Subclasses of Ptime interpreted by programming languages

Siddharth Bhaskar, Cynthia Kop, and Jakob Grue Simonsen

Abstract

We consider the *cons-free programming language* of Neil Jones, a simple pure functional language, which decides exactly the polynomial-time relations and whose tail recursive fragment decides exactly the logarithmic-space relations. We exhibit a close relationship between the running time of cons-free programs and the running time of logspace-bounded auxiliary pushdown automata. As a consequence, we characterize intermediate classes like NC in terms of resource-bounded cons-free computation. In so doing, we provide the first "machine-free" characterizations of certain complexity classes, like P-uniform NC.

Furthermore, we show strong polynomial lower bounds on cons-free running time. Namely, for every polynomial p, we exhibit a relation $R \in$ Ptime such that any cons-free program deciding R must take time at least p almost everywhere. Our methods use a "subrecursive version" of Blum complexity theory, and raise the possibility of further applications of this technology to the study of the fine structure of Ptime.

1 Introduction

By the 1970's, several famously hard questions about the relationship among fundamental complexity classes such as Logspace, Ptime, NPtime, and Pspace had been recognized as worthy of intense interest. Structural complexity theory up to that point had typically relied on recursion-theoretic techniques such as diagonalization and priority arguments, but these seemed to have little to say about the relationship among these classes.

In response, complexity theorists started studying what we might call "second-generation" models of computation such as boolean circuits and alternating Turing machines. On one hand, it seemed like it might be possible to prove lower bounds on non-uniform models of computation such as boolean circuits via hard combinatorics, plausibly to the point of separating the fundamental complexity classes above. On another hand, new connections were found between these fundamental classes and (theretofore) nonstandard paradigms such as alternation, interaction, parallelism, and probabilistic computation.

This program, without ever resolving the open questions about the original complexity classes, led to an enormous increase in our understanding of them. It also led to the formulation of other classes fundamental in their own right. An important example is the class NC (*Nick's class*), and its ramification in the form of the *NC hierarchy* $\{\mathrm{NC}^i\}_{i<\omega}$. These and related classes interpolate Logspace and Ptime as follows, endowing Ptime with a fine structure in many ways more robust than the deterministic time-$O(n^c)$ hierarchy.

$$\textsc{Logspace} \subseteq \textsc{Log}\mathrm{DCFL} \subseteq \textsc{Log}\mathrm{CFL} = \mathrm{SAC}^1 \subseteq \mathrm{AC}^1 \subseteq \mathrm{NC}^2 \subseteq \cdots \subseteq \mathrm{NC} \subseteq \textsc{Ptime}.$$

The precise definition of all of these classes is not as important as the big picture, but suffice it to say that each of them can be realized by logarithmic space alternating Turing machines (ATM) running within a prescribed time [15, 16], polynomial-sized circuit families of a prescribed depth, or auxiliary pushdown automata (AuxPDAs) running within a prescribed time.

In this paper, we study *program*-based complexity theory; that is, resource-bounded computation by pure functional programs that operate on string data. In particular, we study the *cons-free programs* of Jones [9]. This class of programs computes exactly the Ptime relations and its tail recursive fragment computes exactly the Logspace relations. Here, we show that we can recover something like the hierarchy above—and in particular, the class NC—by considering *time-bounded* cons-free computation.

While there are many other characterizations of complexity classes in terms of programming languages, the primary focus of this research seems to be towards studying termination, correctness, efficiency, and expressiveness of these languages. By contrast, we argue that cons-free programs are a serious candidate for a "standard model of computation" for structural complexity theory. For one, the cons-free framework is versatile: not only do their time classes interpolate Logspace with Ptime, but their higher-order variants capture classes in the deterministic exponential time hierarchy [11].

Furthermore, in this paper, we demonstrate *absolute* polynomial lower bounds for cons-free programs—in other words, for any $c < \omega$, we construct a problem which can be decided by some cons-free program, but which necessarily takes time at least n^c on inputs of length n. While this does not resolve any open problems, it stands in stark contrast to circuit depth, where we do not know any super-constant lower bounds for polynomial-size uniform circuit families.

Our method for showing this lower bound is recursion-theoretic in flavor. While we do not know which general barriers (relativization, natural proofs, etc.) constrain this machinery, we believe that our result demonstrates that cons-free programs are a model of computation worthy of further consideration.

1.1 Our contributions

In this paper, we show that cons-free running time is closely related to the running time of (deterministic) logspace AuxPDAs (Theorem 7). Consequently, we obtain a characterization of NC by cons-free programs running in quasipolynomial time and of LogDCFL by cons-free programs running in polynomial time (Theorems 8 and 9). We believe this is the first characterization of these classes by a "natural" functional programming language (i.e., a language with the ability to make general recursive definitions, unlike [5, 13]).

We show furthermore that the correspondence between running times refines to a correspondence between the number of moves of an AuxPDA's input head and the number of calls to `hd` by a cons-free program. Hence we obtain a characterization of PUNC (*P-uniform* NC) by cons-free programs with a quasipolynomial number of calls to `hd` (Theorem 10). We believe this is the first known characterization of PUNC, or any other P-uniform class, by a non-machine, non-circuit model of computation.

We finally show, for every $c < \omega$, the existence of a problem in Ptime (and hence cons-free computable) that requires time at least n^c to decide by any cons-free program (Theorems 11 and 12). These results are summarized in Table 1.

Complexity class	Cons-free characterization
(LOGSPACE)	(tail recursive)
LOGDCFL	$n^{O(1)}$ time
NC	$2^{\log^{O(1)}(n)}$ time
PUNC	$2^{\log^{O(1)}(n)}$ calls to `hd`
(PTIME)	(unbounded / $2^{n^{O(1)}}$ time)

Table 1: Our main results are the middle three rows of this table: the complexity classes on the left are captured by cons-free programs under the constraints on the right. Notice that any tail recursive program runs in polynomial time. In addition we show cons-free time $O(n^c) \subsetneq$ PTIME for each $c < \omega$.

1.2 Preliminaries

We assume familiarity with the basics of Turing machines, circuits, and automata at the level of standard textbooks, e.g. [17, 3]. In particular, it will be helpful to have some fluency with AuxPDAs (but we will recall the definition in Section 5). An acquaintance with basic recursion theory, the notion of an indexing, and in particular having seen Blum's axioms [4], will be helpful but is not necessary. Finally, we assume familiarity with proving correctness and estimating complexity of programs in a simple, pure functional language.

Here is a glossary for various classes interpolating PTIME and LOGSPACE discussed above. Primary sources for the first three items in the list include [18, 15, 16, 20]; these classes are also discussed in graduate-level textbooks such as [3]. The P-uniform classes in the last bullet point were introduced in [1].

- NC is the class of problems acceptable by logspace AuxPDAs (deterministic or nondeterministic) in quasipolynomial ($2^{\log^{O(1)}(n)}$) time, by logspace ATMs in polylogarithmic ($\log^{O(1)}(n)$) time, and by logspace-uniform circuit families in polylogarithmic depth. It is an extremely robust class, admitting a number of other characterizations especially by models of parallel computation.

- NC^k (resp. AC^k) is the class of all problems acceptable by logspace-uniform bounded-fanin (resp. unbounded-fanin) circuit families in depth $O(\log^k(n))$. This hierarchy is *interleaved with* the hierarchy of problems solvable by AuxPDAs in time $2^{O((\log n)^k)}$, and ATMs in time $O((\log n)^k)$. (In other words, each hierarchy is dense in the other as linear orders.) The time-$2^{O((\log n)^k)}$ hierarchy on nondeterministic AuxPDAs corresponds to the circuit class SAC^k of depth $O(\log^k(n))$ logspace-uniform circuit families of *semi-unbounded* fan-in.

- When $k = 1$, $2^{O((\log n)^k)} = n^{O(1)}$, i.e., the set of polynomials. The class of problems decidable by deterministic and nondeterministic AuxPDAs in time $n^{O(1)}$ are the classes LOGDCFL and LOGCFL respectively. These are so named because they are the closure of the deterministic context-free and context-free languages respectively under logarithmic-space reductions.

- Any of the logspace-uniform circuit families also have *P-uniform* versions, which are more expressive. For any circuit class that is also an AuxPDA-time class, the respective P-uniform class can be characterized by AuxPDAs with an identical constraint on moves of the input head. For example PUNC (P-uniform NC) can be characterized by (nondeterministic or deterministic) AuxPDAs which move their input heads at most $2^{\log^{O(1)}(n)}$ times.

1.3 Related work

There are three main bodies of research that underlie the present paper. One is the study of ATMs, AuxPDAs, circuit families, and the resulting complexity classes discussed above. Secondly, there is the study of abstract complexity classes or "Blum complexity theory," and finally, the study of cons-free computation.

The first body of research encompasses a wide area of complexity theory, both classical and modern, and is highly connected to other areas of theory and indeed computer science in general (such as parallel architectures). It is impossible to give an adequate summary of the history of this field in such a small space, so we shall content ourselves with listing some papers which we view as "inflection points" of particular relevance to our work.

Alternation was introduced by Chandra, Kozen, and Stockmeyer [6]. Pushdown automata are classical models of computation from formal language theory, and the study of AuxPDA complexity classes such as LogCFL was initiated by Sudborough [18]. Ruzzo [15, 16] established general connections between ATMs, AuxPDAs, and circuits. Venkateswaran [20] formulated the notion of a semi-unbounded circuit and connected it with LogCFL. Allender [1] introduced the P-uniform versions of important circuit classes and bounded-access computation on ATMs and AuxPDAs.

Blum complexity theory was first introduced by Blum [4]. The standard model of computation for classical recursion theory is an indexing of the partial recursive functions that satisfies a handful of axioms. Blum extended this framework by associating a complexity measure Φ_e with every program index e, and stipulating that the relation $\Phi_e(x) \leq y$ should be a decidable relation in (e, x, y). There are very few proposed adaptations of the Blum axioms to indexings of complexity classes, but one of these is due to Alton [2], who used it to illustrate a few proof-of-concept examples.

Cons-free programs were introduced by Jones [9] to capture Logspace and Ptime. This was extended to a correspondence between higher-type cons-free programs and the deterministic exponential hierarchy in [10]. In [11], Kop and Simonsen investigated nondeterministic higher-type cons-free programs and showed that the corresponding hierarchy surprisingly collapsed.

Finally, there is also a superficial similarity between the present paper and the field of implicit complexity theory (ICC): both aim to find machine-free characterizations of complexity classes, including NC and other intermediate classes between Logspace and Ptime [5, 13]. ICC characterizations are "function algebras" obtained by closing a set of primitive functions under a restricted number of functionals (such as branching and a various weak versions of primitive recursion), whereas our characterization uses a programming language with the ability to make arbitrary recursive definitions. On the other hand, our characterizations are not "implicit" in the strictest sense, since they refer to an external resource bound.

1.4 Organization of this paper

In Section 2 we present the cons-free programming language. We give cons-free programs an alternate operational semantics in Section 3 and discuss extensions of the base language in Section 4; these will come in handy in Section 5 when we compile cons-free programs and AuxPDAs into one another.

We finish Section 5 by deriving the "capturing" results referenced in Table 1. Finally, in Section 6, we present our absolute polynomial-time lower bounds. Readers familiar with cons-free programs can skip to Section 5, perhaps after skimming the short Section 3.

2 Cons-free programs

Consider a simple functional programming language defined over a "standard" set of string primitives; viz, `hd`, `tl`, `null`, `nil`, and `cons`. (These correspond to, respectively: isolating the first character of a string; removing that first character; a test of whether a string is empty; a constant naming the empty string; pushing a given character on a given string.) It is well known that such a language is not only Turing complete, but furthermore that Turing machines and programs can be compiled into one another with very little loss of running time (cf. Chapter 18 of [8]).

If we forbid any occurrence of `cons` in our programs, we obtain the *cons-free programs* of Jones [9]. This is a radically different model of computation. The resulting language is clearly not Turing-complete, but surprisingly computes exactly the LOGSPACE or PTIME relations depending on whether we consider its tail recursive or general recursive variant. However, we lose any relationship between program running time and Turing machine time; indeed, the program simulating a polynomial-time Turing machine may take exponential time in general.

In this section, we present the syntax and semantics of cons-free programs.

Definition 1. The *atomic types* are the type $2 = \{0, 1\}$ of booleans and the type $2^\star$ of binary strings of 0's and 1's of any finite length, including the empty string ε. A *product type* is an expression $\alpha_0 \times \cdots \times \alpha_{n-1}$, for some $n < \omega$, where each α_i is an atomic type. A *function type* is an expression $\sigma \to \tau$, where σ and τ are product types.

(Corner cases: when $n = 0$, we get the singleton type 1 as a product type. We recover the atomic types as product types when $n = 1$. We recover product types τ as function types $1 \to \tau$.)

The domain of a product type is the Cartesian product of the domains of its factors; the domain of a function type $\sigma \to \tau$ is the space of functions from the domain of σ to the domain of τ.

Definition 2. For each product type α and each function type ρ, let $\mathtt{Var}_\alpha$ be an infinite set of variables of type α and $\mathtt{RFsymb}_\rho$ be an infinite set of recursive function symbols of type ρ. A *term* is any expression which can be derived according to the inference rules in Figure 1.

Definition 3. A *program* consists of a list $(\mathtt{f}_0, \ldots, \mathtt{f}_k)$ of distinct recursive function symbols and, for each $0 \le i \le k$, a variable $\mathtt{x}_i$ of type α, and a term T_i of type β, where $\alpha \to \beta$ is the type of $\mathtt{f}_i$. A program is typically written

$$\begin{gathered} \mathtt{f}_0(\mathtt{x}_0) = T_0 \\ \vdots \\ \mathtt{f}_k(\mathtt{x}_k) = T_k \end{gathered}$$

The phrase $\mathtt{f}_i(\mathtt{x}_i) = T_i$ is the *recursive definition* of $\mathtt{f}_i$. The term $\mathtt{f}_0(\mathtt{x}_0)$ is the *head* of `p`.

We equip programs with a standard, call-by-value, environment-based big-step semantics. By a *value* we mean an element of the domain of some type, and by an *environment*, we mean a finite function mapping variables to values.

$$\frac{\mathtt{x} \in \mathtt{Var}_\alpha}{\mathtt{x} : \alpha} \qquad \frac{}{\mathtt{tt} : 2} \qquad \frac{}{\mathtt{ff} : 2} \qquad \frac{}{\mathtt{nil} : 2^\star}$$

$$\frac{T : 2^\star}{\mathtt{hd}(T) : 2} \qquad \frac{T : 2^\star}{\mathtt{tl}(T) : 2^\star} \qquad \frac{T : 2^\star}{\mathtt{null}(T) : 2}$$

$$\frac{T_0 : 2 \qquad T_1 : \alpha \qquad T_2 : \alpha}{\mathtt{if}\ T_0\ \mathtt{then}\ T_1\ \mathtt{else}\ T_2 : \alpha} \qquad \frac{T : \alpha \qquad \mathtt{f} \in \mathtt{RFsymb}_{\alpha \to \beta}}{\mathtt{f}(T) : \beta}$$

$$\frac{T_0 : \alpha_0 \qquad \dots \qquad T_{n-1} : \alpha_{n-1}}{(T_0, \dots, T_{n-1}) : \alpha_0 \times \dots \alpha_{n-1}} \qquad \frac{T : \alpha_0 \times \dots \times \alpha_{n-1} \qquad i < n}{T[i] : \alpha_i}$$

Figure 1: Program terms. The symbols α and β range over arbitrary product types, and symbols like α_i range over atomic types.

Definition 4. For a program $\mathtt{p}$, term T, value v of the same type as T, and environment ρ, we define the relation $\rho \vdash_{\mathtt{p}} T \to v$ according to the inference rules in Figure 2. We typically suppress the subscript $\mathtt{p}$ from $\vdash$ for legibility. The term $T^{\mathtt{f}}$ is the right-hand side of the recursive definition of $\mathtt{f}(\mathtt{x})$ in $\mathtt{p}$.

Definition 5. Given a program $\mathtt{p}$, define $[\![\mathtt{p}]\!]$ by $[\![\mathtt{p}]\!](x) = w$ iff there is a derivation of $[\mathtt{x}_0 = x] \vdash_{\mathtt{p}} \mathtt{f}_0(\mathtt{x}_0) \to w$, where $\mathtt{f}_0(\mathtt{x}_0)$ is the head of $\mathtt{p}$. (Since $\mathtt{p}$ is deterministic, $[\![\mathtt{p}]\!]$ is easily seen to be a partial function.)

Finally, we make an important definition of a kind of "triviality" for programs:

Definition 6. A program $\mathtt{p}$ is *explicit* if there is a partial ordering $<$ on its set of recursive function symbols, such that for any recursive function symbols $\mathtt{f}$ and $\mathtt{g}$, if $\mathtt{g}$ occurs in the recursive definition of $\mathtt{f}$ in $\mathtt{p}$, then $\mathtt{g} < \mathtt{f}$.

Explicit programs have no nontrivial recursion; indeed, we could transform the program into an equivalent one with no recursive calls whatsoever. An important fact about explicit programs is that they *run in constant time*; namely, there is a constant, independent to the input of the program, which bounds the size of all derivations of $[\mathtt{x}_0 = x] \vdash_{\mathtt{p}} \mathtt{f}_0(\mathtt{x}_0) \to w$.

Cons-free dialects. Not every paper on cons-free computation works with exactly the same programming language. Rather, they differ with regards to what the data is (strings or tuples of strings), whether the language is nondeterministic, and whether it includes higher types. Here, we allow the formation of (fixed-width) tuples of booleans and string data, but we restrict ourselves to deterministic, first-order programs. First, a word on each of these choices:

The cons-free languages presented in [10, 11] allow fixed-width tupling, but the language presented in [9] does not. For the purpose of computability, this distinction is inessential, but for the purpose of complexity, we need to form tuples to efficiently simulate AuxPDAs. (Indeed, it is not clear whether the result relating cons-free and AuxPDA running time (Theorem 7) holds for the "tuple-free dialect" of cons-free programs.)

$$\frac{}{\rho \vdash \mathtt{tt} \to 1} \qquad \frac{}{\rho \vdash \mathtt{ff} \to 0} \qquad \frac{}{\rho \vdash \mathtt{nil} \to \varepsilon}$$

$$\frac{\rho(\mathtt{x}) = v}{\rho \vdash \mathtt{x} \to v} \qquad \frac{\rho \vdash T \to w \qquad [\mathtt{x} \mapsto w] \vdash T^{\mathtt{f}} \to v}{\rho \vdash \mathtt{f}(\mathtt{x}) \to v}$$

$$((\exists v)\, \sigma v = w)\ \frac{T \to w}{\rho \vdash \mathtt{hd}(T) \to \sigma} \qquad ((\exists \sigma)\, \sigma v = w)\ \frac{\rho \vdash T \to w}{\rho \vdash \mathtt{tl}(T) \to v}$$

$$(w = \varepsilon)\ \frac{\rho \vdash T \to w}{\rho \vdash \mathtt{null}(T) \to 1} \qquad (w \neq \varepsilon)\ \frac{\rho \vdash T \to w}{\rho \vdash \mathtt{null}(T) \to 0}$$

$$\frac{\rho \vdash T_0 \to 1 \qquad \rho \vdash T_1 \to w}{\rho \vdash \mathtt{if}\ T_0\ \mathtt{then}\ T_1\ \mathtt{else}\ S \to w} \qquad \frac{\rho \vdash T_0 \to 0 \qquad \rho \vdash T_1 \to w}{\rho \vdash \mathtt{if}\ T_0\ \mathtt{then}\ S\ \mathtt{else}\ T_1 \to w}$$

$$\frac{\rho \vdash T_0 \to v_0 \qquad \ldots \qquad \rho \vdash T_{n-1} \to v_{n-1}}{\rho \vdash (T_0, \ldots, T_{n-1}) \to (v_0, \ldots, v_{n-1})} \qquad (i < n)\ \frac{T \to (v_0, \ldots, v_{n-1})}{\rho \vdash T[i] \to v_i}$$

Figure 2: Program Semantics. The symbol σ ranges over characters; S, T, T_0, T_1, T_n, etc. range over terms; and v and w range over values (strings or booleans, depending on the context).

The difference between deterministic and nondeterministic cons-free programs is an important one. While nondeterminism does not increase the computational power of first-order programs, it makes higher-order programs dramatically more expressive [11]. Even within the first-order case, the presence of nondeterminism might make programs more efficient. We fully expect that Theorem 7 can be extended to a similar relationship between nondeterministic cons-free programs and nondeterministic AuxPDAs, by essentially the same proof. If so, the difference between polynomial-time computation with deterministic and nondeterministic cons-free programs would correspond to the difference between LogDCFL and LogCFL. However, we cannot prove absolute polynomial lower bounds on nondeterministic cons-free running time.

Finally, higher-order cons-free programs can be used to capture complexity classes in the *deterministic exponential hierarchy*, which ramifies the class of elementary relations [10]. In light of the present paper, it is natural to ask if we can similarly characterize running time on higher-order cons-free programs in terms of a known resource. We leave this question open.

3 The one-stack implementation

It will also be useful to define an operational, *one-stack implementation* of cons-free programs. It can be regarded as an alternate semantics for programs, as we shall justify below. (The one-stack implementation is adapted from the *two-stack implementation* of recursive programs in an arbitrary signature by Moschovakis [12].)

Definition 7. A *term with parameters* is defined by extending the rules in Figure 1 by the additional rule that a value x of type α is a term of type α. A term with parameters is *variable-free* if it contains no variables.

Definition 8. Given a program $\texttt{p}$ with head $\texttt{f}_0(\texttt{x}_0)$, its *one-stack implementation* is defined as follows:

- Its *configurations* are stacks of the form $\begin{array}{|c|c|c|c|c|}\hline T_0 & T_1 & T_2 & \dots & T_r \\ \hline\end{array}$ where each T_i is a variable-free term with parameters. (We think of T_r as the top and T_0 as the bottom of the stack.)
- On input x, the *initial configuration* is the stack $\begin{array}{|c|}\hline \texttt{f}_0(x) \\ \hline\end{array}$.
- A *terminal configuration* is a stack of the form $\begin{array}{|c|}\hline v \\ \hline\end{array}$ for any value v.
- The *transition function* is defined by cases on the top one or two stack items as follows:

  1. $\begin{array}{|c|c|c|}\hline T_0 & \dots & \texttt{tt} \\ \hline\end{array} \longmapsto \begin{array}{|c|c|c|}\hline T_0 & \dots & 1 \\ \hline\end{array}$
  2. $\begin{array}{|c|c|c|}\hline T_0 & \dots & \texttt{ff} \\ \hline\end{array} \longmapsto \begin{array}{|c|c|c|}\hline T_0 & \dots & 0 \\ \hline\end{array}$
  3. $\begin{array}{|c|c|c|}\hline T_0 & \dots & \texttt{nil} \\ \hline\end{array} \longmapsto \begin{array}{|c|c|c|}\hline T_0 & \dots & \varepsilon \\ \hline\end{array}$
  4. $\begin{array}{|c|c|c|}\hline T_0 & \dots & \texttt{if } T_0 \texttt{ then } T_1 \texttt{ else } T_2 \\ \hline\end{array} \longmapsto \begin{array}{|c|c|c|c|}\hline T_0 & \dots & \texttt{if } T_0 \texttt{ then } T_1 \texttt{ else } T_2 & T_0 \\ \hline\end{array}$
  5. $\begin{array}{|c|c|c|c|}\hline T_0 & \dots & \texttt{if } T_0 \texttt{ then } T_1 \texttt{ else } T_2 & 1 \\ \hline\end{array} \longmapsto \begin{array}{|c|c|c|}\hline T_0 & \dots & T_1 \\ \hline\end{array}$
  6. $\begin{array}{|c|c|c|c|}\hline T_0 & \dots & \texttt{if } T_0 \texttt{ then } T_1 \texttt{ else } T_2 & 0 \\ \hline\end{array} \longmapsto \begin{array}{|c|c|c|}\hline T_0 & \dots & T_2 \\ \hline\end{array}$
  7. $\begin{array}{|c|c|c|}\hline T_0 & \dots & (\texttt{f}/\varphi)(t_0, \dots, t_{i-1}, T_i, \dots, T_{n-1}) \\ \hline\end{array} \longmapsto$
     $\begin{array}{|c|c|c|c|}\hline T_0 & \dots & (\texttt{f}/\varphi)(t_0, \dots, t_{i-1}, T_i, \dots, T_{n-1}) & T_i \\ \hline\end{array}$
  8. $\begin{array}{|c|c|c|c|}\hline T_0 & \dots & (\texttt{f}/\varphi)(t_0, \dots, t_{i-1}, T_i, \dots, T_{n-1}) & t_i \\ \hline\end{array} \longmapsto$
     $\begin{array}{|c|c|c|}\hline T_0 & \dots & (\texttt{f}/\varphi)(t_0, \dots, t_i, T_{i+1}, \dots, T_{n-1}) \\ \hline\end{array}$
  9. $\begin{array}{|c|c|c|}\hline T_0 & \dots & \texttt{f}(t) \\ \hline\end{array} \longmapsto \begin{array}{|c|c|c|}\hline T_0 & \dots & T^{\texttt{f}}(\texttt{x} \leftarrow t) \\ \hline\end{array}$
  10. $\begin{array}{|c|c|c|}\hline T_0 & \dots & \varphi(t_0, \dots, t_{n-1}) \\ \hline\end{array} \longmapsto \begin{array}{|c|c|c|}\hline T_0 & \dots & v \\ \hline\end{array}$
      where v is the result of applying φ to $\vec{t}$.

  The symbol φ is meant to range over $\{\texttt{hd}, \texttt{tl}, \texttt{null}, \texttt{nil}\}$, as well as the projection or indexing function $T \mapsto T[i]$, even though this is usually written postpositionally. (To be clear: in this case, the transition function is $\begin{array}{|c|c|c|}\hline T_0 & \dots & (t_0, \dots, t_{n-1})[i] \\ \hline\end{array} \longmapsto \begin{array}{|c|c|c|}\hline T_0 & \dots & t_i \\ \hline\end{array}$.)

  In case the top stack element T_r is a tuple of the form $(T_0, \dots, T_{n-1})$, we are in case 7, with $(\texttt{f}/\varphi)$ simply being regarded as "empty."

We write $\mapsto^\star$ to denote the transitive closure of the transition function $\mapsto$.

Remark 1. There are various configurations which are *stuck*, meaning they are not terminal, but there is no outgoing transition function.

Example. Consider the following program for computing the parity of the number of 1's in a string:

```
even(x) =if null(x) then tt else
          if hd(x) then odd(tl(x)) else even(tl(x))
 odd(x) =if null(x) then ff else
          if hd(x) then even(tl(x)) else odd(tl(x))
```

See Figure 3 for the derivation verifying $[\![\texttt{even}]\!](1) = 0$, which we have split into two for formatting purposes, followed by the computation on the 1-stack implementation. Observe that the one-stack simulation essentially performs a "depth-first traversal" through the derivation.

This motivates the following fundamental result relating the big-step semantics to the one-stack semantics in a rather tight sense: not only do they compute the same partial function, but the natural measures of "time complexity" in each model are identical up to a constant factor. We postpone its lengthy but straightforward proof to Appendix I.

Theorem 1. *For any program* $\texttt{p}$ *with head* $\texttt{f}_0(\texttt{x}_0)$ *and values* x *and* w*, there is a derivation of* $[\![\texttt{p}]\!](x) = w$ *if and only if there is a computation of the one-stack implementation of* $\texttt{p}$ *starting in* $\boxed{\texttt{f}_0(x)}$ *and ending in* $\boxed{w}$*. Moreover, the number of steps in the one-stack implementation and the size of the derivation are bounded within a constant factor (depending on the program, independent of the input) of each other.*

4 Conservative extensions of cons-free programs

In writing actual cons-free programs, we want the flexibility to use additional data types in addition to those explicitly given in the program syntax, but we do not want to increase the expressive power (or even efficiency) of our language—this is what we mean by a "conservative extension." This is a brief overview of well-known conservative extensions of cons-free programs. We indicate the justification that these extensions are, in fact, conservative, but do not give formal proofs.

For each extension, we must indicate how to implement it with previously-defined operations. We will estimate both the running time of these implementations as well as the number of calls to `hd` that they make. The reason for this is that in the next section, we capture complexity classes like NC via time-bounded cons-free programs, and their P-uniform versions via cons-free programs with a bounded number of calls to `hd`.

Some notational preliminaries: we identify any natural number with its set of predecessors; e.g., $k = \{0, 1, \ldots, k-1\}$, and by, e.g., $\Gamma + 1$ we mean the disjoint union of Γ and a singleton. By an *alphabet* we simply mean a finite set of size at least two.

Fixed finite sets. For any fixed finite set S, we can form the type of elements in S, in which we can name any element of S and compare any two elements of S. We implement this type simply as a fixed-width tuple of booleans.

Nested tupling. In the base cons-free language, the data is not closed under the tupling operation—we have atomic types, tuples of atomic types, and that's it. However, we can treat tuples of tuples as an extended cons-free data type, allowing us to form, e.g., tuples of finite sets, which are themselves encoded by tuples of booleans. In practice, this is done by "flattening." For example, suppose that A and B are finite sets encoded by 4-tuples and 5-tuples of booleans respectively. Then $A \times B$ can be encoded by 9-tuples of booleans. The projection (or "untupling") operations on nested tuple data may be implemented by composing tupling and projection operations on the base data.

Indices. Let us assume that we are working with programs with a single input string x of length n. It is not hard to see that we can extend the cons-free language by a type of *indices* of x, i.e., natural numbers less than or equal to n. We can simply encode these by suffixes of x of the

$$\dfrac{\dfrac{[\mathtt{x}\leftarrow 1]\vdash \mathtt{x}\to 1}{[\mathtt{x}\leftarrow 1]\vdash \mathtt{tl}(\mathtt{x})\to \varepsilon} \quad \dfrac{[\mathtt{x}\leftarrow \varepsilon]\vdash \mathtt{x}\to \varepsilon \quad \dfrac{(\varepsilon=\varepsilon)\,\dfrac{[\mathtt{x}\leftarrow \varepsilon]\vdash \mathtt{x}\to \varepsilon}{[\mathtt{x}\leftarrow \varepsilon]\vdash \mathtt{null}(\mathtt{x})\to 1} \quad [\mathtt{x}\leftarrow \varepsilon]\vdash \mathtt{ff}\to 0}{[\mathtt{x}\leftarrow \varepsilon]\vdash \mathtt{if}(\mathtt{null}(\mathtt{x}))\cdots\to 0}}{[\mathtt{x}\leftarrow \varepsilon]\vdash \mathtt{odd}(\mathtt{x})\to 0}}{\dfrac{[\mathtt{x}\leftarrow 1]\vdash \mathtt{odd}(\mathtt{tl}(\mathtt{x}))\to 0}{\ddots}}$$

$$\dfrac{[\mathtt{x}\leftarrow 1]\vdash \mathtt{x}\to 1 \quad \dfrac{(1\neq\varepsilon)\,\dfrac{[\mathtt{x}\leftarrow 1]\vdash \mathtt{x}\to 1}{[\mathtt{x}\leftarrow 1]\vdash \mathtt{null}(\mathtt{x})\to 0} \quad \dfrac{\dfrac{[\mathtt{x}\leftarrow 1]\vdash \mathtt{x}\to 1}{[\mathtt{x}\leftarrow 1]\vdash \mathtt{hd}(\mathtt{x})\to 1} \quad \dfrac{\ddots}{[\mathtt{x}\leftarrow 1]\vdash \mathtt{odd}(\mathtt{tl}(\mathtt{x}))\to 0}}{[\mathtt{x}\leftarrow 1]\vdash \mathtt{if}(\mathtt{hd}(\mathtt{x}))\cdots\to 0}}{[\mathtt{x}\leftarrow 1]\vdash \mathtt{if}(\mathtt{null}(\mathtt{x}))\cdots\to 0}}{[\mathtt{x}\leftarrow 1]\vdash \mathtt{even}(\mathtt{x})\to 0}$$

$\boxed{\texttt{even}(1)}$

$\boxed{\texttt{if null}(1)\texttt{ then tt else if hd}(1)\texttt{ then odd}(\texttt{tl}(1))\texttt{ else even}(\texttt{tl}(1))}$

$\boxed{\texttt{if null}(1)\texttt{ then tt else if hd}(1)\texttt{ then odd}(\texttt{tl}(1))\texttt{ else even}(\texttt{tl}(1))}\boxed{\texttt{null}(1)}$

$\boxed{\texttt{if null}(1)\texttt{ then tt else if hd}(1)\texttt{ then odd}(\texttt{tl}(1))\texttt{ else even}(\texttt{tl}(1))}\boxed{0}$

$\boxed{\texttt{if hd}(1)\texttt{ then odd}(\texttt{tl}(1))\texttt{ else even}(\texttt{tl}(1))}$

$\boxed{\texttt{if hd}(1)\texttt{ then odd}(\texttt{tl}(1))\texttt{ else even}(\texttt{tl}(1))}\boxed{\texttt{hd}(1)}$

$\boxed{\texttt{if hd}(1)\texttt{ then odd}(\texttt{tl}(1))\texttt{ else even}(\texttt{tl}(1))}\boxed{1}$

$\boxed{\texttt{odd}(\texttt{tl}(1))}$

$\boxed{\texttt{odd}(\texttt{tl}(1))}\boxed{\texttt{tl}(1)}$

$\boxed{\texttt{odd}(\texttt{tl}(1))}\boxed{\varepsilon}$

$\boxed{\texttt{odd}(\varepsilon)}$

$\boxed{\texttt{if null}(\varepsilon)\texttt{ then ff else if hd}(\varepsilon)\texttt{ then even}(\texttt{tl}(\varepsilon))\texttt{ else odd}(\texttt{tl}(\varepsilon))}$

$\boxed{\texttt{if null}(\varepsilon)\texttt{ then ff else if hd}(\varepsilon)\texttt{ then even}(\texttt{tl}(\varepsilon))\texttt{ else odd}(\texttt{tl}(\varepsilon))}\boxed{\texttt{null}(\varepsilon)}$

$\boxed{\texttt{if null}(\varepsilon)\texttt{ then ff else if hd}(\varepsilon)\texttt{ then even}(\texttt{tl}(\varepsilon))\texttt{ else odd}(\texttt{tl}(\varepsilon))}\boxed{1}$

$\boxed{\texttt{ff}}$

$\boxed{0}$

Figure 3: The proof tree and 1-stack simulation verifying $[\![\texttt{even}]\!](1) = 0$
.

appropriate length. Under our identification of a natural number with its set of predecessors, the domain of the type of indices is exactly $n+1$.

So, let `i` be an index variable. We implement $\mathtt{i}=0$ by `null(i)`. We implement $\mathtt{i}-1$ by `tl(i)`. We can implement any comparison operator ($\mathtt{i}=\mathtt{j}$, $\mathtt{i}<\mathtt{j}$, or $\mathtt{i}\leq\mathtt{j}$) by a recursive program `f(i, j)` which makes some explicit test if one of the variables is zero, otherwise calls $\mathtt{f}(\mathtt{i}-1,\mathtt{j}-1)$. We can implement the maximum index `max` (whose denotation is n) by `x`, where `x` is a variable always bound to the input string. We can compute a "truncated successor" (i.e., $i\mapsto\min\{i+1,n\}$) by `if eq(i, x) then x else S(i, x)`, where $\mathtt{S}(\mathtt{i},\mathtt{j})=$ `if` $\mathtt{eq}(\mathtt{i},\mathtt{j}-1)$ `then j else` $\mathtt{S}(\mathtt{i},\mathtt{j}-1)$.

Each of the "index primitives" `eq`, $+1$, -1, and `max` can be computed in polynomial time from the cons-free primitives `tl`, `nil`, and `null`; none requires any calls to `hd`. However, given an index, we also want to find the *bit* of the input string at the given index. Namely, if $x=x_nx_{n-1}\ldots x_1$, then $\mathtt{bit}(x,i)=x_i$. (Notice that we index from n to 1 and that $\mathtt{bit}(x,0)$ is undefined.)

Since any index variable `i` is assumed to be bound to a suffix of the input string, `bit(i)` can simply be implemented by `hd(i)`. This is clearly a constant-time implementation that makes exactly one call to `hd`.

Counting modules. For any fixed a,b, we may form the type of all numbers less than $a(n+1)^b$, where n is the length of the input string x. In other words, we get access to numbers bounded by a *fixed polynomial in the length of the input.* This is effected by considering tuples of the form $(y,x_0,\ldots,x_{b-1})$ where y is a member of the finite set a, and $x_0,\ldots,x_{b-1}$ are suffixes of the input string x; this tuple encodes the number $y(n+1)^b+\sum_{i<b}|x_i|(n+1)^i$. Such types are called *counting modules*, after [9].

For example, if $a=1$ and $b=2$, then we encode elements of $(n+1)^2$ by a pair of suffixes (x_0,x_1) of x. If the variable `x` is bound to the input value, then `(x, x)` names the maximum element. Given a variable `c` of type $2^\star\times2^\star$, we can test whether it encodes the zero element of the counting module by

$$\mathtt{if\ null(c[0])\ then\ null(c[1])\ else\ ff}.$$

We can decrement `c` by taking the predecessor `P(c)`, defined by

$$\mathtt{if\ c[0]=0\ then\ (x,tl(c[1]))\ else\ (tl(c[0]),c[1])}.$$

Notice that each of these operations is implemented by a constant-time program that does not make any calls to `hd`. This is clearly extensible to arbitrary a and b.

Using a zero-test, `max`, and `P` as primitives, we can compare counting module numbers and compute (truncated) successor `S` in polynomial time, as we did for indices. These allow us to implement the counting module arithmetic operations (suitably truncated) as in Table 2.

Each of these operations runs in polynomial time from comparison, `P`, and `S`, and hence can be implemented in polynomial time from the base cons-free primitives. Moreover, none of them makes any call to `hd`.

Short strings. Given a fixed alphabet Δ of size d and natural numbers a,b, we can form the type of strings over Δ of length at most $b\log(n+1)+a$, where n is the length of the input string x. This is because there is a bijection

$$\Delta^{b\log(n+1)+a}\simeq(d^a)(n+1)^{b\log d}$$

Operation	Implementation
subtraction ($(c,d) \mapsto \max\{c-d, 0\}$)	$\texttt{subtr(c,d)} = \texttt{if d} = 0 \texttt{ then c else P(subtr(c,P(d)))}$
addition ($(c,d) \mapsto \min\{c+d, a(n+1)^b\}$)	$\texttt{add(c,d)} = \texttt{if d} = 0 \texttt{ then c else S(add(c,P(d)))}$
multiplication ($(c,d) \mapsto \min\{cd, a(n+1)^b\}$)	$\texttt{mult(c,d)} = \texttt{if d} = 0 \texttt{ then } 0 \texttt{ else add(mult(c,P(d)),d)}$
integer quotient ($(c,d) \mapsto \lfloor \frac{c}{d} \rfloor$)	$\texttt{quot(c,d)} = \texttt{if c} = 0 \texttt{ then } 0 \texttt{ else S(quot(subtr(c,d),d))}$
remainder ($(c,d) \mapsto c \mod d$)	$\texttt{rem(c,d)} = \texttt{if c} < \texttt{d then c else rem(subtr(c,d),d)}$

Table 2: Arithmetic in counting modules

$\texttt{hd}^\circ\texttt{(s)}$	$\texttt{rem(s},\|\Delta\|)$
$\texttt{tl}^\circ\texttt{(s)}$	$\texttt{quot(s},\|\Delta\|)$
$\texttt{null}^\circ\texttt{(s)}$	$\texttt{s} = 0$
$\texttt{nil}^\circ$	0
$\texttt{cons}^\circ(\sigma,\texttt{s})$	$\texttt{add(mult(s},\|\Delta\|),\sigma)$

Figure 4: The definition of *short-string primitives* by counting module operations.

given by

$$c_0c_1 \dots c_{\ell-1} \mapsto \sum_{i<\ell} c_i d^i,$$

where $\ell = b \log(n+1) + a$. Therefore, we can identify the set of short strings with the domain of some counting module, which we have already constructed as an extended data type.

Moreover, we can implement the short-string primitives using the operations on counting modules as in Figure 4. Notice that these short-string primitives $\texttt{hd}^\circ$, $\texttt{tl}^\circ$, $\texttt{null}^\circ$, $\texttt{nil}^\circ$, and $\texttt{cons}^\circ$ differ from the base primitives `hd`, `tl`, `null`, `nil`, and `cons`. Notice moreover that $\texttt{cons}^\circ$ is "truncated," in the sense that it only returns a meaningful answer when the string `s` has length strictly less than the maximum length.

Since these short-string primitives can all be implemented in constant time using counting module arithmetic, they can all be implemented in polynomial time from the base primitives, without cons, of course. Moreover, none of them requires any calls to the cons-free primitive `hd`.

This last point bears repeating: all five string primitives, $\texttt{hd}^\circ$, $\texttt{tl}^\circ$, $\texttt{null}^\circ$, $\texttt{nil}^\circ$, and $\texttt{cons}^\circ$, when applied to a type of *very short strings* relative to the input x, can be implemented using only *three* of the base string primitives, namely `tl`, `null`, and `nil`!

Conclusion. By "extended cons-free data types," let us mean fixed finite sets, indices of the input string, counting modules, short strings, and tuples formed of all of these, with the corresponding primitive operations. The conclusion of all the preceding discussion is that usage of extended cons-free data types does not change the expressive power of cons-free programs, and moreover that it cannot increase the efficiency by more than a polynomial factor.

Theorem 2. *For any cons-free program* `p` *that runs in time $t(n)$ on inputs of length n and uses extended cons-free data types, there is a program* `q` *in the base cons-free language such that $[\![\texttt{p}]\!] = [\![\texttt{q}]\!]$, and a polynomial π such that* `q` *runs in time $t(n) \cdot \pi(n)$.*

5 AuxPDAs and programs

An AuxPDA is a finite automaton with a read-only input tape augmented with both a read/write work tape and a pushdown store. In this section we prove our main results (cf. Table 1) by compiling cons-free programs and AuxPDAs into one another, moreover with little loss of complexity in either direction.

In subsection 5.1 we compile AuxPDAs into programs by using recursion to simulate the stack, making liberal use of the extended data types from the previous section. In subsection 5.2 we compile programs into AuxPDAs using the one-stack implementation as an intermediary. But first we define AuxPDAs themselves along with some important associated notions. As we only consider logspace AuxPDAs, that is, AuxPDAs the length of whose work tape is bounded by a logarithm in the length of the input, we view the work tape as finite with special characters marking the endpoints.

Definition 9. A *(deterministic, logspace) AuxPDA* is a tuple consisting of a finite set Q of states, an *input alphabet* Σ, a *tape alphabet* Δ, a *stack alphabet* Γ, a *tape bound* $n \mapsto a\log(n) + b$, and a *transition function*

$$\tau : Q \times \Sigma_{\bowtie} \times \Delta_{\bowtie} \times (\Gamma + 1) \to Q \times 3 \times \Delta_{\bowtie} \times 3 \times (\Gamma + 1),$$

where $\Sigma_{\bowtie}$ and $\Delta_{\bowtie}$ are obtained from Σ and Δ by adding the *left* and *right* endpoint characters $\triangleright, \triangleleft$.

The idea is that on a given state, character read by the input head, character read by the tape head, and character at the top of the stack (or if the stack is empty, hence $\Gamma + 1$), the AuxPDA moves to a new state, moves the input head in one of 3 directions (left, right, stay put), writes a character under the tape head, moves the tape head in one of 3 directions, and either pushes a character in Γ on the stack, or pops the stack (hence $\Gamma + 1$).

The transition function is forbidden from changing any endpoint character, overwriting any non-endpoint character with an endpoint character, moving a head right of a right endpoint, or moving a head left of a left endpoint.

Furthermore, an element $q_0 \in Q$ is distinguished as *initial.* There are an additional two distinct states $q_{acc}, q_{rej} \in Q$, the *accept* and *reject* state respectively. A distinct character in Δ is designated the *blank* character.

Definition 10. A *configuration on input x* of an AuxPDA consists of the state, the position of the input head, the contents of the work tape (split into the halves to the left and to the right of the work head), and the contents of the stack. This is an object of type $Q \times (|x| + 2) \times \Delta^\star \times \Delta^\star \times \Gamma^\star$.

More precisely, the string to the left of the head is in reverse order, the character under the head is grouped with the string to the right, and we strip the endpoint characters from the configuration. For example, in Figure 5, the work tape would be encoded by the pair of strings $(\lambda_0\lambda_1 \ldots \lambda_\ell, \rho_0\rho_1 \ldots \rho_r)$, with ρ_0 being the character under the head.

A *surface configuration* of an AuxPDA consists of the state, the position of the input head, the character read by the input head, the contents of the work tape (split by the work head) and the top of the stack (or a special character if the stack is empty); in other words, an object of type $Q \times (|x| + 2) \times \Sigma_{\bowtie} \times \Delta^\star \times \Delta^\star \times (\Gamma + 1)$. Given a configuration σ and a fixed input, its associated surface configuration is simply called its *surface.*

Initially, the input head and the work head are at the left endpoints of their respective tapes. On input x, the input tape is initialized to $\triangleright x \triangleleft$, and the work tape is initialized to the string $\triangleright w \triangleleft$,

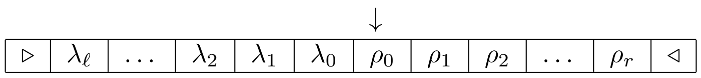

Figure 5: The work tape of an AuxPDA, depicting strings to the left and right of the input head.

where w is a string of $a \log |x| + b$ blanks. A configuration is *terminal* if its stack is empty and it activates a pop. Given a nonterminal configuration of an AuxPDA on some input, the transition function uniquely defines a *next* configuration as described in Definition 9.

Definition 11. Given an AuxPDA $\mathcal{A}$ with input alphabet Σ and a string $x \in \Sigma^\star$, the *computation of* $\mathcal{A}$ *on input* x is the orbit of $\mathcal{A}$'s initial configuration on input x under the "next" operation defined by the transition function of $\mathcal{A}$. The length of this orbit (in fact, the length of any orbit) is an ordinal in $\omega + 1$.

We say that $\mathcal{A}$ *accepts* x in case the orbit is finite and the state of the final configuration is q_{acc}, and $\mathcal{A}$ *rejects* x in case the orbit is finite and the state of the final configuration is q_{rej}.

We will typically write a generic computation as $\{\sigma_i\}_{i<\alpha}$, where α is its length and σ_i is the i-th configuration in the computation. We think on the length of a computation as its "running time" (cf. Definition 12 below).

Remark 2. Notice that the transition function does not induce a corresponding "next" operation on surface configurations. If the current surface configuration activates a "push" operation on the stack, then the next surface configuration *is* a function of the current surface configuration plus the input x, but if it activates a "pop," then even given the input x, we do not know what the new top of the stack will be. However, we can always determine the next surface configuration from the current surface configuration, the input, and the second element on the stack.

In practice, this means that surface configurations encode quite a bit. Suppose that $(\sigma_i)_{i<\alpha}$ is the computation of an AuxPDA on some input x. Let g_i and s_i be the stack and the surface of the configuration σ_i respectively. Notice that g_{i+1} is obtained from g_i by either a push or pop operation. For every $i < \alpha$, let $f(i) > i$ be the least number such that g_i is not a prefix of $g_{f(i)}$ (i.e., the first time the stack "dips below" its value at stage i). Of course, f may be undefined for some values of i if the computation is non-terminating. If the computation is terminating then $\alpha < \omega$, and for any i such that g_i is the empty stack, $f(i) = \alpha$. (In particular, $f(0) = \alpha$ in any terminating configuration.)

We claim that for every i such that $f(i)$ is defined, s_j for $i \leq j < f(i)$ depends only on s_i plus the fixed input x. This is by induction on j: if s_{j-1} activates a push, then it determines s_j, and if it activates a pop, then g_i must be a *proper* prefix g_{j-1}; hence, the second element of g_{j-1} must have been pushed after stage i (and thus dependent only on s_i by induction). Therefore, we can determine s_j.

Notice also that if s_i activates a pop operation, $f(i) = i + 1$; if s_i activates a push operation and $f(i)$ is defined, then $i < i + 1 < f(i + 1) < f(i)$, $g_{f(i+1)} = g_i$, and $f(f(i + 1)) = f(i)$. The computation converges iff $f(0)$ is defined, in which case $f(0) - 1$ is the terminal configuration.

Definition 12. Given an AuxPDA $\mathcal{A}$ with input alphabet Σ and a language $X \subseteq \Sigma^\star$, we say that $\mathcal{A}$ *decides* X in case for each string $x \in \Sigma^\star$,

- $x \in X \implies \mathcal{A}$ accepts x, and
- $x \notin X \implies \mathcal{A}$ rejects x.

Definition 13. Given an AuxPDA $\mathcal{A}$ and a function $t : \omega \to \omega$, we say $\mathcal{A}$ *runs in time* $t(n)$ in case for any $n \in \omega$ and $x \in \Sigma^n$, the computation of $\mathcal{A}$ on input x has length at most $t(n)$. If in addition $\mathcal{A}$ decides a language X, we say that $\mathcal{A}$ *decides* X *in time* $t(n)$.

5.1 Cons-free simulations of AuxPDAs

In this subsection, fix a logspace AuxPDA $\mathcal{A}$ and let a, b be positive integers such that $a \log n + b$ is an upper bound for the work space of $\mathcal{A}$ on inputs of length n. Notice that the domain and codomain of the transition function are tuples of finite types (and hence finite types themselves), and can therefore be encoded by a fixed-length tuple of booleans.

Recall Definition 6 of explicit programs. Even though that definition was made for cons-free programs, one could talk about explicit programs in the extended cons-free primitives, or some subset thereof. The definition is the same, and the remark that explicit programs run in constant time (where each program primitive is unit cost) is still valid.

Lemma 1. *There is an explicit cons-free program* `trans` *such that* $[\![\texttt{trans}]\!](r) = \tau(r)$ *for every* $r \in Q \times \Sigma_{\bowtie} \times \Delta_{\bowtie} \times (\Gamma + 1)$*. Moreover,* `trans` *requires no calls to any of the cons-free primitives* `hd`*,* `tl`*,* `nil`*, or* `null`*.*

Proof. In fact, the lemma holds for *any* function from a finite type to a finite type. Since any finite type can be encoded by a fixed-width tuple of booleans, any function whose domain and codomain are finite types may be computed by a single term built up using tuple indexing (to obtain the bits of the input), if-then-else statements, and boolean constants. The resulting program is clearly explicit, runs in constant time, and does not call the primitives. □

The crucial observation is that, given an input x, the set of surface configurations, while not a finite type, can be identified as an extended cons-free data type depending on the length of x. This is because each surface configuration is a tuple consisting of:

- a state (which is an element of a fixed finite set),
- the position of the input head (which can be encoded by the appropriate suffix of x plus a finite amount of information indicating if the head is on the left or right endpoint character),
- the character under the input head (which is an element of a fixed finite set),
- the contents of the work tape (which can be encoded by two short strings, namely the string to the left of and to the right of the head on the work tape), and
- the top of the stack—or some special character marking that it's empty (which is an element of some fixed finite set).

Why did we choose to include both the position and the character under the input head as part of the surface configuration, when the cons-free data encoding the former—a suffix of the input—already contains the latter? The answer is that this way, we can extract an input to the transition function from the surface configuration *without* calling `hd`, as we now show. (For the next several lemmas, recall the definition of the *short-string primitives* in Figure 4.)

Lemma 2. *There is a cons-free program* `trin` *such that, for any surface configuration* s*,* $[\![\texttt{trin}]\!](s)$ *encodes the input to the transition function implicit in* s*. Moreover,* `trin` *is explicit in the extended cons-free primitives and requires no calls to* `hd`*.*

Proof. Given a surface configuration $s = (q, p, \sigma, w_\lambda, w_\rho, \gamma)$, the corresponding input to the transition function is $r = (q, \sigma, \texttt{hd}^\circ(w_\rho), \gamma)$, since the string w_ρ contains the character under the head of the work tape.

Observe that r is clearly computable from s by splitting it up into its constituent coordinates, applying $\texttt{hd}^\circ$ to one of them, and combining them again. This clearly describes a program which is explicit in the extended cons-free primitives and requires no calls to `hd`. □

Now we establish a series of "simulation primitives."

Lemma 3. *There is an explicit cons-free program* `out` *such that for any surface configuration s, $[\![\texttt{out}]\!](s)$ is true, false, or diverges, depending on whether the state in s is q_{acc}, q_{rej}, or neither. Moreover* `out` *is explicit and does not contain any calls to the primitives.*

Proof. The program `out` simply extracts the initial segment of s encoding the state, compares it to the constants encoding q_{acc} and q_{rej}, and returns `tt`, `ff`, or diverges accordingly. □

Lemma 4. *There is a cons-free program* `push` *such that for any surface configuration s, $[\![\texttt{push}]\!](s)$ is true or false depending on whether s activates a push on the stack. Moreover* `push` *is explicit in the extended cons-free primitives and does not contain any calls to* `hd`*.*

Proof. The program `push` applies `trin` to s to extract the input r to the transition function encoded by s, then applies `trans` to r to extract the instructions $\tau(r)$ of the transition function. Then it extracts the segment of $\tau(r)$ corresponding to the instructions for the stack, which is an element of $\Gamma + 1$; the 1 means "pop," and each $\gamma \in \Gamma$ means "push γ." We simply compare this part of $\tau(r)$ to the constant denoting a "pop" and negate the result. □

Lemma 5. *There is a cons-free program* `next` *such that for any surface configuration s on input x, $[\![\texttt{next}]\!](x, s)$ returns the next surface configuration if s activates a push. Moreover,* `next` *is explicit in the cons-free primitives. It makes zero calls to* `hd` *unless the head of the input tape of $\mathcal{A}$ moves, in which case it makes exactly one call to* `hd`*.*

Proof. As in the proof of the previous lemma, the program `next` extracts the output $\tau(r)$ of the transition function from the input r encoded by the surface configuration s explicitly, making no calls to `hd`. The state and top-of-the-stack of the next surface configuration are both contained in $\tau(r)$, the latter because s activates a push.

As for the work tape, $\tau(r)$ contains the character δ written under the head of the work tape and the dynamics of the head of the work tape (stay put, left, right). The new contents of the work tape (w'_λ, w'_ρ) are obtained from the old contents (w_λ, w_ρ) and the character δ by

- $w'_\lambda = w_\lambda$ and $w'_\rho = \texttt{cons}^\circ(\delta, \texttt{tl}^\circ(w_\rho))$ if the head stays put,
- $w'_\lambda = \texttt{tl}^\circ(w_\lambda)$ and $w'_\rho = \texttt{cons}^\circ(\texttt{hd}^\circ(w_\lambda), \texttt{cons}^\circ(\delta, \texttt{tl}(w_\rho)))$ if the head moves left, and
- $w'_\lambda = \texttt{cons}^\circ(\delta, w_\lambda)$ and $w'_\rho = \texttt{tl}^\circ(w_\rho)$ if the head moves right.

Notice that none of these require any calls to `hd`.

The new position of the input head can be calculated from the old position by doing nothing, applying `S`, or applying `P`—i.e., a single index primitive—depending on whether the head of the input tape stays put, moves right, or moves left. Notice that `S` and `P` do not require any calls to hd. If the head of the input tape stays put, the new character under the input string is simply the old character under the input string.

If the head of the input tape moves right or left, then we have to call the bit primitive *once* on the input string x and the new position to calculate the new character under the head. This one call to bit requires one call to hd. □

For the next lemma, recall from Remark 2 that in case s activates a pop, the next surface configuration is computable from the current surface configuration, the input string, and the surface of the previous most recent configuration the top of whose stack is about to be uncovered by the pop operation.

Lemma 6. *There is a cons-free program* follow *which satisfies the following property. Suppose there are configurations σ, τ, and ρ with surfaces s, t, and r, such that*

- *the stack of σ extends the stack of τ by one character, and*
- *$\sigma \mapsto \rho$ in the transition relation.*

Then $[\![\texttt{follow}]\!](x, s, t) = r$. Moreover, follow *is explicit in the extended cons-free primitives. It makes zero calls to* hd *unless the head of the input tape of $\mathcal{A}$ moves, in which case it makes exactly one call to* hd*.*

Proof. The proof is identical to the proof of Lemma 5 except that the new top of the stack is extracted from the top-of-the-stack character in t, rather than from the output of the transition function as in next. This extraction is explicit and requires no calls to the primitives. □

Finally,

Lemma 7. *There is a cons-free program* init *such that for any string x, $[\![\texttt{init}]\!](x)$ encodes the surface of the initial configuration of $\mathcal{A}$ on input x. Moreover,* init *can be computed in logarithmic time from the extended cons-free primitives using no calls to* hd*.*

Proof. The initial position is 0, the initial character under the input head is $\triangleright$, the initial state is q_0, the initial state to the left of the work head is ε, the initial state to the right is $\triangleright y \triangleleft$ (where y is a string of blanks). The short string y can be constructed using $a\log(n) + b$ calls to $\texttt{cons}^\circ$. Nothing here uses hd. □

To simulate AuxPDAs by cons-free programs, we mimic Glück's pure functional simulation of pushdown automata (AuxPDAs without the work tape).

Definition 14. Given init, out, push, next, and follow as primitives, define $\texttt{sim}_\mathcal{A}$ by:

$$\texttt{sim}_\mathcal{A}(\texttt{x}) = \texttt{out}(\texttt{g}(\texttt{x}, \texttt{init}(\texttt{x})))$$

$$\texttt{g}(\texttt{x}, \texttt{s}) = \begin{cases} \texttt{g}(\texttt{x}, \texttt{follow}(\texttt{x}, \texttt{s}, \texttt{g}(\texttt{x}, \texttt{next}(\texttt{x}, \texttt{s})))) & \texttt{if push(s)} \\ \texttt{s} & \texttt{otherwise} \end{cases}$$

Theorem 3. *For any input x, $\mathcal{A}$ accepts, rejects, or diverges on input x in case $[\![\texttt{sim}_\mathcal{A}]\!](x) = 1$, $[\![\texttt{sim}_\mathcal{A}]\!](x) = 0$, or $[\![\texttt{sim}_\mathcal{A}]\!](x)$ diverges respectively.*

Proof. In this proof we will slightly abuse notation, writing, e.g., $\mathtt{g}(x, s) = t$ for

$$[\mathtt{x} = x, \mathtt{s} = s] \vdash_{\mathtt{sim}_{\mathcal{A}}} \mathtt{g}(\mathtt{x}, \mathtt{s}) \to t.$$

Fix x, and let $(\sigma_i)_{i<\alpha}$ be the computation of $\mathcal{A}$ on input x. Define s_i, g_i, and the partial function f as in Remark 2. Notice that if s_i activates a push operation, $s_{i+1} = \mathtt{next}(x, s_i)$; additionally, $s_{f(i)-1}$ activates a pop operation, and $\mathtt{follow}(x, s_i, s_{f(i+1)-1}) = s_{f(i+1)}$.

We claim that for each $i < \alpha$ in the domain of f, $\mathtt{g}(x, s_i) = s_{f(i)-1}$. The proof proceeds by induction on $f(i) - i$. If $f(i) - i = 1$, then σ_i activates a pop operation, and $\mathtt{g}(x, s_i) = s_i$. Otherwise, $i < i + 1 < f(i + 1) < f(i)$ and $f(i) = f(f(i + 1))$; by induction, $\mathtt{g}(x, s_{i+1}) = s_{f(i+1)-1}$, $\mathtt{g}(x, s_{f(i+1)}) = s_{f(f(i+1))-1} = s_{f(i)-1}$, and

$$\begin{aligned}
\mathtt{g}(x, s_i) &= \mathtt{g}(x, \mathtt{follow}(x, s_i, \mathtt{g}(x, \mathtt{next}(x, s_i))))\\
&= \mathtt{g}(x, \mathtt{follow}(x, s_i, \mathtt{g}(x, s_{i+1})))\\
&= \mathtt{g}(x, \mathtt{follow}(x, s_i, s_{f(i+1)-1}))\\
&= \mathtt{g}(x, s_{f(i+1)})\\
&= s_{f(i)-1}.
\end{aligned}$$

Conversely, we claim that if $\mathtt{g}(x, s_i) = v$, then $f(i)$ converges and $v = s_{f(i)-1}$. The proof proceeds by induction on the size of the derivation of $\mathtt{g}(x, s_i) = v$. If s_i activates a pop, then $v = s_i$, $f(i) = i + 1$, and we're done. Otherwise, $\mathtt{next}(x, s_i) = s_{i+1}$, $\mathtt{g}(x, s_{i+1}) = s_{f(i+1)-1}$ by induction, $\mathtt{g}(x, s_{f(i+1)}) = s_{f(i)-1}$ by induction, and hence $\mathtt{g}(x, s_i) = s_{f(i)-1}$.

Finally, if $\mathcal{A}$ halts on x, then $\mathtt{g}(x, s_0) = s_{f(0)-1}$. Then $\mathcal{A}$ respectively accepts or respectively iff $s_{f(0)-1}$ is respectively an accepting or rejecting state iff $\mathtt{out}(s_{f(0)-1}$ is respectively 1 or 0. If $\mathcal{A}$ does not halt on x, then 0 is not in the domain of f, so $\mathtt{g}(x, s_0)$, and hence $\mathtt{f}(x)$, does not converge. □

Next, we estimate the running time of $[\![\mathtt{sim}_{\mathcal{A}}]\!](x)$ assuming that `init`, `out`, `push`, `next`, and `follow` are all unit-cost operations. In this case, the running time of $\mathtt{sim}_{\mathcal{A}}$ is simply bounded above by an affine function of the running time of $\mathcal{A}$.

Theorem 4. *Suppose that $\mathcal{A}$ converges on input x. Then the running time of $\mathtt{sim}_{\mathcal{A}}$ on x from the simulation primitives is bounded above by $3 + 5t$, where t is the running time of $\mathcal{A}$ on x.*

Proof. Define $(\sigma_i)_{i<\alpha}$, s_i, g_i, and f as above. Then the running time of $\mathcal{A}$ on x is $f(0)$. We claim that the number of steps needed to compute $\mathtt{g}(x, s_i)$ is at most $5(f(i) - i)$. Then the running time of $\mathtt{f}(x)$, which consists of making the recursive call $\mathtt{out}(\mathtt{g}(x, \mathtt{init}(x)))$, is at most $3 + 5f(0)$, which is what we wanted to prove. The proof is again by induction on $f(i) - i$.

In the base case, $f(i) - i = 1$, in which case the running time $\mathtt{g}(x, s_i) = s_i$ is at most 5. (One step to rewrite the definition of $\mathtt{g}$, one step to extract the hypothesis of the conditional, one to evaluate it to "false," and a final step to rewrite the conditional to the "else" branch.) Otherwise $f(i+1)-(i+1) < f(i)-i$ and $f(i)-f(i+1) < f(i)-i$; by induction, the running time of $\mathtt{g}(x, s_{i+1})$ is at most $5(f(i + 1) - (i + 1))$ and the running time of $\mathtt{g}(x, s_{f(i+1)})$ is at most $5(f(i) - f(i + 1))$.

The running time of $\mathtt{g}(x, \mathtt{next}(x, s_i))$ is at most $1 + 5(f(i + 1) - (i + 1))$, the running time of

$$\mathtt{follow}(x, s_i, \mathtt{g}(x, \mathtt{next}(x, s_i)))$$

is at most $2 + 5(f(i + 1) - (i + 1))$, and the running time of

$$\mathtt{g}(x, \mathtt{follow}(x, s_i, \mathtt{g}(x, \mathtt{next}(x, s_i))))$$

is at most $5(f(i)-f(i+1))+2+5(f(i+1)-(i+1))$, which is equal to $2+5(f(i)-(i+1))$. Finally, the computation of $\mathtt{g}(x,s_i)$ consists of evaluating an if-then-else statement whose "if"-branch has size 1 and whose "then"-branch has size at most the above; its computation therefore has size at most $4+5(f(i)-(i+1))$, which is less than $5(f(i)-i)$. $\square$

Finally, we estimate the running time of $\mathtt{sim}_{\mathcal{A}}$ from the basic cons-free primitives, keeping particular attention to the number of calls to `hd`.

Theorem 5. *For any AuxPDA $\mathcal{A}$ there is a polynomial p and a constant κ, such that if $\mathcal{A}$ runs within time $t(n) \in \Omega(\log(n))$ on inputs of length n, then $\mathtt{sim}_{\mathcal{A}}$ runs in time $t(n)\cdot p(n)+\kappa$. Moreover, if $\mathcal{A}$ moves the input head at most $t(n)$ times, then $\mathtt{sim}_{\mathcal{A}}$ makes at most $t(n)$ calls to* `hd`.

Proof. By Lemmata 3 through 7, each of the simulation primitives `out`, `push`, `next`, and `follow` can be implemented by programs which are explicit in the extended cons-free primitives, and `init`, which is called only once, only requires logarithmic time in the extended cons-free primitives. Therefore, by Theorem 4, there are constants C and D such that $\mathtt{sim}_{\mathcal{A}}$ can be implemented to run in time $C\cdot t(n)+D$ from the extended cons-free primitives. By Theorem 2, there is a polynomial q such that $\mathtt{sim}_{\mathcal{A}}$ can be implemented to run in time $(C\cdot t(n)+D)\cdot q(n)$; simply take $p=C\cdot q$ and $\kappa = D\cdot q$.

For the second part of the theorem (estimating the number of calls to `hd`), we mimic the proof of Theorem 4. Fix x, and let σ_i, s_i, g_i, and f be defined as in that proof. Let $h(i)$ be the number of times $\mathcal{A}$ moves its input head while moving from configuration σ_i to configuration $\sigma_{f(i)-1}$. Then $h(0)$ is the number of times $\mathcal{A}$ moves its input head during the entire computation; we want to show that $h(0)$ is an upper bound for the number of calls $\mathtt{sim}_{\mathcal{A}}$ makes to `hd`.

We show, by induction on $f(i)-i$, that $h(i)$ is an upper bound for the number of calls to `hd` made during the computation of $\mathtt{g}(x,s_i)$. In the base case, where $f(i)=i+1$, $h(i)=0$ (as $\sigma_i=\sigma_{f(i)-1}$) and $\mathtt{g}(x,s_i)$ makes zero calls to `hd`.

Otherwise, $f(i+1)-(i+1)<f(i)-i$ and $f(f(i+1))-f(i+1)=f(i)-f(i+1)<f(i)-i$. Split up the computation $\sigma_i\to\cdots\to\sigma_{f(i)-1}$ into the four fragments

$$\sigma_i \to \sigma_{i+1} \tag{1}$$
$$\sigma_{i+1}\to\cdots\to\sigma_{f(i+1)-1} \tag{2}$$
$$\sigma_{f(i+1)-1}\to\sigma_{f(i+1)} \tag{3}$$
$$\sigma_{f(i+1)}\to\cdots\to\sigma_{f(i)-1} \tag{4}$$

Fragments (2) and (4) contain $h(i+1)$ and $h(f(i+1))$ movements of the input head respectively. The number of movements of the input head in the entire fragment—i.e., $h(i)$—is $h(i+1)+h(f(i+1))+dh$, where dh is the sum of the number of moves of the input head from $\sigma_i\to\sigma_{i+1}$ and from $\sigma_{f(i+1)-1}\to\sigma_{f(i+1)}$. (So $dh\in\{0,1,2\}$.)

Consider the computation

$$\begin{aligned}
\mathtt{g}(x,s_i) &= \mathtt{g}(x,\mathtt{follow}(x,s_i,\mathtt{g}(x,\mathtt{next}(x,s_i)))) && (5)\\
&= \mathtt{g}(x,\mathtt{follow}(x,s_i,\mathtt{g}(x,s_{i+1}))) && (6)\\
&= \mathtt{g}(x,\mathtt{follow}(x,s_i,s_{f(i+1)-1})) && (7)\\
&= \mathtt{g}(x,s_{f(i+1)}) && (8)\\
&= s_{f(i)-1}. && (9)
\end{aligned}$$

By Lemma 5, the number of calls to `hd` made while evaluating `next`(x, s_i) is exactly the number of moves of the input head in $\sigma_i \to \sigma_{i+1}$. By Lemma 6, the number of calls to `hd` made while evaluating `follow`$(x, s_i, s_{f(i+1)-1})$ is exactly the number of moves of the input head in $\sigma_{f(i+1)-1} \to \sigma_{f(i+1)}$. Hence, the total number of calls to head made between lines (5) and (6) and lines (7) and (8), is dh.

By induction, the number of calls to `hd` made while evaluating `g`(x, s_{i+1}) is at most $h(i+1)$ and the number of calls to `hd` made while evaluating `g`$(x, s_{f(i+1)})$ is at most $h(f(i+1))$. These bound the number of calls to `hd` between lines (6) and (7) and (8) to (9) respectively.

Hence, the number of calls to `hd` made between lines (5) and (9) is at most $h(i+1) + h(f(i+1)) + dh = h(i)$. Since it takes zero calls to `hd` to move from the left-hand to the right-hand side of line (5), we are done.

□

5.2 AuxPDA simulation of cons-free programs

We build an AuxPDA $\mathcal{A}_{\mathtt{p}}$ for any given cons-free program `p` by modifying the one-stack implementation of `p`. Notice that the one-stack implementation of a program is not in and of itself an AuxPDA, as the size of the stack alphabet depends on the input. The key observation, however, is that we can encode each stack item in the one-stack implementation of `p` by $O(\log n)$ characters in some finite alphabet (where n is the length of the input) and use standard programming techniques to efficiently implement its transition function.

Theorem 6. *For every program* `p`*, there is an AuxPDA* $\mathcal{A}_{\mathtt{p}}$*, such that if* `p` *accepts, rejects, or diverges on input* x *within time* $t(n)$*, then* $\mathcal{A}_{\mathtt{p}}$ *accepts, rejects, or diverges on input* x *within time* $O(t(n) \cdot n)$*. Moreover, if* `p` *uses at most* $h(n)$ *calls to* `hd`*, then* $\mathcal{A}_{\mathtt{p}}$ *moves its input head at most* $h(n) \cdot n$ *times.*

Proof. Consider the set of terms that could potentially appear in the stack of the one-stack implementation of program `p` on input x (call these *stack items*). Each stack item is obtained by taking some term which occurs in `p` and substituting its variables with particular values. The set of possible values are tuples consisting of booleans and suffixes of the input string x, since a cons-free program can only form new string values by repeated applications of `tl`.

We claim that each stack item can be encoded in space $O(\log n)$. This is because each suffix of x can be encoded by its length, a $\log n$-bit string, the maximum length of any tuple of values is a constant in `p`, and the maximum length of any term is also a constant in `p`. Fix a stack alphabet for $\mathcal{A}_{\mathtt{p}}$ of size at least two, and a work tape large enough to store at least two stack items, plus some constant amount of overhead.

The idea is that $\mathcal{A}_{\mathtt{p}}$ stores the stack of the one-stack implementation on *its* stack—but the depth of the stack might increase by an $O(\log n)$ factor, because the AuxPDA's stack alphabet is finite. Operationally, $\mathcal{A}_{\mathtt{p}}$ simulates the one-stack implementation step-by-step. To obtain the next stack from the current one, $\mathcal{A}_{\mathtt{p}}$ loads the top one or two stack items onto its work tape, determines which action is appropriate according to Definition 8, and modifies its stack accordingly. By simulating the stack of the one-stack implementation, $\mathcal{A}_{\mathtt{p}}$ correctly computes the output value of `p` on input x.

The running time of $\mathcal{A}_{\mathtt{p}}$ is bounded above by the running time of the one-stack implementation times an upper bound of the time needed to simulate a single step. In so doing, $\mathcal{A}_{\mathtt{p}}$ performs subroutines such as: parsing a term into subterms, term substitution, and modifying binary values representing suffixes. We do not get into the weeds of the implementation of these algorithms, which are certainly nontrivial, but "common knowledge."

What is the running time of these operations? Examine all the cases of the transition function of the one-stack machine in Definition 8. Notice that in cases 1-8, the right-hand side of the transition function is completely determined from the left-hand side, and depends on neither $\mathtt{p}$ nor x. Each of these operations can certainly be implemented in time polynomial in the length of a single stack item, and space linear in the length of a single stack item—these are $\log^{O(1)} n$ and $O(\log n)$ respectively. They require no movement of the input head.

To simulate case 9 of the transition function, $\mathcal{A}_{\mathtt{p}}$ needs to obtain $T^{\mathtt{f}}$ from $\mathtt{f}$, and then substitute the variable in $T^{\mathtt{f}}$ with a value. This procedure depends on $\mathtt{p}$ but not on x, and thus can be effected without moving the input head, as $\mathtt{p}$ can be hard-wired in the finite control of $\mathcal{A}_{\mathtt{p}}$. Again, it takes time polynomial in, and space linear in, the length of a single stack item.

Finally consider case 10. In this case, we are either combining a list of values into a single tuple, extracting a coordinate from a tuple, decrementing a binary numeral (in case $\varphi = \mathtt{tl}$; each string value is encoded by its length, and $\mathtt{tl}$ of that value is the length minus 1), testing whether a binary numeral is equal to zero (in case $\varphi = \mathtt{null}$), or finding the bit of x at the index specified by a particular binary numeral (in case $\varphi = \mathtt{hd}$).

In all but the last of these sub-cases, the procedure depends only on the top stack item, and can be effected in space and time $O(\log n)$. They are independent of $\mathtt{p}$ and x and require no moves of the input head. In the last sub-case—finding the bit of x at a particular index—we need space $O(\log n)$, and at most n moves of the input head, to locate that bit.

Therefore, the time required to simulate a single step of the one-stack implementation is $O(n)$—in fact, $\log^{O(1)} n$ except for the last sub-case of 10—and hence the total time taken by $\mathcal{A}_{\mathtt{p}}$ is bounded above by $O(t(n) \cdot n)$. Moreover, the number of moves of the input head is at most n times the number of times the last sub-case comes up, which is exactly the number of calls to $\mathtt{hd}$. This proves the theorem.

□

5.3 Results and discussion

Since, for every $k \geq 1$ the class $2^{O(\log^k(n))}$ is closed under multiplication by polynomials in n, Theorems 5 and 6 have the following immediate consequence:

Theorem 7. *For any natural number $k \geq 1$, the set of problems decidable by a cons-free program in time $2^{O(\log^k(n))}$ is identical to the set of problems decided by an AuxPDA in time $2^{O(\log^k(n))}$.*

Moreover, the set of problems decidable by a cons-free program using $2^{O(\log^k(n))}$ calls to $\mathtt{hd}$ *is identical to the set of problems decided by an AuxPDA making at most $2^{O(\log^k(n))}$ moves of the input head.*

As an immediate consequence of Theorem 7, we obtain our novel characterizations of LogDCFL, NC, and PUNC. Taking the case $k = 1$,

Theorem 8. *The set* LogDCFL *is exactly the set of all problems decidable by cons-free programs in polynomial time.*

And, taking the union over all $k \geq 1$,

Theorem 9. *The set* NC *is exactly the set of all problems decidable by a cons-free program in quasi-polynomial time (i.e., time $2^{\log^{O(1)}(n)}$).*

Moreover, recall that the *P-uniform* versions of all of these classes are exactly captured by counting the number of moves of the input head rather than the total time of the AuxPDA. In particular,

Theorem 10. *The set* PUNC *is exactly the set of all problems decidable by cons-free programs making a quasi-polynomial number of calls to* `hd`*.*

Calling `hd` can reasonably be identified with "accessing the bits of the input;" without this primitive, we would not be able to distinguish between any two strings of the same length. Hence, cons-free programs give us another natural way of interpreting the slogan of Allender [1] that P-uniform circuit classes are obtained by restricting access to the input on machine models of computation.

Remark 3. By a theorem of Tserunyan [19], for programs operating over arbitrary data, it suffices to count the total number of calls to the primitives instead of counting running time, up to a polynomial factor (cf. [12] chapter 3B). Hence we can identify the set of problems decidable by an AuxPDA in time $2^{O(\log^k(n))}$ with the set of problems decidable by a cons-free programs that makes $2^{O(\log^k(n))}$ total calls to `hd`, `tl`, `null`, and `nil` combined. In that sense, the difference between each complexity class and its P-uniform version comes down to the difference between counting all the cons-free primitives, versus only `hd`.

Remark 4. We believe that we can recover an analogous correspondence between nondeterministic cons-free programs and nondeterministic AuxPDAs, with no essential changes in the proof. Given a nondeterministic AuxPDA, the definition of `trans` would be nondeterministic, and the rest of the simulation would stay the same. Given a nondeterministic program, its one-stack simulation would need to have nondeterminism in the "recursive call" case of the transition function. Then the proofs should go through just as before.

If this were true, we would get a cons-free characterization of the complexity class SAC^k, which corresponds to time $2^{O(\log^k(n))}$ on a nondeterministic AuxPDA, as well as its P-uniform version obtained by bounding moves of the input head.

6 Absolute polynomial lower bounds

Proving lower bounds on logspace-AuxPDA time, or related resources like logspace-uniform circuit depth or logspace-ATM time, is hard. There is certainly no general hierarchy theorem; indeed, even separating exponential from polynomial time on a logspace AuxPDA suffices to separate Ptime from Logspace. But weaker separations that would not resolve any hard open questions are still beyond the state of the art, such as showing strictness for the time-$O(n^c)$ hierarchy for logspace AuxPDAs, or the depth-$O(\log^c(n))$ hierarchy for uniform circuit classes.

In this section, we show that there are problems in Ptime that cannot be solved by a cons-free program in time $O(n^c)$, for any $c < \omega$. This result does not extend to corresponding lower bounds on, say, AuxPDA time: even though cons-free time and AuxPDA time are related within a polynomial factor, that polynomial may depend on the problem in question. We choose to view this as a feature, not a bug: cons-free running time is a "Goldilocks resource," strong enough to recover the same characterizations of complexity classes as AuxPDA time, but weak enough to admit polynomial lower bounds.

Our proof mimics Blum's abstract version of Rabin's theorem [14, 4]. Suppose we have an acceptable indexing $\{\varphi_e\}$ of the partial recursive functions and any complexity measure $\{\Phi_e\}$ satisfying the Blum axioms for this indexing. Then Rabin's theorem says that for any recursive function t, there is a recursive relation X such that any program e computing X must satisfy $(\forall^\infty x)\, t(x) < \Phi_e(x)$ (i.e., take time at least $t(x)$ almost everywhere).

In the present context, we will consider indexings of cons-free programs, and understand "Φ" as "cons-free running time." We will show that we can mimic the proof of Rabin's theorem when $t(x)$ is $|x|^c$, which gives us polynomial lower bounds. The technical adaptations required are a *subrecursive* formulation of Blum complexity due to Alton [2], a delayed diagonalization, and memoization.

In the next two definitions, let *Progs* be the set of cons-free programs.

Definition 15. Let the relation $\Phi \subseteq \mathit{Progs} \times 2^\star \times \omega$ be defined by $\Phi(\mathtt{p}, x) \leq t$ iff there exists some w and a derivation of $[\![\mathtt{p}]\!](x) = w$ of size at most t. We will typically write $\Phi_{\mathtt{p}}(x)$ for $\Phi(\mathtt{p}, x)$.

In the recursion-theoretic context, we have a computable universal partial function $(e, x) \mapsto [\![e]\!](x)$. The universal function of an indexed complexity class is typically not in that class (or else we would get a contradiction by constructing the diagonal relation). The indexing of Ptime by cons-free programs is no exception: it does not have a universal function in Ptime.

However, in recursion-theoretic applications, it often suffices to use the following weakening of the universal function, which we call the *simulation partial function* (or *Sim* following Alton). It takes a resource bound as a parameter in addition to a program code and input, and is only required to be correct if the given program on the given input uses less than the given resource.

Definition 16. Let the partial function $\mathit{Sim} : \mathit{Progs} \times 2^\star \times \omega \rightharpoonup D$, where D is the set of all cons-free data, be defined by $\mathit{Sim}(\mathtt{p}, x, t)$ be $[\![\mathtt{p}]\!](x)$ if $\Phi_{\mathtt{p}}(x) \leq t$, and undefined otherwise.

Encodings of programs. We want to reinterpret Φ and *Sim* as a relation and function respectively on binary strings. In particular, we need an encoding of programs in *Progs* as elements of $2^\star$. We fix any "reasonable" encoding of programs by strings. A program can be thought of as a list of terms, and each term can be thought of as a tree; there are standard ways to encode such objects by strings so that the list indexing and subtree relations remain efficiently computable. In particular, we demand of our encoding that

- it is polynomial-time decidable whether a given string encodes a program,
- occurrences of terms of the program can be indexed by binary strings of length polynomial in the length of the encoding, and
- the indices of immediate sub-occurrences of an occurrence of a term can be computed from the encoding of the whole program and the index of the term in polynomial time.

See Appendix II for an elaboration of this construction.

Let the *width* of a variable be the length of its corresponding product type. Define the width of a program to be the sum, over all occurrences of variables in the program, of the width of that particular occurrence. The one unusual property we demand is that

- the width of any program is bounded by a polynomial in the length of the encoding of the program.

One could imagine reasonable encodings of programs such that the width of the program was bounded by a *log* in the length of the encoding. For example, imagine the program $\mathtt{f}_0(\mathtt{x}) = \mathtt{x}[0]$, where $\mathtt{x}$ is a variable of a product type, and the length of its type is 100,000.

In the encoding of the program, we record the types of $\mathtt{f}_0$ and $\mathtt{x}$ as well as the definition of $\mathtt{f}_0(\mathtt{x})$. But we can record the types of $\mathtt{f}_0$ and $\mathtt{x}$ efficiently using a binary encoding of 100,000, and we can encode the definition of $\mathtt{f}_0(\mathtt{x})$ using a very short term. Nothing forces us to use 100,000 characters to encode this program.

However, we can easily ensure this condition is met by taking any other encoding and padding by the width of the program, in unary. (Notice that this does not affect the other desiderata of the encoding.) We will use this property in the proof of Lemma 8 below.

Definition 17. For any $c < \omega$, let the relation $\Phi^c \subseteq 2^\star \times 2^\star$ be defined as $\Phi^c(e, x) \iff e$ encodes a program $\mathtt{p}$ and $\Phi_{\mathtt{p}}(x) \leq |x|^c$. Similarly, let $\mathit{Sim}^c(e, x) = \mathit{Sim}(\mathtt{p}, x, |x|^c)$, where e encodes $\mathtt{p}$.

(For a string e that does not encode a program, let $\Phi^c(e, x)$ and $\mathit{Sim}^c(e, x)$ be undefined.)

Lemma 8. *For each $c < \omega$, the functions $\Phi^c(e, x)$ and $\mathit{Sim}^c(e, x)$ are computable in time polynomial in $(|e|, |x|)$.*

Proof. Suppose $\mathtt{p}$ is the program encoded by e. We compute both functions in the same way, namely by simulating the one-stack implementation of $\mathtt{p}$ on the input x with a prescribed time bound of $|x|^c$. For Φ we return 1 or 0 depending on whether the computation halts in time, and for Sim, we return the output of the computation if it halts in time, and diverge otherwise.

The total time taken is bounded by $|x|^c$ times the time needed to compute each stack step. Each stack step takes time polynomial in the maximum length of the encoding of a single stack item. Each stack item is obtained by taking a term T occurring in the program $\mathtt{p}$ and substituting some of its subterms by suffixes of x.

Each suffix of x can be encoded by a string of length logarithmic (and hence polynomial) in $|x|$. The number of subterms of T substituted is at most the width of $\mathtt{p}$, hence bounded by a polynomial in the length of e. Finally, we can encode the form of the substitution by the index of the term T and the indices of all of the occurrences of its subterms that are being replaced. Each such index has length bounded by a polynomial in the length of e, and the number of such indices is again at most the width of e. $\square$

Definition 18. For fixed $c < \omega$, define the relations $R \subseteq 2^\star \times 2^\star$ and the function $f : 2^\star \to 2$ by mutual induction as follows:

- $R(e, x) = (\forall y)\ |y| < \log |x| \longrightarrow \big(\Phi^c(e, y) \longrightarrow f(y) = \mathit{Sim}^c(e, y)\big)$,
- $f(x) = \begin{cases} \neg \mathit{Sim}^c(e_0, x) & \text{if } e_0 = (\mu e)\ |e| < \log |x| \wedge e \in \mathit{Progs} \wedge R(e, x) \wedge \Phi^c(e, x) \\ 0 & \text{if there is no such } e \end{cases}$

Here μ means minimization with respect to any total order $\prec \subseteq 2^\star \times 2^\star$ that extends the partial order given by string length; for example, the lexicographically least shortest string.

Remark 5. Note that f is relational, i.e., Boolean-valued. We will write f_c for f when we want to make the dependence on c explicit. Also note that $\prec$ is a well-ordering with order type ω. In particular, the number of $\prec$-predecessors of any given element are finite.

Now, we show that f_c is a cons-free computable function, but one that requires time at least n^c almost everywhere. We prove first the lower and then the upper bound.

Theorem 11. *For any $c < \omega$ and cons-free program $\mathtt{p}$ that computes $f = f_c$, $|x|^c < \Phi_p(x)$ for all but finitely many x.*

Proof. Suppose by contradiction that there is a cons-free program $\mathtt{p}$ and a $c < \omega$ such that $[\![\mathtt{p}]\!] \simeq f$ and for infinitely many x, $\Phi_{\mathtt{p}}(x) \leq |x|^c$. Let e_0 be an encoding of $\mathtt{p}$; then $(\exists^\infty x)\, \Phi^c(e_0, x)$. Moreover, for any y, if $\Phi^c(e_0, y)$, then $\mathit{Sim}^c(e_0, y)$ returns the correct value $[\![\mathtt{p}]\!](y)$, which is $f(y)$. This implies that $(\forall x)\, R(e_0, x)$.

Let $R(\cdot, x)$ abbreviate $\{e \in 2^\star : R(e, x)\}$. Then $R(\cdot, x)$ only depends on $|x|$; therefore, let us write $R(\cdot, n)$ to mean $R(\cdot, x)$, for any string x of length n. Notice that $R(\cdot, n)$ is monotone decreasing in $|x|$, that is:

$$R(\cdot, 0) \supseteq R(\cdot, 1) \supseteq R(\cdot, 2) \supseteq \dots.$$

Let $E = \{e \in \mathit{Progs} : e \prec e_0\}$ be the set of $\prec$-predecessors of e_0. Then E is finite and

$$R(\cdot, 0) \cap E \supseteq R(\cdot, 1) \cap E \supseteq R(\cdot, 2) \cap E \supseteq \dots;$$

therefore, this sequence is eventually constant, being a decreasing sequence of finite sets.

Since $(\exists^\infty x)\, \Phi^c(e_0, x)$, we can choose some x_0 such that $\Phi^c(e_0, x_0)$, $|e_0| < \log |x_0|$, and $R(\cdot, n) \cap E = R(\cdot, m) \cap E$ for any $m, n \geq |x_0|$. Moreover, $R(e_0, x_0)$ (since $(\forall x)\, R(e_0, x)$). We finish the proof by splitting into cases.

- Suppose that e_0 were the $\prec$-least $e \in \mathit{Progs}$ such that $|e| < \log |x_0|$, $R(e, x_0)$, and $\Phi^c(e, x_0)$. Then $f(x_0) = \neg \mathit{Sim}^c(e_0, x)$, but since $\Phi^c(e_0, x)$ and e_0 computes f, $f(x_0) = \neg f(x_0)$, a contradiction.

- Suppose that there were a strictly lesser $e_1 \prec e_0$ such that $e_1 \in \mathit{Progs}$, $|e_1| < \log |x_0|$, $R(e_1, x_0)$, and $\Phi^c(e_1, x_0)$. Then $f(x_0) = \neg \mathit{Sim}^c(e_1, x_0)$, and hence the implication $\Phi^c(e_1, x_0) \to f(x_0) = \mathit{Sim}^c(e_1, x_0)$ is false. Therefore, for sufficiently large x, $R(e_1, x)$ is false. Since $e_1 \in E$, $e_1 \in R(\cdot, |x_0|)$, but $e_1 \notin R(\cdot, n)$ for sufficiently large n, this contradicts the fact that $R(\cdot, n)$ stabilizes for $n \geq |x_0|$.

□

Finally, we show the upper bound.

Theorem 12. *For any $c < \omega$, $f = f_c$ can be computed in* Ptime *(and hence by some cons-free program).*

Proof. Fix $c < \omega$. Let d be a constant such that testing $\Phi^c(e, x)$, $\mathit{Sim}^c(e, x)$, and $e \in \mathit{Progs}$ are all bounded by $O((|e| + |x|)^d)$. We first show that we can compute the list $\langle f(x) : |x| \leq n \rangle$ in time $2^{O(n)}$.

We can construct $\langle f(x) : |x| \leq 1 \rangle$ in time constant in n. Suppose $n \geq 2$ (so that $\log n \leq n - 1$) and we are given the list $\langle f(x) : |x| \leq n - 1 \rangle$. How much additional time does it take to construct $\langle f(x) : |x| \leq n \rangle$?

For each y of length at most $\log n$, we can retrieve $f(y)$ in unit time (since $\log n \leq n - 1$) and, for each string e of length at most n, we can compute $\Phi^c(e, y)$ and $\mathit{Sim}^c(e, y)$ in time $O(n^d)$. Therefore, we can compute $\Phi^c(e, y) \to f(y) = \mathit{Sim}^c(e, y)$ in time $O(n^d)$. Computing $R(e, x)$ for any single $|x| = n$ requires evaluating this expression for all y of length at most $\log n$, which takes time at most $O(n^{d+1})$.

Hence, for $x \in 2^n$ and a string e of length at most n, evaluating $e \in \mathit{Progs} \wedge R(e, x) \wedge \Phi^c(e, x)$ takes time at most $O(n^{d+1})$. Searching for the least witness (if it exists) over all e of length at most $\log n$ then takes time at most $O(n^{d+2})$. Therefore, computing $f(x)$ for $x \in 2^n$ takes time at most

$O(n^{d+2})$, and the amount of *additional* time needed to extend $\langle f(x) : |x| \leq n-1 \rangle$ by $\langle f(x) : |x| = n \rangle$ is $O(n^{d+2})2^n$, which is $2^{O(n)}$. Since $\sum_{i<n} 2^{O(i)}$ is still $2^{O(n)}$, we can construct $\langle f(x) : |x| \leq n \rangle$ all in time $2^{O(n)}$.

Next, we show that we can compute $R(e, x)$ in time $(|e| + |x|)^{O(1)}$. First construct $\langle f(y) : |y| < \log |x| \rangle$ in time $2^{O(\log |x|)} = |x|^{O(1)}$. Then for each y of length at most $\log |x|$, perform a calculation $(\Phi^c(e, y) \to f(y) = \mathit{Sim}^c(e, y))$ that takes time at most $(|e| + |x|)^{O(1)}$; multiplied by $|x|$ values of y, this is still $(|e| + |x|)^{O(1)}$.

Finally, to compute $f(x)$, we need to compute at most four subroutines that take time $(|e| + |x|)^{O(1)}$ for each e of length at most $\log |x|$. This clearly takes time at most $|x|^{O(1)}$, which is what we needed to show. $\square$

Therefore, for each $c < \omega$, we have a relation f_c which is decidable in PTIME (and hence computable by a cons-free program), but which requires time at least n^c for inputs of length n almost everywhere.

7 Discussion

We have shown how to compile between deterministic cons-free programs and deterministic AuxPDAs with compiling functions that roughly preserve running time in either model; hence, we recover characterizations of several complexity classes in terms of cons-free running time. We also exhibit polynomial lower bounds on the cons-free time needed to solve certain problems in PTIME; while modest, analogous lower bounds on corresponding models of computation are not known. Straightforward next steps in this line of work include:

- generalizing the present results to the non-deterministic case,
- attempting to show that the cons-free time-$O(n^c)$ hierarchy is in fact strict, and
- seeing whether other natural resources (ATM space, circuit size, AuxPDA stack depth etc.) are captured by cons-free resources. In particular,
- we wonder whether we can compile cons-free programs into *multihead 2-way deterministic pushdown automata* (roughly, AuxPDAs without a logspace work tape). If we can control the running time and number of heads of the target PDA by the running time of the source program, the results of the previous section would follow from Ibarra [7].

More generally, one of the significant aspects about our work is that it connects two areas (complexity theory and programming language theory) with different standards of proof. That is to say that arguments involving machine and circuit models of computation are (necessarily!) less formal than arguments involving programming languages with a carefully specified syntax and semantics. (This is certainly not an indictment of either discipline, but rather a sociological observation.)

By developing a program of structural complexity theory that uses programming languages as its standard model of computation, we open the possibility of witnessing nontrivial relationships between complexity classes via formally presented program transformations. At the very least, this broadens the accessibility of these results. At the best, it might help strengthen or generalize them. For example, a program transformation that transforms a non-deterministic cons-free program accepting a set X into a non-deterministic cons-free program accepting $\neg X$ might generalize the closure of both NL and LogCFL under complementation, if the transformation roughly preserved running time and tail recursive structure.

Acknowledgements: We are greatly indebted to Amir ben-Amram, Neil Jones, Pekka Orponen, W. Larry Ruzzo, and Eric Allender for answering many questions and freely sharing their wisdom with us.

8 Appendix I: Equivalence of two semantics

Following is the proof of Theorem 1, showing that the big-step semantics and one-stack operational semantics are extensionally equivalent, and their natural associated measures of time complexity are equivalent up to a constant factor depending only on the program.

Proof. For an environment ρ binding the free variables of T, let T^ρ denote the variable-free term obtained by substituting each variable by its ρ-image.

First we show that if there is a derivation of $\rho \vdash T \to v$ of size s, there is a computation $\boxed{T^\rho} \mapsto^\star \boxed{v}$ of length at most $(N+2)s$, where N is the maximum length of any tuple in `p`. We proceed by induction on s.

If $s = 1$, then T is either one of the constants `tt`, `ff`, or `nil`, or a variable `x`. If $T \equiv \texttt{tt}$, then $\boxed{\texttt{tt}} \mapsto \boxed{1}$ is our desired computation; similarly for the other constants. If T is a variable then $\boxed{T^\rho}$ is identical to $\boxed{\rho(\texttt{x})}$; our desired computation is simply the trivial computation $\boxed{T^\rho}$ of length 1.

If $s > 1$ then we break up into cases depending on the form of T.

- If $T \equiv \texttt{if } T_0 \texttt{ then } T_1 \texttt{ else } T_2$ and $\rho \vdash T_0 \to 1$, then there are some s_0 and s_1 such that there is a derivation of $\rho \vdash T_0 \to 1$ of size s_0, of $\rho \vdash T_1 \to v$ of size s_1, and $s = s_0 + s_1 + 1$. By induction there are computations $\boxed{T_0} \mapsto^\star \boxed{1}$ and $\boxed{T_1} \mapsto^\star \boxed{v}$ of lengths at most $(N+2)s_0$ and $(N+2)s_1$ respectively. The following computation has length at most $(N+2)s_0 + (N+2)s_1 + 1 < (N+2)s$.

$$\begin{array}{l}
\boxed{\texttt{if } T_0 \texttt{ then } T_1 \texttt{ else } T_2} \\
\boxed{\texttt{if } T_0 \texttt{ then } T_2 \texttt{ else } T_2}\boxed{T_0} \\
\vdots \\
\boxed{\texttt{if } T_0 \texttt{ then } T_2 \texttt{ else } T_2}\boxed{1} \\
\boxed{T_1} \\
\vdots \\
\boxed{v}
\end{array}$$

  The case $\rho \vdash T_0 \to 0$ is similar, replacing T_1 by T_2.

- If $T \equiv \varphi(T_0, \dots, T_{n-1})$, then by induction there are s_i, v_i for $i < n$ such that $1 + \sum_{i<n} s_i = s$, there is a derivation $\rho \vdash T_i \to v_i$ of size s_i, and $\varphi(v_0, \dots, v_{n-1}) = v$. The following is a computation $\boxed{T} \mapsto^\star \boxed{v}$. It can be broken up into n blocks, block i having length at most $(N+2)s_i + 1$, plus two more configurations. The total length is thus

$$2 + \sum_{i<n} ((N+2)s_i + 1) \le (N+2)\big(1 + \sum_{i<n} s_i\big) \le 2s.$$

$$\begin{array}{l}
\boxed{\varphi(T_0,\dots,T_{n-2},T_{n-1})} \\
\boxed{\varphi(T_0,\dots,T_{n-2},T_{n-1})}\boxed{T_0} \\
\vdots \\
\boxed{\varphi(T_0,\dots,T_{n-2},T_{n-1})}\boxed{v_0} \\
\boxed{\varphi(v_0,\dots,T_{n-2},T_{n-1})} \\
\vdots \\
\boxed{\varphi(v_0,\dots,v_{n-2},T_{n-1})} \\
\boxed{\varphi(v_0,\dots,v_{n-2},T_{n-1})}\boxed{T_{n-1}} \\
\vdots \\
\boxed{\varphi(v_0,\dots,v_{n-2},T_{n-1})}\boxed{v_{n-1}} \\
\boxed{\varphi(v_0,\dots,v_{n-1})} \\
\boxed{v}
\end{array}$$

- The case $T \equiv (T_0,\dots,T_{n-1})$ is similar, except that there is no symbol φ, and the last box $\boxed{v}$ is absent from the computation, hence its length is even smaller.

- If $T \equiv \mathtt{f}(T_0,\dots,T_{n-1})$, then by induction, there are s_i for $i \le n$ and v_i for $i < n$ such that $1+\sum_{i\le n} s_i = s$, for each $i < n$ there is a derivation $\rho \vdash T_i \to v_i$ of size at most s_i, and there is a derivation $[\mathtt{x} = (v_0,\dots,v_{n-1})] \vdash T^{\mathtt{f}} \to v$ of size at most s_n. The following is a computation $\boxed{T} \mapsto^\star \boxed{v}$ of length at most

$$\sum_{i\le n}((N+2)s_i+1) \le (N+2)\big(\sum_{i\le n} s_i\big) + N + 1 \le (N+2)\big(1+\sum_{i\le n} s_i\big) = (N+2)s$$

$$\boxed{\mathtt{f}(T_0,\ldots,T_{n-2},T_{n-1})}$$
$$\boxed{\mathtt{f}(T_0,\ldots,T_{n-2},T_{n-1}) \mid T_0}$$
$$\vdots$$
$$\boxed{\mathtt{f}(T_0,\ldots,T_{n-2},T_{n-1}) \mid v_0}$$
$$\boxed{\mathtt{f}(v_0,\ldots,T_{n-2},T_{n-1})}$$
$$\vdots$$
$$\boxed{\mathtt{f}(v_0,\ldots,v_{n-2},T_{n-1})}$$
$$\boxed{\mathtt{f}(v_0,\ldots,v_{n-2},T_{n-1}) \mid T_{n-1}}$$
$$\vdots$$
$$\boxed{\mathtt{f}(v_0,\ldots,v_{n-2},T_{n-1}) \mid v_{n-1}}$$
$$\boxed{\mathtt{f}(v_0,\ldots,v_{n-1})}$$
$$\boxed{T^{\mathtt{f}}(\mathtt{x} \leftarrow (v_0,\ldots,v_{n-1}))}$$
$$\vdots$$
$$\boxed{v}$$

In the other direction, we show that for any environment ρ binding the free variables of T, if there is a computation $\boxed{T^\rho} \mapsto^\star \boxed{v}$ of length at most ℓ, then there is a derivation of $\rho \vdash T \to v$ of size at most ℓ. The proof is again by induction by ℓ.

If $\ell = 1$ or $\ell = 2$, then by inspecting the rules of Definition 8, we can see that the only possibities for T are a variable $\mathtt{x}$ such that $\rho(\mathtt{x}) = v$, or a constant: 1, 0, or $\mathtt{nil}$. In any of these cases, there is a derivation of $\rho \vdash T \to v$ of size 1.

Suppose that $\ell > 2$. Then we break into cases depending on the form of T.

- If $T \equiv \mathtt{if}\ T_0\ \mathtt{then}\ T_1\ \mathtt{else}\ T_2$, there is some configuration $\boxed{\mathtt{if}\ T_0\ \mathtt{then}\ T_1\ \mathtt{else}\ T_2 \mid b}$ for some $b \in \{1, 0\}$ within $\boxed{T^\rho} \mapsto^\star \boxed{v}$, otherwise there would be no way to "eliminate" $\boxed{T^\rho}$. Therefore, this computation must have the following form (if $b = 1$, the case $b = 0$ is similar):

$$\boxed{\mathtt{if}\ T_0\ \mathtt{then}\ T_1\ \mathtt{else}\ T_2}$$
$$\boxed{\mathtt{if}\ T_0\ \mathtt{then}\ T_1\ \mathtt{else}\ T_2 \mid T_0}$$
$$\vdots$$
$$\boxed{\mathtt{if}\ T_0\ \mathtt{then}\ T_1\ \mathtt{else}\ T_2 \mid 1}$$
$$\boxed{T_1}$$
$$\vdots$$
$$\boxed{v}$$

  Therefore, there are some ℓ_0 and ℓ_1 such that $\ell_0 + \ell_1 + 1 = \ell$ and there computations $\boxed{T_0^\rho} \mapsto^\star \boxed{1}$ and $\boxed{T_1^\rho} \mapsto^\star \boxed{0}$ of lengths ℓ_0 and ℓ_1 respectively. By induction, there are derivations of $\rho \vdash T_0 \to 1$ and $\rho \vdash T_1 \to v$ of sizes at most ℓ_0 and ℓ_1 respectively; hence, there is a derivation of $\rho \vdash T \to v$ of size at most $\ell_0 + \ell_1 + 1 = \ell$.

- If $T \equiv \varphi(T_0, \ldots, T_{n-1})$ then the computation $\boxed{T^\rho} \mapsto^\star \boxed{v}$ has the following form:

$$\begin{array}{l}
\boxed{\varphi(T_0, \ldots, T_{n-2}, T_{n-1})} \\
\boxed{\varphi(T_0, \ldots, T_{n-2}, T_{n-1}) \mid T_0} \\
\vdots \\
\boxed{\varphi(T_0, \ldots, T_{n-2}, T_{n-1}) \mid v_0} \\
\boxed{\varphi(v_0, \ldots, T_{n-2}, T_{n-1})} \\
\vdots \\
\boxed{\varphi(v_0, \ldots, v_{n-2}, T_{n-1})} \\
\boxed{\varphi(v_0, \ldots, v_{n-2}, T_{n-1}) \mid T_{n-1}} \\
\vdots \\
\boxed{\varphi(v_0, \ldots, v_{n-2}, T_{n-1}) \mid v_{n-1}} \\
\boxed{\varphi(v_0, \ldots, v_{n-1})} \\
\boxed{v}
\end{array}$$

  Therefore, there are some v_i and ℓ_i for $i < n$ such that ℓ_i is the length of the computation $\boxed{T_i^\rho} \mapsto^\star \boxed{v_i}$ and $2 + \sum_{i<n}(1 + \ell_i) = \ell$. By induction, there are derivations of $\rho \vdash T_i \to v_i$ of size at most ℓ_i, for each $i < n$. Therefore, there is a derivation of $\rho \vdash T \to v$ of size $\sum_{i<n} \ell_i + 1 < \ell$.

- If $T \equiv (T_0, \ldots, T_{n-1})$, the argument is the similar to the above, except the symbol φ is absent, the final two configurations are identical, and $\ell = 1 + \sum_{i<n}(1 + \ell_i)$.

- If $T \equiv \mathtt{f}(T_0, \ldots, T_{n-1})$, then the computation $\boxed{T^\rho} \mapsto^\star \boxed{v}$ has the following form:

$$\begin{array}{l}
\boxed{\mathtt{f}(T_0, \ldots, T_{n-2}, T_{n-1})} \\
\boxed{\mathtt{f}(T_0, \ldots, T_{n-2}, T_{n-1})}\boxed{T_0} \\
\vdots \\
\boxed{\mathtt{f}(T_0, \ldots, T_{n-2}, T_{n-1})}\boxed{v_0} \\
\boxed{\mathtt{f}(v_0, \ldots, T_{n-2}, T_{n-1})} \\
\vdots \\
\boxed{\mathtt{f}(v_0, \ldots, v_{n-2}, T_{n-1})} \\
\boxed{\mathtt{f}(v_0, \ldots, v_{n-2}, T_{n-1})}\boxed{T_{n-1}} \\
\vdots \\
\boxed{\mathtt{f}(v_0, \ldots, v_{n-2}, T_{n-1})}\boxed{v_{n-1}} \\
\boxed{\mathtt{f}(v_0, \ldots, v_{n-1})} \\
\boxed{T^{\mathtt{f}}(\mathtt{x} \leftarrow (v_0, \ldots, v_{n-1}))} \\
\vdots \\
\boxed{v}
\end{array}$$

Therefore there are v_i for $i < n$ and ℓ_i for $i \leq n$ such that the computation $\boxed{T_i^\rho} \mapsto^\star \boxed{v_i}$ has length ℓ_i for $i < n$, the computation $\boxed{T^{\mathtt{f}}(\mathtt{x} \leftarrow (v_i)_{i<n})} \mapsto^\star \boxed{v}$ has length ℓ_n, and $\ell = \sum_{i \leq n}(1 + \ell_i)$. By induction there are derivations of $\rho \vdash T_i \to v_i$ of length ℓ_i for $i < n$ and a derivation of $[\mathtt{x} = (v_i)_{i<n}] \vdash T^{\mathtt{f}} \to v$ of length ℓ_n. Hence there is a derivation of $\rho \vdash T \to v$ of length $1 + \sum_{i \leq n} \ell_i \leq \ell$.

□

Appendix II: Encodings of programs

Here, we sketch an encoding of cons-free programs as used in Section 6 in a little more detail.

Given any set of atoms A, the set of *nested lists* over A is the closure of A over the listing operator. Concretely, every element of A is a nested list over A, and given any finite set $\{\ell_0, \ldots, \ell_{n-1}\}$ of nested lists over A, the list $\langle \ell_0, \ldots, \ell_{n-1} \rangle$ is a nested list over A.

Suppose A is countable, and fix an arbitrary encoding of A by binary strings. Then it is a standard construction to encode nested lists over A by binary strings such that the list primitives are polynomial time computable, see, e.g., [8]. Briefly, one starts by constructing a polynomial-time pairing system on binary strings. This means a polynomial-time computable bijection $2^\star \times 2^\star \to 2^\star$ with polynomial-time inverse functions isolating the first and second coordinates from a given pair.

Then one extends this pairing system to a polynomial-time tupling system. These allow us to check whether a string encodes a list, extract the length of the list, extract its first character, remove the first character, and add a character to the front of the list, all in polynomial time. By constructing lists out of strings which encode lists, we encode nested lists over A.

Notice that the set of program terms can be identified with the set of nested lists over some set A of atoms that consists of variables, recursive and primitive function symbols, and program keywords. A program can then be encoded by a list of such terms (the right and left halves of each line of the program).

By including the type of each variable and recursive function symbol as part of its encoding, we can easily check whether a given string encodes a well-formed term by simple recursion over the structure of the list, a polynomial-time operation. We can check whether a a given term contains or does not contain a given variable. Using these two operations, we can check whether a string encodes a well-formed program, all in polynomial time.

We can index occurrences term in a program according to (i) which line of the program they are located in, (ii) on the right- or left-hand side of the definition, and (iii) the sequence of list primitives needed to obtain this occurrence from the term it occurs in. (Nested lists can be identified with trees, and what this basically means is that we index vertices in the tree according to the path needed to get there from the root.)

We can obtain the encoding of the term named by a given index in time polynomial in the length of the program encoding, simply by extracting the correct term from the string encoding the program and applying the list primitives specified by the root-to-vertex path. Then, we can obtain indexings for the sub-occurrences of the original occurrence by taking the encoding of the resulting term, extracting its length, and extending the original root-to-vertex path by all of its one-point extensions.

This establishes the properties needed by Lemma 8 of Section 6.

References

[1] Eric W. Allender. P-uniform circuit complexity. *J. ACM*, 36(4):912–928, October 1989.

[2] Donald A. Alton. *"Natural" Programming Languages and Complexity Measures for Subrecursive Programming Languages: An Abstract Approach*, page 248–285. London Mathematical Society Lecture Note Series. Cambridge University Press, 1980.

[3] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach.* Cambridge University Press, 2009.

[4] Manuel Blum. A machine-independent theory of the complexity of recursive functions. *J. ACM*, 14(2):322–336, April 1967.

[5] G. Bonfante, R. Kahle, J. Y. Marion, and I. Oitavem. Towards an implicit characterization of NC^k. In Zoltán Ésik, editor, *Computer Science Logic*, pages 212–224, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[6] Ashok K. Chandra, Dexter C. Kozen, and Larry J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, January 1981.

[7] Oscar H. Ibarra. On two-way multihead automata. *Journal of Computer and System Sciences*, 7(1):28–36, 1973.

[8] Neil D. Jones. *Computability and complexity. From a programming perspective.* London: MIT Press, 1997.

[9] Neil D. Jones. LOGSPACE and PTIME characterized by programming languages. *Theoretical Computer Science*, 228(1):151–174, 1999.

[10] Neil D. Jones. The expressive power of higher-order types or, life without CONS. *Journal of Functional Programming*, 11(1):55–94, 2001.

[11] Cynthia Kop and Jakob Grue Simonsen. The power of non-determinism in higher-order implicit complexity: characterising complexity classes using non-deterministic cons-free programming. In Hongseok Yang, editor, *Programming Languages and Systems*, Lecture notes in computer science, pages 668–695. Springer, 2017.

[12] Yiannis N. Moschovakis. *Abstract Recursion and Intrinsic Complexity.* Lecture Notes in Logic. Cambridge University Press, 2018.

[13] Karl-Heinz Niggl and Henning Wunderlich. Implicit characterizations of FPTIME and NC revisited. *The Journal of Logic and Algebraic Programming*, 79(1):47–60, 2010. Special Issue: Logic, Computability and Topology in Computer Science: A New Perspective for Old Disciplines.

[14] Michael O. Rabin. Degree of difficulty of computing a function and a partial ordering of recursive sets. Technical Report 2, Hebrew University, 1960.

[15] Walter L. Ruzzo. Tree-size bounded alternation. *Journal of Computer and System Sciences*, 21(2):218–235, 1980.

[16] Walter L. Ruzzo. On uniform circuit complexity. *Journal of Computer and System Sciences*, 22(3):365–383, 1981.

[17] M. Sipser. *Introduction to the Theory of Computation.* Thomson Course Technology, 2006.

[18] I. H. Sudborough. Time and tape bounded auxiliary pushdown automata. In Jozef Gruska, editor, *Mathematical Foundations of Computer Science 1977*, pages 493–503, Berlin, Heidelberg, 1977. Springer Berlin Heidelberg.

[19] Anush Tserunyan. *(1) Finite generators for countable group actions; (2) Finite index pairs of equivalence relations; (3) Complexity measures for recursive programs.* PhD thesis, University of California, Los Angeles, 2013.

[20] H. Venkateswaran. Properties that characterize logcfl. *Journal of Computer and System Sciences*, 43(2):380–404, 1991.