Recursion, iteration, and circuit complexity

Siddharth Bhaskar[0000−0003−4157−8768]

University of Southern Denmark
`bhaskar@imada.sdu.dk`

Abstract. *Boolean circuits* and *branching programs* are two fundamental finite models of computation, and the asymptotic size and depth of families of such objects are important resources in complexity theory. We present a first-order functional programming language with tree-structured advice that captures computability by families of boolean circuits and whose tail recursive fragment captures computability by families of branching programs. In addition, the size and depth of each finitary model is reflected in the size and depth of the advice in the corresponding language. In this way are able to interpret the relationship between two fundamental finitary models of computation as a recursion/iteration distinction in a precise way.

Keywords: Boolean circuits · Branching programs · Cons-free computation.

1 Introduction

To the left is a *boolean circuit*, and to the right a *branching program*, each computing the XOR function on the bits x_1 and x_2.

$$\begin{tikzcd}
 & {} & \\
 & \lor \arrow[u] & \\
\land \arrow[ur] & & \land \arrow[ul] \\
\neg \arrow[u] & & \neg \arrow[u] \\
x_1 \arrow[u] \arrow[uurr] & & x_2 \arrow[u] \arrow[uull]
\end{tikzcd}
\qquad
\begin{tikzcd}
 & {} \arrow[d] & \\
 & x_1 \arrow[dl, "0"'] \arrow[dr, "1"] & \\
x_2 \arrow[d, "1"'] \arrow[drr, "0"] & & x_2 \arrow[dll, "1"'] \arrow[d, "0"] \\
1 & & 0
\end{tikzcd}$$

What are the semantics of each model? Imagine that we have an assignment of x_1 and x_2 to boolean values. Then,

- in the circuit, we compute a bit at each node by applying the operation labeling that node to the bits computed by its predecessors—at source nodes we query the value of the labeled bit—until we arrive at the sink;
- in the branching program, we start at the source, repeatedly query the value of the bit labeled by our current node and follow the 0-edge or the 1-edge depending on the result, until we arrive at a sink.

A single circuit or branching program thus computes a relation on binary strings of a fixed length; a *family* of these (one for each natural-number length) computes a relation on all binary strings.

Notice that the semantics of boolean circuits is naturally expressed as a divide-and-conquer recursion: to evaluate a circuit, evaluate the sub-circuits and combine. Similarly, the semantics of branching programs is naturally iterative, or equivalently *tail recursive*: to evaluate a branching program, ask a query, branch, and repeat.

This iterative/recursive distinction is no coincidence. In the present paper, we establish an analogy

$$\frac{\text{branching programs : iteration}}{\text{boolean circuits: recursion}} \tag{1}$$

by defining a simple functional programming language that captures computation by families of boolean circuits and whose tail recursive fragment captures computation by families of branching programs. In so doing we establish a new connection between two fundamental conceptual pairs, one from programming languages and the other from complexity theory.

Nonuniformity. We establish (1) by constructing four compiling functions, viz.,

1. from boolean circuit families to programs,
2. from programs back to boolean circuit families,
3. from branching program families to tail recursive programs, and
4. from tail recursive programs back to branching program families.

But there is a mismatch: families of circuits/branching programs are a *nonuniform* model of computation, meaning there's no a priori relationship between circuits for different input lengths, whereas programs are uniform. So we either have to impose uniformity on circuits or nonuniformity on programs. We choose the latter by endowing programs with *advice,* an additional input that a program may use during its computation. Advice is a well-known device in in complexity theory; there it is string-valued, while ours takes the form of a labeled tree.

Thus, the programs in 1-4 should really be understood as *programs with advice:* a program with a sequence of trees, one for each natural number, such that the program can use the n^{th} tree on strings of length n as an auxiliary input. And this lets us state the real point, which is that

> *The size and depth of circuits and branching programs is tightly connected to the size and depth of the advice trees for (nonuniform) recursive and iterative programs respectively.*

This is a gloss on Theorems 1 and 2, the main technical outcome of our work.

At first glance, this might seem entirely unremarkable. After all, the advice trees in question look a lot like the circuits or branching programs they come from. Why shouldn't we expect their depth and size to be closely related as well?

This would be a fair point if we were working with just one or the other model. The point is that we have located *both* models in a common setting. In circuit complexity, branching programs and boolean circuits are two different models with two different theories. The fact that we can toggle between the two by passing between full and tail recursion in some programming language is a novel perspective on their relationship. The fact that both models are so directly captured by the same type of advice thus becomes a feature, not a bug.

Related work. The present paper is foreshadowed by several earlier results relating recursion to time classes and tail recursion to space classes. Over finite ordered structures L is captured by (deterministic) transitive closure logic whereas P is captured by least fixed-point logic [4]. This can be interpreted as an iteration/recursion distinction. The direct ancestor of the present paper is Neil Jones' seminal paper [5] identifying P and L relations as those computable by general and tail recursive programs over a cons-free language over strings.

That paper can be viewed in the larger tradition *implicit computational complexity*; roughly, the enterprise of identifying various complexity classes as exactly the set of functions or relations computed by programs in some (non-Turing complete) language. In the cons-free framework, the original characterization of P and L was extended upward to exponential-time classes [6, 7], downward to the NC hierarchy [1], and sideways to classes of functions, not just relations [2].

Here we have presented a fourth extension: to certain nonuniform classes, via the introduction of advice. For example, Theorems 1 and 2 give us characterizations of $\mathsf{P/poly}$ and $\mathsf{L/poly}$ as the set of relations computed by cons-free programs and tail recursive programs respectively in the presence of polynomial-sized advice. As far as we are aware, we are the first to provide any implicit characterizations of nonuniform classes.

2 An oblivious cons-free language with advice

Types We use four types in our language. We have two copies of boolean values, viz.,

- a type $\mathsf{2}$ of booleans and
- a type $\mathsf{2}_{\mathsf{obv}}$ of *oblivious booleans*, so called because they do not depend on the bits of the input string.

Then we use two types which depend on a natural number n:

- A type $\mathsf{Fin}(n) = \{0, 1, \ldots, n-1\}$ of natural numbers bounded by n.
- A type $\mathsf{Adv}(n)$ of binary trees labeled by elements of $\mathsf{Fin}(n)$.

Additionally, each program has a single read-only input of type $\mathsf{Str}(n)$, binary strings of length n. However, it makes for a cleaner presentation to suppress explicit reference to this input from the syntax of the language, so we can avoid having program terms of this type.

While our language uses dependent types, it can hardly be called a "dependently typed language." The natural number n is not visible to the program (there are no program terms of type $\mathbb{N}$) and is fixed through the duration of a single computation. There is no way to create new types, let alone define new types from values.

Henceforth we suppress the dependence of the types on n for legibility, and define the syntax of our language as if it were statically typed.

Primitive operations Our language is *cons-free* in the sense that any Str or Adv object can only be read or destructed—there is no way to create new values of these types. In particular, we have:

- $\mathtt{tt}, \mathtt{ff} : 2_{\mathsf{obv}}$ (constants naming boolean values $\top$ and $\bot$)
- $\mathtt{bit} : \mathsf{Str} \times \mathsf{Fin} \to 2$ (What bit does the input string carry at a given index? As mentioned above, we will omit explicit reference to the first argument of $\mathtt{bit}$, as there is only one string which it can be called upon..)
- The advice primitives:
  - $\mathtt{empty} : \mathsf{Adv} \to 2_{\mathsf{obv}}$ (is the tree empty?)
  - $\mathtt{label} : \mathsf{Adv} \to \mathsf{Fin}$ (the label of the root of nonempty tree)
  - $\mathtt{left}, \mathtt{right} : \mathsf{Adv} \to \mathsf{Adv}$ (the left and right subtrees)

Finally, what can we do to elements of Fin? As it happens, our results are completely independent of the set of primitive operations on this data type. So let us simply fix an arbitrary set Φ of operations, each of type

$$\mathsf{Fin} \times \cdots \times \mathsf{Fin} \to \sigma,$$

where σ is either Fin or 2_{obv}. This can include, for example, arithmetic operations like addition or multiplication, augmented somehow to handle overflow. It can include relations like equality or comparison. But we emphasize that Φ is completely arbitrary—it can even be empty!

Program terms First, our variables and recursive function symbols:

- We have a single variable C of type Adv and countably many variables of type Fin.
- We have countably many recursive function symbols for each type of the form $\mathsf{Adv} \times \mathsf{Fin} \times \cdots \times \mathsf{Fin} \to \sigma$, where there are zero or more inputs of type Fin and $\sigma \in \{2, 2_{\mathsf{obv}}, \mathsf{Adv}, \mathsf{Fin}\}$.

Remark 1. The limitation to a single Adv-variable means, essentially, that we maintain a *single* pointer to the initial Adv-object, and (since Adv-data can only be destructed) that pointer can only move from the root to some leaf. In other words, advice is not just read-only, it is *read-once.*

We impose this limitation because it makes some combinatorics easier, but we are unsure whether it is essential. We will remark exactly where we use this assumption below, and discuss what would happen were we to drop it in the Appendix.

Next, our term constructors. Of course, each variable is a term of its respective type.

– Each primitive operation and recursive function symbol applied to a term is a term, with types behaving as they should. So, for example,

$$\frac{T : \mathsf{Fin}}{\mathtt{bit}(T) : 2} \qquad \frac{T : \mathsf{Adv}}{\mathtt{left}(T) : \mathsf{Adv}} \quad \frac{T_1 : \mathsf{Adv} \qquad T_2 : \mathsf{Fin}}{\mathtt{f}(T_1, T_2) : 2}\ \mathtt{f} : \mathsf{Adv} \times \mathsf{Fin} \to 2$$

and similarly for all the other primitive and recursive function symbols. Note how we suppress reference to the first Str argument of $\mathtt{bit}$.

– We can take cases on values of the type 2_{obv}, Fin, and Adv using oblivious booleans, viz.,

$$\frac{T_0 : 2_{\mathsf{obv}} \qquad T_1 : \tau \qquad T_2 : \tau}{\mathtt{if}\ T_0\ \mathtt{then}\ T_1\ \mathtt{else}\ T_2 : \tau}\ \tau \in \{2_{\mathsf{obv}}, \mathsf{Fin}, \mathsf{Adv}\}$$

– We can convert oblivious booleans into booleans (but not vice versa), viz.,

$$\frac{T : 2_{\mathsf{obv}}}{T : 2}$$

(Notice that this means that $\mathtt{tt}, \mathtt{ff}$ also name the values in 2.)

– Finally, we can take cases on booleans using booleans, viz.,

$$\frac{T_0 : 2 \qquad T_1 : 2 \qquad T_2 : 2}{\mathtt{if}\ T_0\ \mathtt{then}\ T_1\ \mathtt{else}\ T_2 : 2}$$

Definition 1. *A term is* explicit *in case it contains no occurrences of any recursive function symbol. A term is* tail recursive *if it is produced by the following context-free grammar:*

$$T := E \mid \mathtt{f}(\boldsymbol{E}) \mid \mathtt{if}\ E\ \mathtt{then}\ T\ \mathtt{else}\ T,$$

where E stands for any explicit term, and $\boldsymbol{E}$ for a tuple of explicit terms.

Programs A *program* consists of a collection $\{\mathtt{f}_0, \dots, \mathtt{f}_{k-1}\}$ of recursive function symbols plus, for each $i < k$, a line of the form

$$\mathtt{f}_i(\mathtt{C}, \mathtt{x}_i) = T_i,$$

where (1) $\mathtt{x}_i$ is a list of zero or more Fin-type variables, (2) $\mathtt{f}_i(\mathtt{C}, \mathtt{x}_i)$ and T_i are terms of the same type, (3) the only Fin-type variables that occur in T_i are contained in $\mathtt{x}_i$, and (4) the only recursive function symbols that occur in T_i are contained in $\{\mathtt{f}_0, \dots, \mathtt{f}_{k-1}\}$. The *head* of the program is the term $\mathtt{f}_0(\mathtt{C}, \mathtt{x}_0)$.

Remark 2. For T_0 and T_1 to be *terms of the same type*, we mean that $T_0 : \tau \iff T_1 : \tau$ for any type τ. In particular, if we make a recursive definition $\mathtt{f}(\mathtt{x}) = T$ where $T : 2_{\mathsf{obv}}$, then $\mathtt{f}(\mathtt{x})$ must have type 2_{obv} as well.

Definition 2. *A program is* tail recursive *if each term that occurs inside it is tail recursive.*

Semantics We adopt a strict, call-by-value semantics. The basic form of judgment we define is

$$n, s, \Gamma \vdash_p T \to v,$$

where $n \in \mathbb{N}$, $s \in \mathsf{Str}(n)$, Γ is an environment; i.e., a finite function from variables to values, p is a program, T is a program term, and v is a value. Moreover,

- T occurs (as a subterm of some term) in p,
- every variable that occurs in T is a member of the domain of Γ,
- if T has type τ, then v has type $\tau(n)$, and
- if a variable $\mathtt{v}$ in the domain of Γ has type τ, then $\Gamma(\mathtt{v})$ has type $\tau(n)$.

Following are the rules. Since n, s, and p are fixed within a single proof tree, we omit them below for legibility.

- In case T is a single variable:

$$\frac{}{\Gamma \vdash \mathtt{v} \to v}\ {\scriptstyle \Gamma(\mathtt{v}) = v}$$

- In case T is $\varphi(T_1, \ldots, T_n)$ for some primitive function symbol φ:

$$\frac{\Gamma \vdash T_1 \to v_1 \qquad \ldots \qquad \Gamma \vdash T_n \to v_n}{\Gamma \vdash \varphi(T_1, \ldots, T_n) \to v}\ {\scriptstyle \varphi(v_1, \ldots, v_n) = v}$$

  A particular case of this scheme is where $\varphi \equiv \mathtt{bit}$. This is exactly where the suppressed Str-input s becomes important:

$$\frac{\Gamma \vdash T_1 \to v}{\Gamma \vdash \mathtt{bit}(T_1) \to s_v}$$

- In case T is $\mathtt{f}(T_1, \ldots, T_n)$ for some recursive function symbol $\mathtt{f}$:

$$\frac{\Gamma \vdash T_1 \to v_1 \qquad \ldots \qquad \Gamma \vdash T_n \to v_n \qquad \Delta \vdash T^{\mathtt{f}} \to v}{\Gamma \vdash \mathtt{f}(T_1, \ldots, T_n) \to v}$$

  where Δ is the environment $\{\mathtt{x}_1 \mapsto v_1, \ldots, \mathtt{x}_n \mapsto v_n\}$ and $T^{\mathtt{f}}$ is right-hand side of the recursive definition of $\mathtt{f}$ in p.
- And finally, in case T is $\mathtt{if}\ T_0\ \mathtt{then}\ T_1\ \mathtt{else}\ T_2$:

$$\frac{\Gamma \vdash T_0 \to \top \qquad \Gamma \vdash T_1 \to v}{\Gamma \vdash \mathtt{if}\ T_0\ \mathtt{then}\ T_1\ \mathtt{else}\ T_2 \to v}$$

$$\frac{\Gamma \vdash T_0 \to \bot \qquad \Gamma \vdash T_2 \to v}{\Gamma \vdash \mathtt{if}\ T_0\ \mathtt{then}\ T_1\ \mathtt{else}\ T_2 \to v}$$

  (The rules are identical regardless of whether we're dealing with oblivious or non-oblivious boolean values.)

Next, we name a specific class of programs which are comparable to families of circuits or branching programs insofar as they compute subsets of strings.

Definition 3. *An* advice sequence *is a dependent function* $A : \prod_{n:\mathbb{N}} \mathsf{Adv}(n)$*. Instead of* $A(n)$ *we will write* A_n*, by analogy with families of circuits, which are typically written* $\{C_n\}_{n\in\mathbb{N}}$*.*

Definition 4. *An* acceptor *is an always-convergent program whose head has the form* $\mathtt{f}_0(\mathtt{C})$ *for some recursive function symbol* $\mathtt{f}_0 : \mathsf{Adv} \to 2$*. For an acceptor* p*, define*

$$[\![p]\!] : \sum_{n:\mathbb{N}} \mathsf{Str}(n) \times \mathsf{Adv}(n) \to 2$$

as follows. For any $s : \mathsf{Str}(n)$ *and* $C : \mathsf{Adv}(n)$*,* $[\![p]\!](C, s)$ *is the unique boolean* b *such that*

$$n, s, \{\mathtt{C} = C\} \vdash_p \mathtt{f}_0(\mathtt{C}) \to b.$$

Definition 5. *A* ready acceptor *is a pair* (p, A) *where* p *is an acceptor and* A *an advice sequence. (Here we use the word* ready *in an archaic sense that means* advised *or* counseled.*) For a ready acceptor* (p, A)*, define*

$$[\![p, A]\!] : \sum_{n\in\mathbb{N}} \mathsf{Str}(n) \to 2$$

by $[\![p, A]\!](s) = [\![p]\!](A_n, s)$*, for strings* s *of length* n*.*

In other words, ready acceptors decide subsets of binary strings. The convention in complexity theory is to call sets of strings *languages,* which is slightly unfortunate (they are semantic objects after all) but we will follow this practice, and trust that it will not cause too much trouble.

Questions about the relative power of ready acceptors vs. ready tail recursive acceptors with approximately the same amount of advice often reduce to hard open problems in complexity theory that we do not hope to resolve. However, it is possible to construct a function computable by an *acceptor* but no tail recursive acceptor; see the Appendix for details.

Oblivious types are oblivious. At some point, we are compelled to state the obvious fact about terms of the "oblivious" types 2_{obv}, Adv, and Fin—namely, that their denotations are independent of the string input.

Lemma 1. *Unless* $T : 2$*, the relation* $n, s, \Gamma \vdash_p T \to v$ *is independent of* s*.*

Proof (Proof sketch). In fact something stronger holds, namely that every rule in the derivation of $n, s, \Gamma \vdash_p T \to v$ is independent of s. The proof goes by induction on the size of this derivation. If T is a single variable, then v depends only on Γ. If T is $\mathtt{if}\ T_0\ \mathtt{then}\ T_1\ \mathtt{else}\ T_2$ or $\mathtt{f}(T_1, \ldots, T_n)$, then each of the premises of $n, s \vdash_p T \to v$ are independent of s. (In the latter case note that the definition $T^{\mathtt{f}}$ of $\mathtt{f}$ must also have oblivious type.)

The only thing left to check are calls to the primitives other than `bit`. But these denote operations on either the type $\mathsf{Adv}(n)$ or the type $\mathsf{Fin}(n)$, so have nothing do with s.

3 General recursion and boolean circuits

The results in this section mimic the well-known compilations between boolean circuit families on one hand and Turing machines with advice on the other.

A variation of the circuit model. For our purposes it will be useful to consider a slight of boolean circuits, where we replace the boolean operations $\wedge$, $\vee$, and $\neg$ by if-then-else branching plus boolean constants.

Definition 6. *A* boolean circuit on n inputs *is a finite partial order of boolean-valued variables P, Q, $R, \ldots$ such that each variable P is defined from its immediate predecessors in one of three ways:*

- $P = \texttt{if}\ Q\ \texttt{then}\ R\ \texttt{else}\ S$,
- $P = \texttt{tt}$, $P = \texttt{ff}$, *or*
- $P = \texttt{bit}(i)$, *for any* $i < n$.

So each variable has either 0 or 3 immediate predecessors depending on its definition. Additionally, the partial order has a unique maximum (the sink*).*

Remark 3. The size of the circuit is simply the number of variables, and the depth is the length of the longest chain in this partial order. Translating between $\{\wedge, \vee, \neg\}$-circuits and if-then-else circuits increases the depth and size by at most a constant factor, which for our purposes is inessential.

Definition 7. *Given a boolean circuit with sink P on n inputs, we define the denotation $[\![P]\!] : \mathsf{Str}(n) \to 2$ by cases as follows.*

- *If $P = \texttt{if}\ Q\ \texttt{then}\ R\ \texttt{else}\ S$, then $[\![P]\!](s)$ is $[\![R]\!](s)$ or $[\![S]\!](s)$ depending on whether $[\![Q]\!](s)$ is true or false ($\top$ or $\bot$) respectively.*
- *If $P = \texttt{tt}$ or $P = \texttt{ff}$, then $[\![P]\!](s)$ is true or false respectively.*
- *If $P = \texttt{bit}(i)$, then $[\![P]\!](s)$ is s_i, the i^{th} bit of s.*

Encoding circuits by Adv*-objects.* Any finite rooted partial order has a set of immediate predecessors, which are themselves finite rooted partial orders. This decomposition is the basic idea behind encoding a circuit as an Adv-object.

Of course the sub-circuits of a circuit are not necessarily disjoint, but we ignore this for the purposes of the encoding: as our programs are cons-free and purely functional, they cannot distinguish between two different occurrences of the same sub-circuit in a circuit.

Definition 8. *We define a map $C \mapsto C^\star$ from boolean circuits on n inputs to $\mathsf{Adv}(n)$-objects as follows. If A and B are trees, then by (A, B) we mean the tree with left subtree A and right subtree B, labeled by $0 : \mathsf{Fin}(n)$. (This is a "dummy labeling;" we need a label, so we put 0.)*

- *If $C = \texttt{bit}(i)$, $C^\star$ is a single leaf labeled i.*
- *If $C = \texttt{if}\ C_0\ \texttt{then}\ C_1\ \texttt{else}\ C_2$, then $C^\star$ is $(C_0^\star, (C_1^\star, C_2^\star))$*

– *If $C = \texttt{tt}$, $C^\star$ is a root with a single left child. If $C = \texttt{ff}$, then $C^\star$ is a root with a single right child.*

Remark 4. The depth of $C^\star$ is bounded by twice the depth of C, plus some constant, so

$$\mathrm{depth}(C^\star) \in O(\mathrm{depth}(C)). \tag{2}$$

The size (number of distinct subtrees) of $C^\star$ is bounded by twice the size of C, plus some constant, so

$$\mathrm{size}(C^\star) \in O(\mathrm{size}(C)). \tag{3}$$

3.1 A circuit evaluator

The following program is an acceptor that correctly evaluates a given circuit (encoded as an Adv-object `C`) on an (implicitly given) string input.

```
eval(C) = if isLeaf(C) then bit(label(C)) else
          if isBool(C) then isTrue(C) else
          if eval(ifBr(C)) then eval(thenBr(C)) else eval(elseBr(C))
```

where

– `isLeaf(C)` checks that both subtrees of `C` are empty,
– `isBool(C)` checks that exactly one subtree of `C` is empty,
– `isTrue(C)` checks that the right subtree of `C` is empty,
– `ifBr(C)` is `left(C)`,
– `thenBr(C)` is `left(right(C))`, and
– `elseBr(C)` is `right(right(C))`.

We omit a proof of correctness, which wouldn't add any insight over reading the program.

3.2 Compiling recursive programs into circuits

In this section we fix a ready acceptor (p, A) and construct a family $\{C_n\}$ of circuits deciding the same language as (p, A).

Definition 9. *A* special pair *is a tuple (Γ, T) such that T is a $\mathsf{2}$-valued p-term and Γ is an environment binding the Fin-typed variables of T to values in $\mathsf{Fin}(n)$ and `C` to a subtree of A_n.*

Definition 10. *We define a circuit family $\{C_n\}_{n\in\mathbb{N}}$ from p and A as follows. For each special pair (Γ, T) and $n \in \mathbb{N}$, we define a circuit letter $(\Gamma, T)^\diamond_n$ of C_n by the cases below. (We omit the fixed subscript n for legibility.)*

– *If $T : \mathsf{2}_{\mathsf{obv}}$, then $(\Gamma, T)^\diamond = \texttt{tt}$ or $(\Gamma, T)^\diamond = \texttt{ff}$, depending on whether $n, \Gamma \vdash T \to \top$ or $n, \Gamma \vdash T \to \bot$.*

– *If T has the form* $\mathtt{bit}(T_0)$*, then* $(\Gamma, T)^\diamond = \mathtt{bit}(v)$*, where* $v : \mathsf{Fin}(n)$ *is the unique value satisfying* $n, \Gamma \vdash T_0 \to v$*.*
– *If T has the form* $\mathtt{if}\ T_0\ \mathtt{then}\ T_1\ \mathtt{else}\ T_2$ *then in* C_n

$$(\Gamma, T)^\diamond = \mathtt{if}\ (\Gamma, T_0)^\diamond\ \mathtt{then}\ (\Gamma, T_1)^\diamond\ \mathtt{else}\ (\Gamma, T_2)^\diamond.$$

– *Finally, suppose that* $T \equiv \mathtt{f}(T_1, \ldots, T_n)$*; then each* T_i *is either* Fin- *or* Adv-*valued. Say that*

$$\mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n) = T^{\mathtt{f}}$$

is the recursive definition of $\mathtt{f}$ *in* p*. Then in* C_n*, define*

$$(\Gamma, T)^\diamond = (\Delta, T^{\mathtt{f}})^\diamond,$$

where Δ *is the environment binding* $\mathtt{x}_i$ *to the unique value* v_i *such that* $n, \Gamma \vdash T_i \to v_i$*.*

*Finally, the **head** of the circuit* C_n *is* $(\{\mathtt{C} = A_n\}, \mathtt{f}_0(\mathtt{C}))^\diamond$*.*

Notice that this last two cases may introduce lines of the form $P = Q$ in the circuit, which was not part of our original syntax, and there may be $(\Gamma, T)^\diamond$ which are not accessible from the head of the circuit. But these may be easily eliminated without increasing the size or depth of the circuit.

Correctness and efficiency. We can prove that

$$n, s, \Gamma \vdash T \to b \implies [\![(\Gamma, T)^\diamond_n]\!](s) = b$$

by induction on the size of the derivation of $n, s, \Gamma \vdash T \to b$. The proof is entirely straightforward. It shows, moreover, that the circuit is genuinely acyclic—at least the part accessible from the head, which is all we care about. Since the program p always converges, we get the converse implication for free. This shows that the ready acceptor (p, A) and the circuit family $\{C_n\}$ compute exactly the same language of binary strings.

The size of C_n is bounded by the number of special terms, which is again bounded by a fixed polynomial (dependent only on p) in n (i.e., the size of $\mathsf{Fin}(n)$) and the number of distinct subtrees of A_n (i.e., the size of A_n). This gives us

$$\text{size}(C_n) \in (n + \text{size}(A_n))^{O(1)}. \tag{4}$$

Finally, let's work on bounding the depth of C_n by the depth of A_n. A *chain* in C_n is a linearly ordered subset of special terms. No special term (Γ, T) may occur twice in a chain, or else the computation would diverge. Notice that for each subtree A of A_n, the number of special terms where $\Gamma(\mathtt{C}) = A$ is bounded above by a fixed polynomial in n.

Moreover, if environments Γ and Δ occur within are consecutive special terms in a chain, it must be the case that $\Delta(\mathtt{C})$ is the denotation of some Adv-valued term with respect to the environment Γ. This means that $\Delta(\mathtt{C})$ will be a subtree of $\Gamma(\mathtt{C})$, i.e., below $\Gamma(\mathtt{C})$ as a node in A_n. Therefore, the set of subtrees that occur as $\Gamma(\mathtt{C})$ for some (Γ, T) in a single chain must all be contained within a single root-to-leaf path in A_n. (This is where we use our limitation to a single Adv-variable $\mathtt{C}$; in the Appendix, we consider the consequences of loosening this assumption.)

Hence for special terms within a single chain, there are at most $\mathrm{depth}(A_n)$ values that occur as $\Gamma(\mathtt{C})$, and for each of these there are at most polynomially many special terms that bind $\mathtt{C}$ to that value. Therefore,

$$\mathrm{depth}(C_n) \in (n + \mathrm{depth}(A_n))^{O(1)}. \tag{5}$$

The following theorem sums up equations (2)-(5). For a family F of functions of type $\mathbb{N} \to \mathbb{N}$ to be *polynomially closed*, we mean that the sum or product of any two functions in F is bounded above almost everywhere by another function in F. By a *size-F* (or *depth-F*) family of circuits or advice, we mean that, e.g., $n \mapsto \mathrm{size}(C_n)$ is bounded above almost everywhere by a function in F.

Theorem 1. *The following are equivalent, for any language L of binary strings and polynomially closed families F, G of functions $\mathbb{N} \to \mathbb{N}$ containing the identity:*

1. *There is a size-F, depth-G family of circuits computing L.*
2. *There is a size-F, depth-G advice sequence A and an acceptor p such that (p, A) computes L.*

Remark 5. Theorem 1 gives us a characterization of $\mathsf{P/poly}$ by recursive programs with polynomial-sized advice. At first glance, the depth-G adjectives of Theorem 1 seems not to be doing much, as the least such G is the class of polynomials, and *every* language is computed by a polynomial-depth (indeed, constant-depth!) circuit family.

However, notice that our bounds are *simultaneously* on size and depth. Not every language may be computable in, for example, simultaneous polynomial-depth and quasi-polynomial ($2^{\mathrm{polylog}(n)}$) size. Moreover, the transformations above preserve uniformity. In other words, if we start with a computable advice sequence, we get a computable circuit family and vice versa—and being definable in computable polynomial depth is a real restriction. (We could surely say something much stronger if we cared to measure the complexity of the transformation.)

Still: most interesting depth bounds are sub-polynomial, and it is a definite limitation of our work that we do not have a characterization of, for example, the nonuniform NC hierarchy. As programs with no advice whatsoever already capture P, we need more restrictive assumptions if we want a tighter bound in equation 9.

A natural candidate is to consider *time-bounded* classes of programs since, in the absence of advice, these capture classes in the uniform NC hierarchy [1]. However, [7] cautions us that different extensions of the cons-free language may not interact in predictable ways when combined.

4 Tail recursion and branching programs

Encoding branching programs as Adv*-objects.* If a branching program B consists of a single node, it is labeled by either `tt` or `ff`. In this case, define the Adv-object $B^\star$ by either a root with a single left child, or a root with a single right child, with "dummy labelings" of 0.

Otherwise a branching program is a single source labeled by some $i < n$ with a 0-labeled outgoing edge and a 1-labeled outgoing edge to other branching programs B_0 and B_1 respectively, In this case define $B^\star$ to be the tree with root labeled by i and left and right subtrees $B_0^\star$ and $B_1^\star$ respectively.

Remark 6. The depth of $B^\star$ is bounded by the depth of B plus 1, so

$$\text{depth}(B^\star) \in O(\text{depth}(B)). \tag{6}$$

The size (number of distinct subtrees) of $B^\star$ is bounded by the size of B, plus some constant, so

$$\text{size}(B^\star) \in O(\text{size}(B)). \tag{7}$$

4.1 A branching program evaluator

The following tail recursive acceptor correctly evaluates a given branching program (encoded as an Adv-object `C`) on an (implicitly given) string input.

```
eval(C) = if isBool(C) then isTrue(C) else
          if bit(label(C)) then eval(left(C)) else eval(right(C))
```

where, as above,

- `isBool(C)` checks that exactly one subtree of `C` is empty and
- `isTrue(C)` checks that the right subtree of `C` is empty.

Again, we omit a proof of correctness.

4.2 Compiling tail recursive programs into branching programs

Just as in Section 3.2, fix a ready acceptor (p, A); assume additionally that p is tail recursive.

Definition 11. *We define a family $\{B_n\}_{n\in\mathbb{N}}$ of branching programs. For each special pair (Γ, T) there is a node $(\Gamma, T)^\dagger_n$ of B_n defined as follows, omitting the fixed subscript n for legibility:*

- *if $T : 2_{\mathsf{obv}}$, then $(\Gamma, T)^\dagger$ is a single leaf labeled by the unique boolean value b such that $n, \Gamma \vdash T \to b$,*

– *if* $T \equiv \texttt{bit}(T_0)$*, then* $(\varGamma, T)^\dagger$ *is the branching program*

$$\begin{tikzcd} & v \arrow[dl, "0"'] \arrow[dr, "1"] & \\ 0 & & 1 \end{tikzcd}$$

where v *is the unique* $\mathsf{Fin}(n)$*-value such that* $n, \varGamma \vdash T_0 \to v$*,*

– *if* $T \equiv \texttt{if}\ T_0\ \texttt{then}\ T_1\ \texttt{else}\ T_2$*,* $(\varGamma, T)^\dagger$ *is defined by taking a copy of* $(\varGamma, T_0)^\dagger$ *and redirecting any arrows into* 0*-leaves and* 1*-leaves to, instead, the sources in* $(\varGamma, T_0)^\dagger$ *and* $(\varGamma, T_1)^\dagger$ *respectively.*

– *if* $T \equiv \texttt{f}(T_1, \ldots, T_n)$*, then both arrows out of* $(\varGamma, T)^\dagger$ *point to* $(\varDelta, T^{\texttt{f}})^\dagger$*, where* $\texttt{f}(\texttt{x}_1, \ldots, \texttt{x}_n) = T^{\texttt{f}}$ *is the recursive definition of* $\texttt{f}$ *in* p*, and* $\varDelta$ *is the environment binding* $\texttt{x}_i$ *to the unique value* v_i *such that* $n, \varGamma \vdash T_i \to v_i$*.*

Finally, the ***source*** *of* B_n *is the node* $(\{\texttt{C} = A_n\}, \texttt{f}_0(\texttt{C}))^\dagger_n$*. We may remove all nodes not accessible from this source.*

Correctness and efficiency. Just like the relationship between general recursion and circuits, we have

$$n, s, \varGamma \vdash T \to b \implies [\![(\varGamma, T)^\dagger_n]\!](s) = b,$$

which can be proved by a straightforward induction of the size of the derivation on the left-hand side. And just like before, this shows us that B_n is a well-defined (acyclic) branching program, and we get the converse implication from the fact that p never diverges.

The arguments bounding the size and depth of B_n in terms of the size and depth of A_n are exactly like those for circuits. The size of B_n is bounded above by the number of special pairs; hence„

$$\mathrm{size}(B_n) \in (n + \mathrm{size}(A_n))^{O(1)}. \tag{8}$$

For any subtree A of A_n, there is a fixed polynomial-in-n upper bound on the number of special pairs $(\varGamma, T)$ such that $\varGamma(C) = A$. If environments $\varGamma$ and $\varDelta$ occur consecutively in a chain in B_n, then $\varDelta(\texttt{C})$ is the denotation of some Adv-valued term with respect to environment $\varGamma$; hence $\varDelta(\texttt{C})$ is below $\varGamma(\texttt{C})$ as vertices in A_n. Therefore, if we look at environments $\varGamma$ that occur within a chain in B_n, the values $\varGamma(\texttt{C})$ occur within a chain in A_n. But there are at most polynomial-in-n special pairs $(\varGamma, T)$ in that chain before $\varGamma(\texttt{C})$ must decrease, so

$$\mathrm{depth}(B_n) \in (n + \mathrm{depth}(A_n))^{O(1)}. \tag{9}$$

Hence, collecting equations (6)-(9), we get the tail recursive analogue of Theorem 1 Consequently, we obtain a characterization of $\mathsf{L/poly}$ by tail recursive programs with polynomial-sized advice.

Theorem 2. *The following are equivalent, for any language* L *of binary strings and polynomially closed families of functions* $\mathbb{N} \to \mathbb{N}$ *containing the identity:*

1. *There is a size-F, depth-G family of branching programs computing L.*
2. *There is a size-F, depth-G advice sequence A and a tail recursive acceptor p such that (p, A) computes L.*

As every language can be computed with a polynomial-depth (indeed, depth-n) family of branching programs, our comments in Remark 5 apply here as well.

5 Future directions

There are two parts to this section. First, let us discuss specific technical improvements to the present paper. We are, in some sense, "as lazy as possible," doing as little as we need to showcase the core analogy (1). However, we anticipate by working a little harder we can extend our capturing results along at least three axes, viz.:

1. The uniformity of our advice sequences is certainly controlled by the uniformity of our circuit families, but we do not bother to measure it. If we did, we would capture circuit classes of variable uniformity, and possibly recover the original characterizations of P and L as a special case.
2. The size and depth of our advice sequences is within a polynomial of the size and depth of our circuit families, but we could probably tighten this by paying attention to the running time of our programs. If we did, we might capture interesting circuit classes like NC.
3. Finally, we only consider circuits and branching programs over boolean values, whereas one could consider arithmetic, algebraic, or even more general data. If we did, we might capture arithmetic and algebraic analogues of the complexity classes under discussion here.

Secondly, and more generally, let us discuss some connections suggested by this work that are vague but tantalizing. In the uniform setting, circuit size and depth are closely related to (Turing machine) time and space, whereas the situation is reversed for branching programs [3, 10, 12]. While this is no longer true in the nonuniform setting, this at least strongly suggests that space-time tradeoffs in complexity theory might be related to recursion-iteration tradeoffs in programming languages.

This is not completely out of the blue; for example, the (iterative) dynamic programming implementation of a recursive scheme essentially trades time for space. However nothing like a general account of such phenomena is known.

References

1. Siddharth Bhaskar, Cynthia Kop, and Jakob Grue Simonsen. Subclasses of ptime interpreted by programming languages. *Theory of Computing Systems*, 67:437 – 472, 2022.
2. Siddharth Bhaskar and Jakob Grue Simonsen. Read/write factorizable programs. *Journal of Functional Programming*, 33, 2023.

3. Allan Borodin. On relating time and space to size and depth. *SIAM J. Comput.*, 6:733–744, 1977.
4. Neil Immerman. Languages that capture complexity classes. *SIAM Journal on Computing*, 16(4):760–778, 1987.
5. Neil D. Jones. Logspace and ptime characterized by programming languages. *Theoretical Computer Science*, 228(1):151 – 174, 1999.
6. Neil D. Jones. The expressive power of higher-order types or, life without cons. *Journal of Functional Programming*, 11:55 – 94, 2001.
7. 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. *ArXiv*, abs/1701.05382, 2017.
8. Yiannis N. Moschovakis. *Abstract Recursion and Intrinsic Complexity.* Lecture Notes in Logic. Cambridge University Press, 2018.
9. Michael S. Paterson and Carl E. Hewitt. Comparative schematology. In *Project MAC Conference on Concurrent Systems and Parallel Computation*, 1970.
10. Walter L. Ruzzo. On uniform circuit complexity. *20th Annual Symposium on Foundations of Computer Science (sfcs 1979)*, pages 312–318, 1979.
11. Jerzy Tiuryn. A simplified proof of DDL < DL. *Information and Computation*, 81(1):1–12, 1989.
12. Ingo. Wegener. *The complexity of Boolean functions.* Wiley-Teubner series in computer science. Teubner, Stuttgart, 1987.

Appendix

Here we collect two extended remarks.

General recursion is strictly more powerful than tail recursion for acceptors. There is function of type $\sum_{n:\mathbb{N}} \mathsf{Str}(n) \times \mathsf{Adv}(n) \to 2$ computable by an acceptor but no tail recursive acceptor. Since nonuniform complexity classes are captured by ready acceptors, this result does not obviously separate any of them, but it is still worth recording here.

Definition 12. *The* leaf support problem *is the following decision problem, parameterized by a natural number n: for any $C : \mathsf{Adv}(n)$ and $s : \mathsf{Str}(n)$, does s have a "1" in an index that labels some leaf of C?*

There is a general recursive program solving the leaf support problem; viz.,

```
f(C) =if isLeaf(C) then bit(label(C)) else
      if f(left(C)) then tt else f(right(C)).
```

On the other hand,

Theorem 3. *There is no tail recursive program solving the leaf support problem.*

Proof (Proof sketch). Fix s to the length-2 string $01 : \mathsf{Str}(2)$; then the leaf support problem specializes to deciding, for $C : \mathsf{Adv}(2)$, whether C has a leaf labeled by 1. Further specialize to C which have *at most one* (i.e., zero or one)

1-labeled leaf. But this is already impossible: even restricted to these inputs, we are forced to query every leaf of C, which requires time depth(C) in the worst case. However, every tail recursive program will halt in time polynomial in depth(C) over these inputs.

The inability of a tail recursive program to "view" each leaf of a read-only binary tree is the oldest known extensional separation between recursion and iteration, dating back to Paterson & Hewitt in 1970 [9]. An elaboration of the above sketch can be found as the "Second Proof of Theorem 1" in that paper, and a more modern treatment can be found in Theorem 2G.1 of Moschovakis [8]. It is an open question whether this can help us separate complexity classes.

What if we dropped the read-once limitation on advice? Could we still prove equation 6 if our programs had more than one Adv-valued variables? Each such variable can be thought of as a pebble on (or pointer to) some location in the advice tree. So suppose we have a finite rooted tree and a some finite number of pebbles, which start at the root of the tree. Suppose there are two types of moves: we can move a pebble from its current location to any child, or we can move a pebble on top of any other currently occupied location.

We are interested in how long we can carry this out without repeating a configuration, by which we mean an assignment of pebbles to locations. (Note that the number of distinct configurations is the size of the tree raised to the number of pebbles.) With a single pebble we are bound by the depth of the tree, and this is exactly the combinatorial fact we use above. With two pebbles we are bounded by the depth squared, and begin to feel hopeful.

Alas, with only three pebbles, we can find a sequence of distinct configurations at least as long as the size of any tree. (Finding this is a nice exercise!) Thus we might start to doubt that Theorem 1 fails in the presence of more Adv-variables.

However such pessimism might also be premature: pebble games exploit only one kind of limitation on programs. Speaking loosely, pebble games are played by omniscient beings who can see the whole tree; a program, by contrast, has limited memory, and only sees the parts of the tree in front of it. Exploiting some sort of "information-theoretic" limitation might be enough to control the length of any pebble game realized by an actual program by a polynomial in the depth. We can see an example of exactly such a combination of tree pebbling and information theory in Theorem 1 of [11], so it is quite plausible. However, for the purposes of this paper, we desire the results in the simplest setting possible.