2025. HyRec: Artifact for "Products of Recursive Programs for Hypersafety Verification". Zenodo. doi:10.5281/zenodo.16930073

[14] Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. 2019. Semantic Program Alignment for Equivalence Checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 1027–1040. doi:10.1145/3314221.3314596

[15] Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. 2014. Temporal Logics for Hyperproperties. In *Principles of Security and Trust*, Martín Abadi and Steve Kremer (Eds.). Springer, Berlin, Heidelberg, 265–284. doi:10.1007/978-3-642-54792-8_15

[16] Michael R. Clarkson and Fred B. Schneider. 2010. Hyperproperties. *JCS* 18, 6 (Sept. 2010), 1157–1210. doi:10.3233/JCS-2009-0393

[17] Loris D'Antoni and Rajeev Alur. 2014. Symbolic Visibly Pushdown Automata. In *Computer Aided Verification*, Armin Biere and Roderick Bloem (Eds.). Springer International Publishing, Cham, 209–225. doi:10.1007/978-3-319-08867-9_14

[18] Loris D'Antoni and Margus Veanes. 2017. The Power of Symbolic Automata and Transducers. In *Computer Aided Verification*, Rupak Majumdar and Viktor Kunčak (Eds.). Springer International Publishing, Cham, 47–67. doi:10.1007/978-3-319-63387-9_3

[19] Martin Davis, Ron Sigal, and Elaine J. Weyuker. 1994. *Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science* (2nd. ed ed.). Academic Press, Harcourt, Brace, Boston.

[20] Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. 2016. Relational Verification Through Horn Clause Transformation. In *Static Analysis*, Xavier Rival (Ed.). Springer, Berlin, Heidelberg, 147–169. doi:10.1007/978-3-662-53413-7_8

[21] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, Berlin, Heidelberg, 337–340. doi:10.1007/978-3-540-78800-3_24

[22] Marco Eilers, Peter Müller, and Samuel Hitz. 2018. Modular Product Programs. In *Programming Languages and Systems*, Amal Ahmed (Ed.). Springer International Publishing, Cham, 502–529. doi:10.1007/978-3-319-89884-1_18

[23] Javier Esparza, David Hansel, Peter Rossmanith, and Stefan Schwoon. 2000. Efficient Algorithms for Model Checking Pushdown Systems. In *Computer Aided Verification*, E. Allen Emerson and Aravinda Prasad Sistla (Eds.). Springer, Berlin, Heidelberg, 232–247. doi:10.1007/10722167_20

[24] Mahmudul Faisal Al Ameen, Naoki Kobayashi, and Ryosuke Sato. 2024. Asynchronous Unfold/Fold Transformation for Fixpoint Logic. *Science of Computer Programming* 231 (Jan. 2024), 103014. doi:10.1016/j.scico.2023.103014

[25] Azadeh Farzan. 2023. Commutativity in Automated Verification. In *2023 38th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. 1–7. doi:10.1109/LICS56636.2023.10175734

[26] Azadeh Farzan, Dominik Klumpp, and Andreas Podelski. 2022. Sound Sequentialization for Concurrent Program Verification. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 506–521. doi:10.1145/3519939.3523727

[27] Azadeh Farzan and Anthony Vandikas. 2019. Automated Hypersafety Verification. In *Computer Aided Verification*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, Cham, 200–218. doi:10.1007/978-3-030-25540-4_11

[28] Azadeh Farzan and Anthony Vandikas. 2020. Reductions for Safety Proofs. *Proc. ACM Program. Lang.* 4, POPL (Jan. 2020), 1–28. doi:10.1145/3371081

[29] Patrice Godefroid, Gerhard Goos, Juris Hartmanis, and Jan Leeuwen (Eds.). 1996. *Partial-Order Methods for the Verification of Concurrent Systems*. Lecture Notes in Computer Science, Vol. 1032. Springer, Berlin, Heidelberg. doi:10.1007/3-540-60761-7

[30] Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and Sai Deep Tetali. 2010. Compositional May-Must Program Analysis: Unleashing the Power of Alternation. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '10)*. Association for Computing Machinery, New York, NY, USA, 43–56. doi:10.1145/1706299.1706307

[31] Benny Godlin and Ofer Strichman. 2013. Regression Verification: Proving the Equivalence of Similar Programs. *Software Testing Verif & Rel* 23, 3 (May 2013), 241–258. doi:10.1002/stvr.1472

[32] Jens Oliver Gutsfeld, Markus Müller-Olm, and Christoph Ohrem. 2024. Deciding Asynchronous Hyperproperties for Recursive Programs. *Proc. ACM Program. Lang.* 8, POPL (Jan. 2024), 2:33–2:60. doi:10.1145/3632844

[33] Matthias Heizmann, Jürgen Christ, Daniel Dietsch, Evren Ermis, Jochen Hoenicke, Markus Lindenmann, Alexander Nutz, Christian Schilling, and Andreas Podelski. 2013. Ultimate Automizer with SMTInterpol. In *Tools and Algorithms for the Construction and Analysis of Systems*, Nir Piterman and Scott A. Smolka (Eds.). Springer, Berlin, Heidelberg, 641–643. doi:10.1007/978-3-642-36742-7_53

[34] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. 2009. Refinement of Trace Abstraction. In *Static Analysis*, Jens Palsberg and Zhendong Su (Eds.). Vol. 5673. Springer Berlin Heidelberg, Berlin, Heidelberg, 69–85.

doi:10.1007/978-3-642-03237-0_7

[35] Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. 2010. Nested Interpolants. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '10)*. Association for Computing Machinery, New York, NY, USA, 471–482. doi:10.1145/1706299.1706353

[36] Hossein Hojjat and Philipp Rümmer. 2018. The ELDARICA Horn Solver. In *2018 Formal Methods in Computer Aided Design (FMCAD)*. 1–7. doi:10.23919/FMCAD.2018.8603013

[37] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2007. *Introduction to Automata Theory, Languages, and Computation* (3rd ed ed.). Pearson/Addison Wesley, Boston.

[38] Shachar Itzhaky, Sharon Shoham, and Yakir Vizel. 2024. Hyperproperty Verification as CHC Satisfiability. In *Programming Languages and Systems*, Stephanie Weirich (Ed.). Springer Nature Switzerland, Cham, 212–241. doi:10.1007/978-3-031-57267-8_9

[39] Xiaodong Jia, Ashish Kumar, and Gang Tan. 2021. A Derivative-Based Parser Generator for Visibly Pushdown Grammars. *Proc. ACM Program. Lang.* 5, OOPSLA (Oct. 2021), 1–24. doi:10.1145/3485528

[40] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. 2016. SMT-based Model Checking for Recursive Programs. *Form Methods Syst Des* 48, 3 (June 2016), 175–205. doi:10.1007/s10703-016-0249-4

[41] Salvatore La Torre, P. Madhusudan, and Gennaro Parlato. 2008. An Infinite Automaton Characterization of Double Exponential Time. In *Computer Science Logic*, Michael Kaminski and Simone Martini (Eds.). Springer, Berlin, Heidelberg, 33–48. doi:10.1007/978-3-540-87531-4_5

[42] Salvatore La Torre and Margherita Napoli. 2011. Reachability of Multistack Pushdown Systems with Scope-Bounded Matching Relations. In *CONCUR 2011 – Concurrency Theory*, Joost-Pieter Katoen and Barbara König (Eds.). Springer, Berlin, Heidelberg, 203–218. doi:10.1007/978-3-642-23217-6_14

[43] Salvatore La Torre, Margherita Napoli, and Gennaro Parlato. 2014. Scope-Bounded Pushdown Languages. In *Developments in Language Theory*, Arseny M. Shur and Mikhail V. Volkov (Eds.). Springer International Publishing, Cham, 116–128. doi:10.1007/978-3-319-09698-8_11

[44] Salvatore La Torre, Margherita Napoli, and Gennaro Parlato. 2020. Reachability of Scope-Bounded Multistack Pushdown Systems. *Information and Computation* 275 (Dec. 2020), 104588. doi:10.1016/j.ic.2020.104588

[45] Akash Lal and Thomas Reps. 2008. Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis. In *Computer Aided Verification*, Aarti Gupta and Sharad Malik (Eds.). Springer, Berlin, Heidelberg, 37–51. doi:10.1007/978-3-540-70545-1_7

[46] A Mazurkiewicz. 1987. Trace Theory. In *Advances in Petri Nets 1986, Part II on Petri Nets: Applications and Relationships to Other Models of Concurrency*. Springer-Verlag, Berlin, Heidelberg, 279–324.

[47] Dmitry Mordvinov and Grigory Fedyukovich. 2017. Synchronizing Constrained Horn Clauses. In *LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning*. 338–319. doi:10.29007/gr5c

[48] Dmitry Mordvinov and Grigory Fedyukovich. 2019. Property Directed Inference of Relational Invariants. In *2019 Formal Methods in Computer Aided Design (FMCAD)*. 152–160. doi:10.23919/FMCAD.2019.8894274

[49] Shaz Qadeer and Jakob Rehof. 2005. Context-Bounded Model Checking of Concurrent Software. In *Tools and Algorithms for the Construction and Analysis of Systems*, Nicolas Halbwachs and Lenore D. Zuck (Eds.). Springer, Berlin, Heidelberg, 93–107. doi:10.1007/978-3-540-31980-1_7

[50] G. Ramalingam. 2000. Context-Sensitive Synchronization-Sensitive Analysis Is Undecidable. *ACM Trans. Program. Lang. Syst.* 22, 2 (March 2000), 416–430. doi:10.1145/349214.349241

[51] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95)*. Association for Computing Machinery, New York, NY, USA, 49–61. doi:10.1145/199448.199462

[52] Ron Shemer, Arie Gurfinkel, Sharon Shoham, and Yakir Vizel. 2019. Property Directed Self Composition. In *Computer Aided Verification*, Isil Dillig and Serdar Tasiran (Eds.). Vol. 11561. Springer International Publishing, Cham, 161–179. doi:10.1007/978-3-030-25540-4_9

[53] Marcelo Sousa and Isil Dillig. 2016. Cartesian Hoare Logic for Verifying K-Safety Properties. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, Santa Barbara CA USA, 57–69. doi:10.1145/2908080.2908092

[54] Tachio Terauchi and Alex Aiken. 2005. Secure Information Flow as a Safety Problem. In *Static Analysis*, David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, Chris Hankin, and Igor Siveroni (Eds.). Vol. 3672. Springer Berlin Heidelberg, Berlin, Heidelberg, 352–367. doi:10.1007/11547662_24

[55] Salvatore La Torre, Parthasarathy Madhusudan, and Gennaro Parlato. 2007. A Robust Class of Context-Sensitive Languages. In *22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*. 161–170. doi:10.1109/LICS.2007.9

[56] Hiroshi Unno, Tachio Terauchi, and Eric Koskinen. 2021. Constraint-Based Relational Verification. In *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I*. Springer-Verlag, Berlin, Heidelberg, 742–766. doi:10.1007/978-3-030-81685-8_35

[57] Weikun Yang, Yakir Vizel, Pramod Subramanyan, Aarti Gupta, and Sharad Malik. 2018. Lazy Self-composition for Security Verification. In *Computer Aided Verification*, Hana Chockler and Georg Weissenbacher (Eds.). Vol. 10982. Springer International Publishing, Cham, 136–156. doi:10.1007/978-3-319-96142-2_11

[58] Anna Zaks and Amir Pnueli. 2008. CoVaC: Compiler Validation by Program Analysis of the Cross-Product. In *FM 2008: Formal Methods*, Jorge Cuellar, Tom Maibaum, and Kaisa Sere (Eds.). Springer, Berlin, Heidelberg, 35–51. doi:10.1007/978-3-540-68237-0_5

A Background

A.1 Recursive Programs as VPLs

Following [35], we model a (single-stack) recursive program as a VPL (a well-matched one, for our purpose), where the alphabet $\widetilde{\Sigma}$ is the program statements. Such VPLs can be obtained from a recursive control-flow graph by viewing the graph as a VPA. Not all words in the language is a semantically valid run of the program: a sequence of statements (also called a *trace*) that repsects the control-flow structure and correctly matches the calls and returns may be *infeasible* when semantics is taken into account. Below, we define a concrete alphabet $\widetilde{\Sigma}$ and its semantics. We use it in the examples in Appendix B and in our implementation of HyRec. The theory developed in Section 3–7 does not rely on this concrete alphabet.

For any set X, let $\mathrm{Stack}(X)$ denote the set of stacks whose frames are elements of X, and let $\mathcal{P}(X)$ denote the power set of X. Let L be the set of local variables, F the set of function names, $L_\mathrm{p} \subset L$ the set of function parameters, $\mathrm{param} : F \to \mathcal{P}(L_\mathrm{p})$ that maps each function name to a set of parameters, and $\mathrm{output} : F \to \mathcal{P}(L)$ the maps each function name to a set of output variables. Define $\bar{L}_\mathrm{p} := \{\bar{p} \mid p \in L_\mathrm{p}\}$, which represents dummy variables that record the initial value of the parameters when a function is called. Let D be the domain of the variables and $T = D^{L \cup \bar{L}_\mathrm{p}}$ the set of all valuations.

We assume that $\widetilde{\Sigma}$ consists of *assign* statements $\boxed{x := t}$, *assume* statements $\boxed{\text{assume } \phi}$, *call* statements $\boxed{\text{call } f}$, and *return* statements $\boxed{\text{ret } f}$, where $x \in L$, t is an expression over L, ϕ is an assertion over L, and $f \in F$. The first two types of statements are internal. For a program P over this alphabet $\widetilde{\Sigma}$, we always assume that for any word in P, if a call $\boxed{\text{call } f}$ and a return $\boxed{\text{ret } f}$ match, then $f = g$. Define the semantics function $\mathcal{S} : \widetilde{\Sigma} \to \mathcal{P}(\mathrm{Stack}(T) \times \mathrm{Stack}(T))$ as follows:

$$
\begin{aligned}
\mathcal{S}[\![x := t]\!] &:= \{(S.v,\ S.v') : v' = v \oplus \{x \mapsto v(t)\}\},\\
\mathcal{S}[\![\phi]\!] &:= \{(S.v,\ S.v) : v \models \phi\},\\
\mathcal{S}[\![\texttt{call } f]\!] &:= \{(S.v,\ S.v.v') : \forall p \in \mathrm{param}(f).\ v'(p) = v'(\bar{p}) = v(p)\},\\
\mathcal{S}[\![\texttt{ret } f]\!] &:= \{(S.v_<.v,\ S.v') : v' = v_< \oplus \{y \mapsto v(y) : y \in \mathrm{output}(f)\}\}.
\end{aligned}
$$

Extend $\mathcal{S}$ from letters to words via relation composition. For a program P over $\widetilde{\Sigma}$, a run $\rho \in P$ is said to be infeasible if $\mathcal{S}[\![\rho]\!] = \varnothing$. When talking about the runs of a program, we by default include both the feasible ones and infeasible ones.

B Examples of Product Programs

The formal semantics of the alphabet that we use is presented in Appendix A.1.

B.1 `div`

The grammar for `div` is as follows:

$$
\begin{aligned}
D &\to \boxed{\text{assume } n < d}\ \boxed{q := 0}\\
D &\to \boxed{\text{assume } n \geq d}\ \boxed{\texttt{call div}}\ D\ \boxed{\texttt{ret div}}\ \boxed{q := q + 1}
\end{aligned}
$$

The grammar for synchronously lockstepping two copies of $\boxed{\texttt{call div}}\ D\ \boxed{\texttt{ret div}}$ is as follows, where $\boxed{\sigma : k}$ means performing σ on the k-th stack and S is the start symbol:

$$
\begin{aligned}
S &\to \boxed{\texttt{call div} : 1}\ A\ \boxed{\texttt{ret div} : 1}\\
A &\to \boxed{\text{assume } n < d : 1}\ D_2\\
A &\to \boxed{\text{assume } n \geq d : 1}\ \boxed{\texttt{call div} : 2}\ B\ \boxed{\texttt{ret div} : 2}\ \boxed{q := q + 1 : 1}\\
B &\to \boxed{\text{assume } n < d : 2}\ D_1\\
B &\to \boxed{\text{assume } n \geq d : 2}\ \boxed{\texttt{call div} : 1}\ A\ \boxed{\texttt{ret div} : 1}\ \boxed{q := q + 1 : 2}\\
D_1 &\to \boxed{\text{assume } n < d : 1}\ \boxed{q := 0 : 1}\\
D_1 &\to \boxed{\text{assume } n \geq d : 1}\ \boxed{\texttt{call div} : 1}\ D_1\ \boxed{\texttt{ret div} : 1}\ \boxed{q := q + 1 : 1}\\
D_2 &\to \boxed{\text{assume } n < d : 2}\ \boxed{q := 0 : 2}\\
D_2 &\to \boxed{\text{assume } n \geq d : 2}\ \boxed{\texttt{call div} : 2}\ D_2\ \boxed{\texttt{ret div} : 2}\ \boxed{q := q + 1 : 2}
\end{aligned}
$$

This does not exactly match the lockstep that we formally define when it comes to the ordering of internals but has the same spirits.

C Proofs

Proof of Proposition 3.2. Let $\Sigma = \{a, b\}$, $P_1 = aa^*$, and $P_2 = bb^*$. Each minimal reduction corresponds to a choice function on $(P_1 \parallel P_2)/\equiv_{\mathbb{I}}$. Note that $(P_1 \parallel P_2)/\equiv_{\mathbb{I}}$ is an infinite set, and each element in the quotient has size at least 2. □

Proof of Proposition 3.4. Let $\preceq$ be a CLO induced by a contextual order $\prec$.

To show transitivity, suppose $\alpha \preceq \beta$ and $\beta \preceq \gamma$. Suppose $\alpha = xay$, $\beta = xbz = x'b'z'$, and $\gamma = x'ct$, where $a \prec_x b$ and $b' \prec_{x'} c$.

- If $x = x'$, then $\gamma = xct$ and $a \prec_x c$, so $\alpha \preceq \gamma$.
- If x is a proper prefix of x', then xb is a prefix of γ and $a \prec_x b$, so $\alpha \preceq \gamma$.
- If x' is a proper prefix of x, then $x'b'$ is a prefix of α and $b' \prec_{x'} c'$, so $\alpha \preceq \gamma$.

The three other cases can be verified in the same way, and transitivity holds. Similarly, we can verify reflxivity and antisymmetry.

For the sufficient condition for totality, for any distinct two distinct words, consider the first position where they differ. □

Proof of Theorem 4.2. Let $A_i = (Q_i, Q_i^{\mathsf{in}}, \Gamma_i, \delta_i, Q_i^{\mathsf{F}})$ be a VPA over the visibly pushdown alphabet $\widetilde{\Sigma}_i$. Define the VPA $A = (Q, Q^{\mathsf{in}}, \Gamma, \delta, Q^{\mathsf{F}})$ over $\widetilde{\Sigma}$ as follows:

- $Q = Q_1 \times \cdots \times Q_n$.
- $Q^{\mathsf{in}} = Q_1^{\mathsf{in}} \times \cdots \times Q_n^{\mathsf{in}}$.
- $\Gamma = (\Gamma_1 \setminus \{\bot\}) \uplus \cdots \uplus (\Gamma_2 \setminus \{\bot\}) \uplus \{\bot\}$.
- $\delta^{\mathsf{int}} = \{(\boldsymbol{q}, \sigma, \boldsymbol{q}[i \mapsto q_i']) : \boldsymbol{q} \in Q,\ i \in \{1, \ldots, n\},\ (q_i, \sigma, q_i') \in \delta_i\}$.
- $\delta^{\mathsf{call}} = \{(\boldsymbol{q}, \sigma, \boldsymbol{q}[i \mapsto q_i'], \gamma) : \boldsymbol{q} \in Q,\ i \in \{1, \ldots, n\},\ (q_i, \sigma, q_i', \gamma) \in \delta_i\}$.
- $\delta^{\mathsf{ret}} = \{(\boldsymbol{q}, \sigma, \gamma, \boldsymbol{q}[i \mapsto q_i']) : \boldsymbol{q} \in Q,\ i \in \{1, \ldots, n\},\ (q_i, \sigma, \gamma, q_i') \in \delta_i\}$.
- $Q^{\mathsf{F}} = Q_1^{\mathsf{F}} \times \cdots \times Q_n^{\mathsf{F}}$.

It can be verified that $\mathcal{L}(A) = \mathcal{L}(A_1) \between \cdots \between \mathcal{L}(A_n)$. □

Proof of Proposition 3.12. We prove the statement for a generic I using the sleep set [29] construction. Suppose $\prec$ is represented by a complete and deterministic VPA $(Q_S, Q_S^{\mathsf{in}}, \Gamma_S, \delta_S, Q_S)$. Define the VPA $A = (Q, Q^{\mathsf{in}}, \Gamma, \delta, Q^{\mathsf{F}})$ as follows:

- $Q = 2^{\widetilde{\Sigma}} \times Q_S$.

- $Q^{\mathsf{in}} = \{\varnothing\} \times Q_S^{\mathsf{in}}$.
- $\Gamma = \Gamma_S$.
- δ is the smallest subset of $(Q \times \Sigma^{\mathsf{call}} \times Q \times (\Gamma \setminus \{\bot\})) \cup (Q \times \Sigma^{\mathsf{ret}} \times \Gamma \times Q) \cup (Q \times \Sigma^{\mathsf{int}} \times Q)$ such that for all set $r \in 2^{\widetilde{\Sigma}}$, states $s, s' \in Q_S$, letter $\sigma \in \widetilde{\Sigma} \setminus r$, and stack symbol $\gamma \in \Gamma$, writing $r' = \{\sigma' \in \widetilde{\Sigma} \mid \sigma' \in I(\sigma) \wedge (((\sigma', \sigma) \in \mathrm{ord}(s)) \vee (\sigma' \in r))\}$,
  - if $(s, \sigma, s', \gamma) \in \delta_S$, then $((r, s), \sigma, (r', s'), \gamma) \in \delta$;
  - if $(s, \sigma, \gamma, s') \in \delta_S$, then $((r, s), \sigma, \gamma, (r', s')) \in \delta$; and
  - if $(s, \sigma, s') \in \delta_S$, then $((r, s), \sigma, (r', s')) \in \delta$.
- $Q^{\mathsf{F}} = Q$.

It can be shown that $\mathcal{L}(A) = \mathrm{red}_{\prec}(\widetilde{\Sigma}^*)$. □

Proof of Proposition 3.9 and Proposition 3.13. Let $\widetilde{\Sigma}_1, \widetilde{\Sigma}_2$ be the visibly pushdown alphabet of parentheses and brackets respectively. Define $\prec\colon \Sigma^* \to \mathrm{TO}(\Sigma)$ that maps strings of an even length to $(<) < [<]$, and strings of an odd length to $[<] < (<)$. Let L_1 be the language of balanced parentheses and L_2 be the language of balanced brackets.

Suppose $\mathrm{red}_{\prec}(L_1 \parallel L_2)$ is context-free, and let p be a pumping length. Pumping the string

$$\underbrace{([\cdots([}_{p \text{ times}}\underbrace{(]\cdots(]}_{p \text{ times}})^{2p}$$

leads to a contradiction, so $\mathrm{red}_{\prec}(L_1 \parallel L_2)$ cannot be context-free. □

Proof of Theorem 3.14. If L is closed under I then

$$\mathrm{red}_{I,\prec}(L) = \mathrm{red}_{I,\prec}(\widetilde{\Sigma}^*) \cap L.$$

Suppose $\prec$ is a VP contextual order and $\mathrm{red}_{I,\prec}(P_1 \parallel P_2) \subset P_1 \between P_2 \subset P_1 \parallel P_2$. It follows that

$$\mathrm{red}_{I,\prec}(P_1 \parallel P_2) = \mathrm{red}_{I,\prec}(P_1 \parallel P_2) \cap (P_1 \between P_2) = \mathrm{red}_{I,\prec}(\widetilde{\Sigma}^*) \cap (P_1 \between P_2).$$

By Proposition 3.12, Theorem 4.2, and the closedness of VPLs under intersection, it follows that $\mathrm{red}_{I,\prec}(P_1 \parallel P_2)$ is visibly pushdown. □

Proof of Proposition 3.17. Write $\widetilde{\Sigma}_i = \left\langle \Sigma_i^{\mathsf{call}}, \Sigma_i^{\mathsf{ret}}, \Sigma_i^{\mathsf{int}} \right\rangle$ $(i = 1, 2)$. Suppose for contradiction there is some word in $\mathrm{red}_{\prec}(L_1 \parallel L_2) \setminus (L_1 \parallel L_2)$. Use (,), [, and] to denote a letter in Σ_1^{call}, Σ_1^{ret}, Σ_2^{call}, and Σ_2^{ret} respectively. Consider the shortest prefix where the well-nestedness is broken, and WLOG suppose the last letter in the prefix is a]. Then] is wrongly matched with a (. Then the word has the form $\alpha(\beta]\gamma)\delta$. Note that the (is the last pending call in the prefix $\alpha(\beta$. The first letter from $\widetilde{\Sigma}_1$ in γ) can be swapped to the position of the] to give a smaller interleaving, which is a contradiction. □

Proof of Theorem 3.18. By Proposition 3.17 and Theorem 3.14. □

Proof of Lemma 4.3. The coherency of $\prec'$ is evident. The VPA for $\prec'$ can be constructed from the VPA for $\prec$ and an additional state to track (1) whether there is a pending call and (2) the component which the last pending call is from.

Any equivalence class E of $P_{\between} = P_1 \between \cdots \between P_n$ is the restriction of a unique equivalence class E' of $P_{\parallel} = P_1 \parallel \cdots \parallel P_n$ to $P_{\between}$. The definition of $\prec'$ ensures that the $\min_{\preceq}(E) = \min_{\preceq'}(E')$. Then, since any equivalence class of $P_{\parallel}$ contains a unique equivalence class of $P_{\between}$, it follows that $\mathrm{red}_{\prec}(P_{\between}) = \mathrm{red}_{\prec'}(P_{\parallel})$. □

Proof of Theorem 4.4. Suppose $\prec$ is a VPO and $P_1, \ldots, P_n$ are VPLs (well-matched by our global assumption). By Lemma 4.3, for some coherent VPO $\prec'$, we have

$$\mathrm{red}_{\prec}(P_1 \between \cdots \between P_n) = \mathrm{red}_{\prec'}(P_1 \parallel \cdots \parallel P_n).$$

Then, by Theorem 3.18, $\mathrm{red}_{\prec}(P_1 \between \cdots \between P_n)$ is visibly pushdown. □

Proof of Theorem 4.5. Suppose $A = (Q_A, \{q_A^{\mathsf{in}}\}, \Gamma_A, Q_A, \delta_A)$ and P_i is given by $A_i = (Q_i, Q_i^{\mathsf{in}}, \Gamma_i, F_i, \delta_i)$. Suppose $\prec$ is uniform w.r.t. A_1, A_2. We further assume that F_i has no outgoing transition; A_i can be modified to satisfy this condition without breaking uniformity. Assume $\Gamma_1 \cap \Gamma_2 = \{\bot\}$ and let $\Gamma_{12} = \Gamma_1 \cup \Gamma_2$. For any $q_1 \in Q_1$, $q_2 \in Q_2$, and $\gamma \in \Gamma_{12}$, define $\mathsf{enabled}(q_1, q_2, \gamma)$ as the set of outgoing letters enabled at q_1 or q_2 when the stack top is γ (if $\gamma \notin \Gamma_i$, then $\mathsf{enabled}(q_1, q_2, \gamma)$ contains no letter from Σ_i^{ret}).

We construct $R = (Q, Q^{\mathsf{in}}, \Gamma, F, \delta)$, which simulates the interleaving of A_1 and A_2, with a conceptual stack S, guided by A.

- $Q = Q_1 \times Q_2 \times Q_A \times \Gamma_{12}$. The state $(q_1, q_2, q_A, \gamma) \in Q$ maintains the states q_1, q_2, q_A in A_1, A_2, A respectively and the top γ of the stack S.
- $Q^{\mathsf{in}} = Q_1^{\mathsf{in}} \times Q_2^{\mathsf{in}} \times \{q_A^{\mathsf{in}}\} \times \{\bot\}$.
- $\Gamma = (\Gamma_{12} \times \Gamma_A \times \Gamma_{12}) \cup \{\bot\}$. When $(\gamma, \gamma_A, \gamma')$ is the stack top of R, the top two elements of S are γ and γ' (in that order), and the stack top of A is γ_A.
- $F = F_1 \times F_2 \times Q_A \times \Gamma_{12}$.
- For any $q = (q_1, q_2, q_A, \gamma) \in Q$, if $\min_{\mathsf{ord}(q_A)}(\mathsf{enabled}(q_1, q_2, \gamma)) \in \widetilde{\Sigma}_1$,
  - for all $(q_1, \sigma, q_1') \in \delta_1^{\mathsf{int}}$ and $(q_A, \sigma, q_A') \in \delta_A^{\mathsf{int}}$, we have
    $$(q, \sigma, (q_1', q_2, q_A', \gamma)) \in \delta^{\mathsf{int}};$$
  - for all $(q_1, \sigma, q_1', \gamma_1) \in \delta_1^{\mathsf{call}}$ and $(q_A, \sigma, q_A', \gamma_A) \in \delta_A^{\mathsf{call}}$, we have
    $$(q, \sigma, (q_1', q_2, q_A', \gamma_1), (\gamma_1, \gamma_A, \gamma)) \in \delta^{\mathsf{call}};$$
  - for all $(q_1, \sigma, \gamma, q_1') \in \delta_1^{\mathsf{ret}}$, $(q_A, \sigma, \gamma_A, q_A') \in \delta_A^{\mathsf{ret}}$, and $\gamma' \in \Gamma_{12}$, we have
    $$(q, \sigma, (\gamma, \gamma_A, \gamma'), (q_1', q_2, q_A', \gamma')) \in \delta^{\mathsf{ret}}.$$

  The case where $\min_{\mathsf{ord}(q_A)}(\mathsf{enabled}(q_1, q_2, \gamma)) \in \widetilde{\Sigma}_2$ is symmetrical, and δ is the smallest set defined by these rules.

Let P be the automaton for $P_1 \between P_2$ constructed in Theorem 4.2. By relating accepted runs of R to those of P, one can show that $\mathcal{L}(R) \subset \mathcal{L}(P)$. Furthermore, for any word ρ over $\widetilde{\Sigma}$, $\rho \in \mathcal{L}(R) \Leftrightarrow \rho \in \mathrm{red}_{\prec}(P_1 \between P_2)$. We omit a formal proof by induction but point out the key lemmas.

- $\Rightarrow$: For any word $\rho \in \mathcal{L}(R)$ and words α, β such that $\rho = \alpha\beta$, if R reaches state (q_1, q_2, q_A, γ) after reading α in an accepted run over ρ, then $x \in \mathsf{enabled}(q_1, q_2, \gamma)$ for any letter $x \in \mathrm{alph}(\beta)$ that can commute to the right of α.
- $\Leftarrow$: For any word $\rho \in \mathrm{red}_{\prec}(P_1 \between P_2)$ and words α, β and letter $x \in \widetilde{\Sigma}_i$ such that $\rho = \alpha x \beta$, if P reaches (q_1, q_2) with stack top γ after reading α in an accepted run over ρ, then $x \prec_\alpha y$ for all letter $y \in \mathsf{enabled}(q_1, q_2, \gamma) \setminus \widetilde{\Sigma}_i$.
  Note that this does not hold without the assumption that F_1 and F_2 have no outgoing transition.

Lastly, we observe that $|Q| = |Q_1||Q_2||Q_A||\Gamma_{12}|$. □

For any $\widetilde{\Sigma}_i$, let $\widetilde{\Sigma}_i^{\mathsf{dep}}$ be all letters in $\widetilde{\Sigma}_i$ that do not soundly commute with all letters in other components. In other words, it is the complement of $\widetilde{\Sigma}_i^{\mathsf{indep}}$.

Lemma C.1. *Let the nonterminal Y generate exactly the well-matched runs over $\widetilde{\Sigma}_i^{\mathsf{indep}}$. The following statements are equivalent:*

(1) *P_i is tail-independent (resp. head-independent).*
(2) *For every run ρ of P_i and every pair of positions $j < j'$ (resp. $j > j'$) in ρ such that $\rho_j, \rho_{j'} \in \widetilde{\Sigma}_i^{\mathsf{dep}}$, for all positions k, l in ρ such that $k \rightsquigarrow l$, we have $k < j < l \Rightarrow k < j' < l$.*
(3) *Every run of P_i is derivable from X, whose grammar is given by*

$$X \to Y \mid aX \mid cYrX \mid cXrY,$$

where a, c, and r range over all letters in Σ_i^{int}, Σ_i^{call}, and Σ_i^{ret} respectively. (resp. $X \to Y \mid Xa \mid XcYr \mid YcXr$)

Proof. We prove the equivalence for tail-independence. The case for head-independence is symmetrical.

(1) ⇒ (2): Suppose (1). Let ρ be a run of P_i. Write $\rho = uv$ as in the definition. Let $j < j'$ be a pair of positions such that $\rho_j, \rho_{j'} \in \widetilde{\Sigma}_i^{\mathsf{dep}}$. Then they are both in u. Let k, l be positions such that $k \rightsquigarrow l$ and $k < j < l$. Then k is in u and l is in v, which shows that $k < j < j' < l$.

(2) ⇒ (1): Suppose (2). Let ρ be a run of P_i. If there is no letter in $\widetilde{\Sigma}_i^{\mathsf{dep}}$ in P_i, then any division $\rho = uv$ satisfies the requirement. Otherwise, let u be the shortest prefix of ρ that contains the last letter in $\widetilde{\Sigma}_i^{\mathsf{dep}}$ and write $\rho = uv$. Then all letters in v are in $\widetilde{\Sigma}_i^{\mathsf{indep}}$.
Suppose $k \rightsquigarrow l$ and there is some letter in $\widetilde{\Sigma}_i^{\mathsf{dep}}$. It suffices to show that k is in u and l is in v.
Indeed: since any letter in $\widetilde{\Sigma}_i^{\mathsf{dep}}$ must be in u, the position k is in u; since l is behind the last letter in $\widetilde{\Sigma}_i^{\mathsf{dep}}$, the position l is in v.

(2) ⇒ (3): We will prove the stronger claim that for any well-matched run over $\widetilde{\Sigma}_i$, if for every pair of positions $j < j'$ in ρ such that $\rho_j, \rho_{j'} \in \widetilde{\Sigma}_i^{\mathsf{dep}}$, for all positions k, l in ρ such that $k \rightsquigarrow l$, we have $k < j < l \Rightarrow k < j' < l$, then ρ is derivable from X.
Let ρ be well-matched run over $\widetilde{\Sigma}_i$ such that the condition holds. Assume by induction that the statement holds for shorter runs.
If $\rho = \epsilon$, then ρ is derivable from Y and therefore X.
If $\rho = av$, where $a \in \Sigma_i^{\mathsf{int}}$, then v also satisfies the condition, no matter whether $a \in \widetilde{\Sigma}_i^{\mathsf{dep}}$ or not, and is therefore derivable from X. Thus, ρ is derivable from X.
If $\rho = cwrv$, where $c \in \Sigma_i^{\mathsf{int}}$ and r is the matching return, then w also satisfies the condition and is thus derivable from X; all letters in v must be in $\widetilde{\Sigma}_i^{\mathsf{indep}}$, so v is derivable from Y. Hence, $X \Rightarrow cXrY \Rightarrow^* \rho$.

(3) ⇒ (2): Suppose (3). Let ρ be a run of P_i. Suppose words shorter than ρ all satisfy the requirement in (2). Let $j < j'$ be a pair of positions such that $\rho_j, \rho_{j'} \in \widetilde{\Sigma}_i^{\mathsf{dep}}$. Consider the first step of a derivation of ρ from X. We will show that for all positions k, l in ρ such that $k \rightsquigarrow l$, we have $k < j < l \Rightarrow k < j' < l$.
- $X \Rightarrow Y$. This cannot happen due to the existence of ρ_j.
- $X \Rightarrow aX$. If j is the position of a, then no k, l exist. Otherwise, by induction hypothesis.
- $X \Rightarrow cYrX$. Both j and j' can only be in X on the RHS. Use induction hypothesis.
- $X \Rightarrow cXrY$. Still, both j and j' can only be in X on the RHS. Suppose $k \rightsquigarrow l$ and $k < j < l$. If k, l are the positions of c, r respectively, then they also enclose j'. Otherwise, they are inside X on the RHS, and we use induction hypothesis.

□

Proof of Proposition 5.2. By Lemma C.1, P_i is tail-independent iff $P_i \subset \mathcal{L}(X)$. Note that

$$Y \to \epsilon \mid aY \mid cYrY,$$

where a, c, and r range over all letters in $\Sigma_i^{\mathsf{int}} \cap \widetilde{\Sigma}_i^{\mathsf{indep}}$, Σ_i^{call}, and Σ_i^{ret} respectively. (Recall we assume that calls and returns fully commute with letters in other components.) Thus, the grammar for X can be rewritten to a VPG, which shows $\mathcal{L}(X)$ is a VPL.

We have $P_i \subset \mathcal{L}(X) \iff P_i \cap \mathcal{L}(X)^{\mathsf{c}} = \varnothing$. By [2, Theorem 6], one can convert the VPG for $\mathcal{L}(X)$ into a VPA with $O(1)$ states and stack alphabet of size $O(|\widetilde{\Sigma}_i|)$. By [2, Theorem 2], this VPA can be determinized to have $O(1)$ states and stack alphabet of size $O(|\widetilde{\Sigma}_i|)$. Then, $P_i \cap \mathcal{L}(X)^{\mathsf{c}}$ can be constructed as a VPA with polynomial size, and its emptiness can be decided in polynomial time.

A similar argument, rewriting the grammar to a "reversed" VPG, holds for head-independence. □

Proof of Theorem 5.3. We use the characterization (3) of tail-independence in Lemma C.1.

Let π_i be the projection $\Pi_{\widetilde{\Sigma}_i}$. Let $\widetilde{\Sigma}_i'$ be the subset of $\widetilde{\Sigma}_i$ that soundly commutes with every letter in $\bigcup_{j \neq i} \widetilde{\Sigma}_j$. Let I be the largest sound commutativity relation.

We will prove by induction that, if $P_1, \ldots, P_n$ are tail-independent singleton languages, then every word in $P_1 \parallel \cdots \parallel P_n$ is I-equivalent to some word in $P_1 \between \cdots \between P_n$.

Let $P_1 = \{\rho_1\}, \ldots, P_n = \{\rho_n\}$ be tail-independent singleton languages and suppose the statement holds on smaller instances. Let $\rho \in P_1 \parallel \cdots \parallel P_n$. If ρ contains no call (and thus no return), then the statement holds. Otherwise, write $\rho = \alpha c \beta r \gamma$, where c is the first call in ρ, and $r \in \Sigma_i^{\mathsf{ret}}$ matches c in ρ_i.

- If $\mathrm{alph}(\pi_i(\beta)) \subset \widetilde{\Sigma}_i'$, then $\rho \equiv_I \alpha c \beta' r \beta'' \gamma$, where $\beta' = \pi_i(\beta)$; in other words, we move letters in βr that are in $\widetilde{\Sigma}_i$ forward. Invoke the induction hypothesis on $(\{\pi_j(\beta''\gamma)\})_{j \in [n]}$.
- If $\mathrm{alph}(\pi_i(\beta)) \not\subset \widetilde{\Sigma}_i'$, then the derivation of ρ_i must have the form

  $$X \Rightarrow a_1 X \Rightarrow \cdots \Rightarrow a_1 \cdots a_n X \Rightarrow a_1 \cdots a_n c X r Y \Rightarrow \cdots \Rightarrow \rho_i,$$

  where $Y \Rightarrow^* \pi_i(\gamma)$. Therefore, $\rho \equiv_I \alpha c \beta \gamma' r \gamma''$, where $\gamma'' = \pi_i(\gamma)$; in other words, we move letters in $r\gamma$ that are in $\widetilde{\Sigma}_i$ backward. Invoke the induction hypothesis on $(\{\pi_j(\beta\gamma')\})_{j \in [n]}$.

For any languages $L_1, \ldots, L_n$, the shuffle (resp. well-nested shuffle) of $L_1, \ldots, L_n$ is the union of the shuffle (resp. well-nested shuffle) of $\{\rho_1\}, \ldots, \{\rho_n\}$ for all $\rho_1 \in L_1, \ldots, \rho_n \in L_n$. Therefore, if $P_1, \ldots, P_n$ are tail-independent languages that are not necessarily singletons, it still holds that every word in $P_1 \parallel \cdots \parallel P_n$ is I-equivalent to some word in $P_1 \between \cdots \between P_n$.

The case for head-independent languages is symmetrical. □

Proof of Proposition 6.1. A VP contextual order is given by a complete deterministic VPA $A = (Q, \{q^{\mathsf{in}}\}, \Gamma, Q, \delta)$ together with a map $\mathrm{ord} : Q \to \mathrm{TO}(\Sigma)$. Below, we define A and ord for each of concatenation, nested concatenation, and $\boldsymbol{s}$-lockstep. We assume $\Gamma = Q \uplus \{\bot\}$ and write the VPA like a weakly-hierarchical linear accepting NWA [3], i.e., the automaton pushes q if at state q a call letter is read. Since P_i's are well-matched, we omit the case when a pending return is read.

Concatenation Let A be a complete deterministic VPA with a single state $\star$. Let $\mathrm{ord}(\star)$ be any linear order such that $\Sigma_1 < \cdots < \Sigma_n$, i.e., if $x \in \Sigma_i$, $y \in \Sigma_j$, and $i < j$, then $x < y$.

Nested Concatenation Let A be a complete deterministic VPA with a single state $\star$. Let $\mathrm{ord}(\star)$ be any linear order such that $\Sigma_1^{\mathsf{int}} \cup \Sigma_1^{\mathsf{call}} < \cdots < \Sigma_n^{\mathsf{int}} \cup \Sigma_n^{\mathsf{call}} < \Sigma^{\mathsf{ret}}$.

***s*-lockstep** Let $Q = \{(t_1, \ldots, t_n) \mid 0 \le t_i \le s_i \text{ for each } i\} \cup \{\star\}$. Let $q^{\mathsf{in}} = (0, \ldots, 0)$. For all states $q, q', (t_1, \ldots, t_n) \in Q$ and letters $a_i \in \Sigma_i^{\mathsf{int}}$, $c_i \in \Sigma_i^{\mathsf{call}}$, $r_i \in \Sigma_i^{\mathsf{ret}}$,

$$\delta^{\mathsf{int}}(q, a_i) = q,$$
$$\delta^{\mathsf{call}}(\star, c_i) = \star,$$
$$\delta^{\mathsf{call}}((t_1, \ldots, t_n), c_i) = \begin{cases} (0, \ldots, 0, s_i - 1, s_{i+1}, \ldots, s_n), & \text{if } t_i = 0 \\ (0, \ldots, 0, t_i - 1, t_{i+1}, \ldots, t_n), & \text{otherwise} \end{cases}$$
$$\delta^{\mathsf{ret}}(q, q', r_i) = \begin{cases} q', & \text{if } q' = (t_1, \ldots, t_n) \wedge t_i = 0 \\ \star. & \text{otherwise} \end{cases}$$

$\mathrm{ord}(\star)$ is any linear order such that $\Sigma_1 < \cdots < \Sigma_n$. $\mathrm{ord}((t_1, \ldots, t_n))$ is any linear order such that $\Sigma_1^{\mathsf{int}} < \cdots < \Sigma_n^{\mathsf{int}} < \Sigma_j^{\mathsf{call}} < \Sigma_{j+1}^{\mathsf{call}} < \cdots < \Sigma_n^{\mathsf{call}} < \Sigma_1^{\mathsf{call}} < \cdots \Sigma_{j-1}^{\mathsf{call}} < \Sigma^{\mathsf{ret}}$, where j is smallest index such that $t_j > 0$.

□

Proof of Theorem 6.2. By Proposition 6.1, induction, and the fact that for any permuation π of $\{1, \ldots, n\}$,

$$P_{\pi(1)} \between \cdots \between P_{\pi(n)} = P_1 \between \cdots \between P_n.$$

□

Proof of Proposition 6.3. Suppose $\prec$ is represented by VPA $A = (Q_A, Q_A^{\mathsf{in}}, \Gamma_A, \delta_A, Q_A^{\mathsf{F}})$ together with map ord. Define VPA $A' = (Q_A, Q_A^{\mathsf{in}}, \Gamma_A \uplus \{\star\}, \delta_{A'}, Q_A^{\mathsf{F}})$, where $\delta_{A'}$ is obtained from δ_A by

- replacing every $(q, c, q', \gamma) \in \delta_A^{\mathsf{call}}$ such that $c \in \Sigma'$ with $(q, c, q'', \star)$, where $(q, f(c), q'') \in \delta_A$; and
- replacing every $(q, r, \gamma, q') \in \delta_A^{\mathsf{ret}}$ such that $r \in \Sigma'$ with $(q, r, \star, q'')$, where $(q, f(r), q'') \in \delta_A$.

Let $w = w_1 \cdots w_k$ be any prefix of any run of $P_1 \between \cdots \between P_n$. Let (q, s) be the configuration of A after reading $f(w)$. Then the configuration of A' after reading w is (q, s'), where s' is obtained from s by inserting $\star$ pushed by all $w_i \in \Sigma'$ that is a pending call.

Define map $\mathrm{ord}' : Q_A \to \mathrm{TO}(\widetilde{\Sigma})$ over the states of A' by mapping $q \in Q_A$ to any linear order extending $\{(a, b) \mid (f(a), f(b)) \in \mathrm{ord}(q)\}$. One can show that such an extension always exists, e.g., by the Order-Extension Principle. We can take any extension because reducing $P_1 \between \cdots \between P_n$ under $\mathbb{I}$ only compares a, b from different $\widetilde{\Sigma}_i$'s, and in this case $f(a)$ and $f(b)$ are already comparable by $\mathrm{ord}(q)$. The contextual order $\prec'$ represented by VPA A' together with map ord' satisfies $\mathrm{red}_{\prec'}(P_1 \between \cdots \between P_n) = \mathrm{red}_{f(\prec)}(P_1 \between \cdots \between P_n)$. □

C.1 Semantics

Let $\mathrm{top}(S)$ denote the top frame of a nonempty stack S and let

$$\mathrm{top}(\boldsymbol{S}) := (\mathrm{top}(S_1), \ldots, \mathrm{top}(S_n))$$

for a tuple of nonempty stacks $\boldsymbol{S} = (S_1, \ldots, S_n)$. We have the following lemma, which can be proved by induction. Note the right-hand side of the implication states the equation of two sets.

Lemma C.2. *For any $\rho \in \widetilde{\Sigma}_1 \between \cdots \between \widetilde{\Sigma}_n$ without pending returns, $\boldsymbol{S} \in \prod_{j=1}^n \mathrm{Stack}(V_j)$, and $S \in \mathrm{Stack}(V)$,*

$$\mathrm{top}(\boldsymbol{S}) = \mathrm{top}(S) \implies \mathrm{top}(\mathcal{M}[\![\rho]\!](\boldsymbol{S})) = \mathrm{top}(\mathcal{S}[\![\rho]\!](S)).$$

Proof of Theorem 7.1. Formally, for any $\rho \in \widetilde{\Sigma}_1 \between \cdots \between \widetilde{\Sigma}_n$ without pending returns,

- under the semantics $\mathcal{M}$, $\{A\}\rho\{B\}$ is valid iff $\mathrm{top}(\boldsymbol{S}) \models A \Rightarrow \mathrm{top}(\boldsymbol{S'}) \models B$ for all $(\boldsymbol{S}, \boldsymbol{S'}) \in \mathcal{M}[\![\rho]\!]$;
- under the semantics $\mathcal{S}$, $\{A\}\rho\{B\}$ is valid iff $\mathrm{top}(S) \models A \Rightarrow \mathrm{top}(S') \models B$ for all $(S, S') \in \mathcal{S}[\![\rho]\!]$.

Let $\rho \in P_1 \,\|\!|\, \cdots \,\|\!|\, P_n$. Suppose $\{A\}\rho\{B\}$ under the semantics $\mathcal{S}$. By Lemma C.2, for any $(\boldsymbol{S}, \boldsymbol{S'}) \in \mathcal{M}[\![\rho]\!]$, there is some $(S, S') \in \mathcal{S}[\![\rho]\!]$ such that $\mathrm{top}(\boldsymbol{S}) = \mathrm{top}(S)$ and $\mathrm{top}(\boldsymbol{S'}) = \mathrm{top}(S')$; since $\{A\}\rho\{B\}$ under the semantics $\mathcal{S}$, we have $\mathrm{top}(S) \models A \Rightarrow \mathrm{top}(S') \models B$, and therefore $\mathrm{top}(\boldsymbol{S}) \models A \Rightarrow \mathrm{top}(\boldsymbol{S'}) \models B$. This shows that $\{A\}\rho\{B\}$ under the semantics $\mathcal{M}'$. The other direction is symmetrical. □

D Extra Details

D.1 CHC Encodings

We use the alphabet introduced in Appendix A.1.

Directly Encode an NWA. Let $A = (Q, Q^{\mathsf{in}}, Q^{\mathsf{F}}, \widetilde{\Sigma}, \delta)$ be a linearly accepting, weakly hierarchical NWA without pending returns such that for any $(q, q', \boxed{\text{ret f}}, q'') \in \delta$, any outgoing call letter at q' is $\boxed{\text{call f}}$. For each state $q \in Q$, declare a predicate $I_q(\overline{\boldsymbol{x}}, \boldsymbol{l}, r)$, which represents an assertion that holds at q relating the function arguments $\overline{\boldsymbol{x}}$, the local variables $\boldsymbol{l}$, and the return address $r \in Q \uplus \{\bot\}$ at the current stack frame, where $\bot$ denotes the stack bottom. Each transition in δ translates to a constrained Horn clause:

- $(q, a, q') \in \delta$ such that $a \in \Sigma^{\mathsf{int}}$ translates to

$$I_q(\overline{\boldsymbol{x}}, \boldsymbol{l}, r) \wedge \mathcal{S}[\![a]\!](\boldsymbol{l}, \boldsymbol{l'}) \implies I_{q'}(\overline{\boldsymbol{x}}, \boldsymbol{l'}, r).$$

- $(q, \boxed{\text{call f}}, q') \in \delta$ translates to

$$I_q(\overline{\boldsymbol{x}}, \boldsymbol{l}, r) \wedge \bigwedge_{p \in \mathrm{param}(f)} p' = \overline{p}' = p \implies I_{q'}(\overline{\boldsymbol{x}}', \boldsymbol{l'}, q).$$

- $(q, q', \boxed{\text{ret f}}, q'') \in \delta$ translates to

$$I_q(\overline{\boldsymbol{x}}, \boldsymbol{l}, q') \wedge I_{q'}(\overline{\boldsymbol{x}}', \boldsymbol{l'}, r) \wedge \left(\bigwedge_{p \in \mathrm{param}(f)} \overline{p} = p' \right) \wedge \left(\bigwedge_{l \in \boldsymbol{l}} l'' = \mathsf{ite}(l \in \mathrm{output}(f),\ l,\ l') \right) \implies I_{q''}(\overline{\boldsymbol{x}}', \boldsymbol{l''}, r).$$

To verify a pair of pre/postconditions pre and post, add the clauses

$$\mathrm{pre} \implies I_{q^{\mathsf{in}}}(\overline{\boldsymbol{x}}, \boldsymbol{l}, \bot),$$
$$I_{q^{\mathsf{F}}}(\overline{\boldsymbol{x}}, \boldsymbol{l}, r) \wedge \neg \mathrm{post} \implies \bot$$

for all $q^{\mathsf{in}} \in Q^{\mathsf{in}}$, $q^{\mathsf{F}} \in Q^{\mathsf{F}}$, and $r \in Q \uplus \{\bot\}$. If it is known that $\mathcal{L}(A)$ is well-matched, then it suffices to take $r = \bot$.

The following proposition states the soundness and completeness of the encoding:

Proposition D.1. *Let $A = (Q, Q^{\mathsf{in}}, Q^{\mathsf{F}}, \widetilde{\Sigma}, \delta)$ be a linearly accepting, weakly hierarchical NWA without pending returns such that for any $(q, q', \boxed{\text{ret f}}, q'') \in \delta$, any outgoing call letter at q' is $\boxed{\text{call f}}$. Let $P = \mathcal{L}(A)$. For the recursive program $(P, \mathcal{S})$, the CHC encoding is satisfiable iff $\{\mathrm{pre}\}P\{\mathrm{post}\}$.*

Proof. $\Rightarrow$: Suppose the CHC encoding is satisfiable. Let ρ be a nested word with an accepting run $q_0 \xrightarrow{\rho_1} \cdots \xrightarrow{\rho_n} q_n$. Then ρ can be annotated with an *inductive sequence of nested interpolants*[4] [35, Definition 5] $J_0, \ldots, J_n$, where $J_i = \{(\bar{\boldsymbol{x}}, \boldsymbol{l}) \mid I_{q_i}(\bar{\boldsymbol{x}}, \boldsymbol{l}, r)\}$ (we use $(\bar{\boldsymbol{x}}, \boldsymbol{l})$ to denote the corresponding valuation) and r is the state just before the last pending call in $\rho_1 \cdots \rho_i$, or $\bot$ if no such state exists. By the construction of the CHC, pre $\Rightarrow q_0$ and $q_n \Rightarrow$ post. One can show that such an annotation implies $\{\mathsf{pre}\}\rho\{\mathsf{post}\}$.

$\Leftarrow$: Prove by induction that given a derivation of $I_q(\bar{\boldsymbol{x}}, \boldsymbol{l}, r)$, one can construct a nested word ρ such that

- There is a partial run of A over ρ that ends at state q. Moreover, if $r = \bot$, then ρ is well-matched; otherwise, ρ has a pending call, r is the state before the last pending call in this run.
- ρ can be annotated with an inductive sequence of nested interpolants that starts with some $\{\nu\}$ such that $\nu \models \mathsf{pre}$ and ends with $\{(\boldsymbol{x}, \boldsymbol{l})\}$.

Then, from a derivation of $\bot$, one can extract a counterexample that witnesses $\{\mathsf{pre}\}P\{\mathsf{post}\}$ does not hold. □

Encode a VPG. The semantics of a well-matched syntactic run can be given without an explicit stack, as a relation on valuations rather than stacks of valuations. Let T be the set of all valuations. Unlike the direct encoding of an NWA, the ghost variables for function arguments are not needed.

- $\mathcal{S}'[\![\epsilon]\!] := \{(\nu, \nu) : \nu \in T\}$.
- $\mathcal{S}'[\![\boxed{\text{assume } \phi}]\!] := \{(\nu, \nu) : \nu \models \phi\}$.
- $\mathcal{S}'[\![\boxed{x := t}]\!] := \{(\nu, \nu \oplus \{x \mapsto \nu(t)\}) : \nu \in T\}$.
- $\mathcal{S}'[\![\boxed{\text{call } q}\,\rho\,\boxed{\text{ret } q}]\!] := \{(\nu, \nu \oplus \bigcup_y \{y \mapsto \mu'(y)\}) : (\mu, \mu') \in \mathcal{S}'[\![\rho]\!] \wedge \bigwedge_x \mu(x) = \nu(x)\}$, where ρ is well-matched and x, y range over the input and output variables of q respectively.
- $\mathcal{S}'[\![\rho\tau]\!] = \mathcal{S}'[\![\tau]\!] \circ \mathcal{S}'[\![\rho]\!]$, where ρ, τ are well-matched.

Let X be all variables in the program. Given a well-matched VPG $G = (V, \widetilde{\Sigma}, P, S)$, for each nonterminal $L \in V$, declare a predicate $P_L(X, X')$ that *overapproximates* the semantics of all words produced by L. Each production in P translates to a constrained horn clause:

- $L \to aL'$ translates to

$$\mathcal{S}[\![a]\!](X, X') \wedge P_{L'}(X', X'') \implies P_L(X, X'').$$

- $L \to \boxed{\text{call } q}\,L'\,\boxed{\text{ret } q}\,L''$ translates to

$$\left(\bigwedge_x x' = x\right) \wedge \left(\bigwedge_y y^{(3)} = y''\right) \wedge \left(\bigwedge_{z \neq y} z^{(3)} = z\right) \wedge P_{L'}(X', X'') \wedge P_{L''}(X^{(3)}, X^{(4)}) \implies P_L(X, X^{(4)})$$

  where x, y range over the input and output variables of q respectively.
- $L \to \epsilon$ translates to

$$\bigwedge_{z \in X} z' = z \implies P_L(X, X').$$

To verify a safety property $p(X, X')$, add for all start symbol L the clause

$$P_L(X, X') \wedge \neg p(X, X') \implies \bot.$$

[4] In the context of this definition, an interpolant is a set of valuations. For our purpose, we remove the requirement that $J_0 = \top$ and $J_n = \bot$.

D.2 Parameterized Lockstep

Define $G_0 := \bigcup_{i=1}^{n} (G_i \cup \{Y' \to cWrE : (Y \to cWrV) \in G_i\}) \cup \{E \to \epsilon\}$. The rules are listed in Fig. 8. The nonterminal symbols of the grammar G of the $\boldsymbol{m}$-lockstep of $\mathcal{L}(G_1), \ldots, \mathcal{L}(G_n)$ have the form $[\boldsymbol{s}, \boldsymbol{t}, \prod_{i=0}^{k} W_i]$, where $\boldsymbol{s}, \boldsymbol{t}$ satisfy the constraints in Section 6.1, W_i is a nonterminal in G_0 for each i, and the length and maximum element of $\boldsymbol{s}$ are bounded by those of $\boldsymbol{m}$. Moreover, to account for Ls-Eps, $[\boldsymbol{s}, \boldsymbol{t}, \prod_{i=1}^{j-1} W_i \times E \times \prod_{i=j}^{k} W_i]$ and $[\boldsymbol{s}, \boldsymbol{t}, \prod_{i=1}^{k} W_i]$ are treated as the same symbol for any $k > 1$.

$$\text{G-LsOne}\quad \frac{(X \to \alpha) \in G_0}{([\boldsymbol{s}, \boldsymbol{t}, X] \to \alpha) \in G}$$

$$\text{G-LsInt}\quad \frac{\forall i \in [k].\, (W_i \to \alpha_i) \in G_0 \qquad (U_j \to aW_j) \in G_0 \qquad \text{none of } \alpha_i \text{ where } i < j \text{ starts with an internal}}{\left([\boldsymbol{s}, \boldsymbol{t}, \prod_{i=1}^{j-1} W_i \times U_j \times \prod_{i=j+1}^{k} W_i] \to a[\boldsymbol{s}, \boldsymbol{t}, \prod_{i=1}^{k} W_i]\right) \in G}$$

$$\text{G-LsFstCall}\quad \frac{\forall i \in [k].\, (Y_i \to c_i W_i r_i V_i) \in G_0 \qquad \boldsymbol{t} = \boldsymbol{0}}{\left([\boldsymbol{s}, \boldsymbol{t}, \prod_{i=1}^{k} Y_i] \to c_1\, [\boldsymbol{s}, \mathrm{dec}_{\boldsymbol{s}}(\boldsymbol{t}), W_1 \times \prod_{i=2}^{k} Y_i']\, r_1\, [\boldsymbol{s}, \boldsymbol{t}, \prod_{i=1}^{n} V_i]\right) \in G}$$

$$\text{G-LsRstCall}\quad \frac{\forall i \in [k].\, (Y_i \to c_i W_i r_i V_i) \in G_0 \qquad \boldsymbol{t} \neq \boldsymbol{0} \qquad j = \min\{i : t_i > 0\}}{\left([\boldsymbol{s}, \boldsymbol{t}, \prod_{i=1}^{k} Y_i] \to c_j\, [\boldsymbol{s}, \mathrm{dec}_{\boldsymbol{s}}(\boldsymbol{t}), \prod_{i=1}^{j-1} Y_i' \times W_j \times \prod_{i=j+1}^{k} Y_i']\, r_j\, V_j\right) \in G}$$

Fig. 8. VPG productions for parameterized lockstep . $[k]$ denotes the set $\{1, \ldots, k\}$.

D.3 Implementation of VPG Canonical Reductions via a Queue

Algorithm 1 Direct VPG reduction

Require: G_i's are uniform, with disjoint nonterminals
  procedure Reduction($G_1, \ldots, G_n$)
    $V' \leftarrow$ Start-Symbols($G_1.start, \ldots, G_n.start$)
    $P \leftarrow \bigcup_i G_i.productions$
    $Q \leftarrow$ Queue(V'), $V \leftarrow$ Set(V'), $R \leftarrow$ Vpg(V')
    while Q is nonempty **do**
      $X \leftarrow$ Dequeue(Q)
      $(Y_1, \ldots, Y_m) \leftarrow$ Components(X)
      for all $(Y_1 \to \alpha_1), \ldots, (Y_m \to \alpha_m)$ in P **do**
        $rhs, splits \leftarrow$ Product(X, $((Y_1 \to \alpha_1), \ldots, (Y_m \to \alpha_m))$)
        Add the production $X \to rhs$ to R
        Add the productions $splits$ to P
        for all nonterminal B in β **do**
          if $B \notin V$ **then**
            Enqueue(Q, B)
            Add(V, B)
    return R

Algorithm 1 outlines the scheme for constructing a (canonical) reduction as a VPG that maintains a queue Q of reachable symbols, and is parametric on the implementations of the procedures Start-Symbols, Components, and Product which differ from one reduction to another. By maintaining a queue Q of reachable symbols, Algorithm 1 only adds those productions that are truly needed, making the construction efficient. We assume the input VPGs are *uniform*: for any productions $X \to \alpha$ and $X \to \beta$, the right-hand sides α, β both start with a call, both start with an internal, or both are empty. Any VPG can be converted in linear time to an equivalent one that is uniform. See also Algorithm 2 and 3.

Algorithm 2 Direct VPG reduction for nested_concat of two components

```
1:  function Start-Symbols(V_1^st, V_2^st)
2:      return {[X_1, X_2] : X_1 ∈ V_1^st, X_2 ∈ V_2^st}

3:  function Components(X)
4:      if X is of the form [Y_1, Y_2] then
5:          return (Y_1, Y_2)
6:      else
7:          return (X)

8:  function Product(X, p)
9:      (Y_1 → α_1), ..., (Y_n → α_n) ← p
10:     if n = 1 then
11:         return α_1, ∅
12:     else if α_1 is of the form cZ_1rW_1 then
13:         return c[Z_1, Y_2]rW_1, ∅
14:     else if α_1 is of the form aZ_1 then
15:         return a[Z_1, Y_2], ∅
16:     else
17:         return α_2, ∅
```

E Benchmarks

Table 1, 2, 3, and 4 describe the benchmarks that we use for evaluating HyRec. Table 1 lists the benchmarks adapted from the safe relational benchmarks from [48]; if a benchmark is about recursive functions that are better understood by looking at the code, we list the name of the original benchmark and the number k when it is viewed as a k-property; these recursive functions only call themselves and at most once. The Ackermann benchmark from [48] is discussed in Section 6.2. Table 2 lists the benchmarks adapted from the iterative sequential benchmarks from [27].

The benchmarks involve a variety of common functions on natural numbers and aggregate functions on arrays. We make the following assumptions:

- The implementation of mult recurses on the first parameter.
- On arrays, the functions min, max, sum scan linearly from left to right unless otherwise specified. On binary trees simulated by arrays via a binary heap, they follow the structure of the heap.
- For arrays and trees, pointwise operations (sum and scaling) are fused with aggregate functions. For example, $\mathrm{sum}(A + 2B)$ is implemented as a single function that scans A, B simultaneously from left to right.

Algorithm 3 Direct VPG reduction for $\boldsymbol{s}$-lockstep

function START-SYMBOLS($V_1^{\mathrm{st}}, \ldots, V_n^{\mathrm{st}}$)
  return $\{[\boldsymbol{s}, \boldsymbol{0}, \boldsymbol{X}] : X_1 \in V_1^{\mathrm{st}}, \ldots, X_n \in V_n^{\mathrm{st}}\}$

function COMPONENTS(X)
  if X is of the form $[\boldsymbol{s}, \boldsymbol{t}, Y]$ **then**
    return Y
  else
    return (X)

function PRODUCT(X, $\boldsymbol{p}$)
  $(Y_1 \to \alpha_1), \ldots, (Y_n \to \alpha_n) \leftarrow \boldsymbol{p}$
  if $n = 1$ **then**
    return α_1, $\emptyset$
  $[\boldsymbol{s}, \boldsymbol{t}, Y] \leftarrow X$
  if $n > 1$ and there is i such that $\alpha_i = \epsilon$ **then**
    drop the i-th component in $\boldsymbol{s}, \boldsymbol{t}, Y, \boldsymbol{p}$
    return PRODUCT($[\boldsymbol{s}, \boldsymbol{t}, Y]$, $\boldsymbol{p}$)
  else if there is a smallest i such that α_i is of the form aZ_i **then**
    $\boldsymbol{Y'} \leftarrow \boldsymbol{Y}[i \mapsto Z_i]$
    return $a[\boldsymbol{s}, \boldsymbol{t}, \boldsymbol{Y'}]$, $\emptyset$
  else
    write $\alpha_i = c_i Z_i r_i W_i$ for each i
    $\boldsymbol{t'} \leftarrow (\boldsymbol{t} - 1) \bmod \boldsymbol{s}$
    if $\boldsymbol{t} = \boldsymbol{0}$ **then**
      let $U_i = [c_i Z_i r_i]$ for each i
      $\boldsymbol{Z'} \leftarrow \boldsymbol{U}[1 \mapsto Z_1]$
      return $c_1[\boldsymbol{s}, \boldsymbol{t'}, \boldsymbol{Z'}]r_1[\boldsymbol{s}, \boldsymbol{t}, \boldsymbol{W}]$, $\{U_i \to c_i Z_i r_i E : i = 2, \ldots, n\} \cup \{E \to \epsilon\}$
    else
      $i \leftarrow \min\{j : t_j > 0\}$
      $\boldsymbol{Z'} \leftarrow \boldsymbol{Y}[i \mapsto Z_i]$
      return $c_i[\boldsymbol{s}, \boldsymbol{t'}, \boldsymbol{Z'}]r_i W_i$, $\emptyset$

For succinctness, we may omit ranges of variables that can be inferred from context, such as $n \geq 0$.

We label benchmarks 1–30, 32, 41–57, 77–99 and 103 with $(1, \ldots, 1)$-lockstep; 33–40, 58–73, and 99 with $\boldsymbol{s}$-lockstep where $\boldsymbol{s} \neq (1, \ldots, 1)$; 31, 74–76, and 100–102 with (1, 1)-lockstep of one component and the nested concatenation of the other two.

Note that for benchmarks that are theoretically out of reach for our tool, we still label them with $(1, \ldots, 1)$-lockstep for a uniform treatment.

No.	Description
1	`add-horn`, a 2-property
2	`barthe`, a 2-property
3	$\mathrm{div}(n+d,d) = \mathrm{div}(n,d)+1$
4	$d < d' \implies \mathrm{div}(n,d) \geq \mathrm{div}(n,d')$
5	`double-inc-loop`, a 2-property
6	$n^n < n!$ for $n > 1$
7	$n < n' \implies \mathrm{fib}(n) \leq \mathrm{fib}(n')$
8	$\mathrm{fib}(n) = \mathrm{fib}'(n+1)$, where the first term in fib' is $\mathrm{fib}'(1)$
9	fib is equivalent to fib', where the two recursive calls are swapped
10	$\gcd(n,m) = \gcd(m,n)$
11	`inc-loop-1`, a 2-property
12	`inc-loop-2`, a 3-property
13	`inc-loop-5`, a 6-property
14	`limit-1`, a 2-property
15	`limit-2`, a 2-property
16	$1 < n < n' \implies \mathrm{lucas}(n) \leq \mathrm{lucas}(n')$
17	$n > 2 \implies \mathrm{fib}(n) < \mathrm{lucas}(n)$
18	McCarthy91 is equivalent to McCarthy91′, where the two branches are swapped
19	$n \leq n' \implies \mathrm{McCarthy91}(n) \leq \mathrm{McCarthy91}(n')$
20	$q = \mathrm{div}(n,d) \implies \mathrm{mult}(q,d) + \mathrm{mod}(n,d)$
21	$\mathrm{mod}(n+d,d) = \mathrm{mod}(n,d)$
22	$a < a', b > 0 \implies \mathrm{mult}(a,b) < \mathrm{mult}(a',b)$
23	$0 < a < a', 0 < b < b' \implies \mathrm{mult}(a,b) < \mathrm{mult}(a',b')$
24	the number of digits of n, i.e., $\lfloor \log_{10} n \rfloor + 1$, is monotone
25	$0 < x < y,\ n > 0 \implies x^n < y^n$
26	$x, y > 0,\ n > 0 \implies x^n + y^n \leq (x+y)^n$
27	`sum-6`, a 7-property
28	`twice-sum-1`, a 2-property

Table 1. Benchmarks adapted from [48]

No.	Description
29	array equality is symmetric
30	array equality is transitive
31	$\text{mult}(a, c) + \text{mult}(b, c) = \text{mult}(a + b, c)$
32	$\text{mult}(a, b) + \text{mult}(a, c) = \text{mult}(a, b + c)$
33	equivalence before and after unrolling a loop twice
34	equivalence before and after unrolling a loop three times
35	equivalence before and after unrolling a loop four times
36	equivalence before and after unrolling a loop five times
37	equivalence before and after unrolling a loop twice, with cleaning up
38	equivalence before and after unrolling a loop three times, with cleaning up
39	equivalence before and after unrolling a loop four times, with cleaning up
40	equivalence before and after unrolling a loop five times, with cleaning up
41	the lexicographic order comparator is symmetric
42	a variation of the lexicographic order comparator is symmetric
43	the lexicographic order comparator respects equality
44	a variation of the lexicographic order comparator respects equality
45	the lexicographic order is transitive

Table 2. Benchmarks adapted from the iterative sequential benchmarks from [27]

No.	Description
46	the function inc that recursively sums up n ones is injective
47	$a > 0 \wedge b < b' \implies \text{mult}(a, b) < \text{mult}(a, b')$
48	$\text{div}(n, d) = \text{div}(2n, 2d)$
49	$n \leq n' \implies \text{div}(n, d) \leq \text{div}(n', d)$
50	$n \leq n', d \geq d' \implies \text{div}(n, d) \leq \text{div}(n', d')$
51	$2\ \text{mod}(n, d) = \text{mod}(2n, 2d)$
52	$(1 + 2) + \cdots + ((2n - 1) + 2n) = (2 + 4 + \cdots + 2n) + (1 + 3 + \cdots + (2n - 1))$
53	$m < n \implies m! \leq n!$
54	$a > 1,\ 0 < m < n \implies a^m < a^n$
55	$\text{fib}(n) = 3\ \text{fib}(n - 3) + 2\ \text{fib}(n - 4)$ for $n > 3$
56	the Tribonacci sequence is monotone
57	the Tetranacci sequence is monotone
58	$2\ \text{inc}(n) = \text{inc}(2n)$
59	$\text{inc}(2n) = 2\ \text{inc}(n)$
60	$2\ \text{mult}(a, b) = \text{mult}(2a, b)$
61	$\text{mult}(2a, b) = 2\ \text{mult}(a, b)$
62	$3\ \text{mult}(a, b) = \text{mult}(3a, b)$
63	$\text{mult}(3a, b) = 3\ \text{mult}(a, b)$
64	$3\ \text{mult}(2a, b) = 2\ \text{mult}(3a, b)$
65	$2\ \text{mult}(3a, b) = 3\ \text{mult}(2a, b)$
66	$d \mid n \implies d \mid 2n$
67	$d \mid 2n \impliedby d \mid n$
68	$d \mid n \implies d \mid 3n$
69	$d \mid 3n \impliedby d \mid n$
70	$2\ \text{div}(n, d) \leq \text{div}(2n, d)$
71	$\text{div}(2n, d) \geq 2\ \text{div}(n, d)$
72	$3\ \text{div}(n, d) \leq \text{div}(3n, d)$
73	$\text{div}(3n, d) \geq 3\ \text{div}(n, d)$
74	$\text{div}(n, d) + \text{div}(n', d) \leq \text{div}(n + n', d)$
75	$d \mid n,\ d \mid n' \implies d \mid (n + n')$
76	$\text{mod}(n, d) = 0,\ \text{mod}(n', d) = 0 \implies \text{mod}(n + n', d) = 0$
77	$\text{mult}(a, b) = \text{mult}(b, a)$

Table 3. More arithmetic benchmarks

No.	Description
78	$\mathrm{sum}(A) + \mathrm{sum}(B) = \mathrm{sum}(A + B)$
79	the lexicographic order is antisymmetric
80	$\min(A + B) \geq \min(A) + \min(B)$
81	$\min(A) \leq \max(A)$ for nonempty array A
82	$2\ \mathrm{sum}(A) = \mathrm{sum}(A)$
83	$2\ \mathrm{sum}(A) + \mathrm{sum}(B) = \mathrm{sum}(2\,A + B)$
84	pointwise $\leq$ on arrays is transitive
85	pointwise $\leq$ on arrays is reflexive
86	pointwise $\leq$ on arrays is antisymmetric
87	$A \leq B \implies 2A \leq 2B$
88	$A \leq B \implies A + C \leq A + C$
89	$\max(A) < \min(B) \implies \max(B) < \min(C) \implies \min(A) < \min(C)$
90	$\max(A) < \min(B) \implies \max(B) < \min(C) \implies \min(A) < \max(C)$
91	$\min(T) \leq \max(T)$ for tree T
92	$\min(T) \leq \max(T)$ for tree T, another implementation of min, max
93	$\max(T) < \min(T') \implies \min(T) < \min(T')$
94	$T \leq T' \implies 2T \leq 2T'$, where T, T' has the same shape and $\leq$ is pointwise
95	$\mathrm{sum}(T) + \mathrm{sum}(T') = \mathrm{sum}(T + T')$
96	$\min(A) \leq \max(A)$, where min, max recurse on the two halves of A
97	$\max(A) < \min(B) \implies \min(A) < \min(B)$, where min, max recurse on the two halves
98	$\min(T) + \min(T') \leq \min(T + T')$
99	sum up two consecutive elements at a time = sum up one element at a time
100	$\min(A[i..k]) = \min(\min(A[i..j]), \min(A[j..k]))$
101	$\max(A[i..k]) = \max(\max(A[i..j]), \max(A[j..k]))$
102	$\mathrm{sum}(A[i..k]) = \mathrm{sum}(A[i..j]) + \mathrm{sum}(A[j..k])$
103	$i < i' < j' < i \implies \mathrm{sum}(A[i'..j']) < \mathrm{sum}(A[i..j])$ if every element of A is positive

Table 4. More array benchmarks