Theory Comput Syst (2017) 60:299–313
DOI 10.1007/s00224-016-9673-5

A Difference in Complexity Between Recursion and Tail Recursion

Siddharth Bhaskar[1,2]

Published online: 19 March 2016

Abstract There are several ways to understand computability over first-order structures. We may admit functions given by arbitrary recursive definitions, or we may restrict ourselves to "iterative," or *tail recursive*, functions computable by nothing more complicated than while loops. It is well known that in the classical case of recursion theory over the natural numbers, these two notions of computability coincide (though this is not true for all structures). We ask if there are structures over which recursion and tail recursion coincide in terms of computability, but in which a general recursive program may compute a certain function more efficiently than any tail recursion, according to some natural measure of complexity. We give a positive answer to this question, thus answering an open question in Lynch and Blum (Math. Syst. Theory. **12**(1), 205-211 1979).

Keywords Abstract recursion theory · Tail recursion · Lower bounds · Arithmetic

1 Introduction

Let Φ be a signature and $\mathbf{A}$ be a first-order Φ-structure. Any such structure $\mathbf{A}$ has an associated recursion theory, that is to say a family of functions and relations that are recursive over $\mathbf{A}$.

✉ Siddharth Bhaskar
siddhu@bhaskars.com

[1] Department of Mathematics, UCLA, Los Angeles, CA 90095, USA

[2] Present address: Department of Mathematics, IU Bloomington, Bloomington, IN 47405, USA

We say that a function is recursive over $\mathbf{A}$ in case it is computed by a system of recursive equations whose non-logical symbols come from Φ. We call this system a *(McCarthy) recursive program* after John McCarthy, who pioneered the idea in [6]. In the case that $\mathbf{A}$ is

$$\mathbf{N}_u := (\mathbb{N}, 0, 1, S, Pd, =)$$

where S is the successor and Pd the predecessor function, we recover classical recursion theory.

There are several other ways to understand computability over $\mathbf{A}$. For example, we may consider *iterative programs*. A program over a given structure is iterative if, roughly speaking, it can be expressed using recursion no more complicated than "loops," such as for loops or while loops. Iterative programs express a broad class of familiar algorithms, but not the ones that use more complicated "divide-and-conquer" recursion. There is a restricted class of recursive programs, called *tail recursive programs,* which compute exactly the iterative functions.

Over any structure, every tail recursive function is automatically recursive. In other words,

$$\text{tail}(\mathbf{A}) \subseteq \text{rec}(\mathbf{A})$$

for all $\mathbf{A}$, where $\text{tail}(\mathbf{A})$ and $\text{rec}(\mathbf{A})$ are the set of tail recursive and recursive functions over $\mathbf{A}$ respectively.

In the classical setting, the converse is true:

$$\text{tail}(\mathbf{N}_u) = \text{rec}(\mathbf{N}_u).$$

However, this is *not* true in general. Examples of structures for which it fails can be found in [4, 5, 9, 10].

It is natural to ask whether there is a difference in *complexity* between recursion and tail recursion, even when they might agree in terms of computability. In other words:

> Is there a structure $\mathbf{A}$ such that $\text{rec}(\mathbf{A}) = \text{tail}(\mathbf{A})$, some partial function f on $\mathbf{A}$, and some natural measure of efficiency on recursive programs such that there is a recursive program compuing f more efficiently than any tail recursive program?

In this paper we give a positive answer to this question, thus answering an open question from Lynch and Blum [5].

Related Work Several different groups of authors have studied recursion theory over general first-order structures. Each of them draws the distinction between classes corresponding to recursion and tail recursion. In particular, there are:

- *Finite algorithmic procedures* on $\mathbf{A}$ (FAP($\mathbf{A}$)) of Harvey Freidman [1], augmented with a *stack* (FAPS($\mathbf{A}$)) by Moldestad, Stoltenberg-Hansen, and Tucker [3]
- **While**($\mathbf{A}$) and **While***($\mathbf{A}$) of Tucker and Zucker [11]
- *Program schemata* and *recursive schemata* in the field of schematology in computer science. (See, for example, the book of Greibach [2])

Each pair of computability classes corresponds to recursion and tail recursion respectively. In the first two cases, instead of realizing tail recursive programs as a restriction of more general recursive programs, the fundamental object is an abstract register machine or while-program, which corresponds to tail recursion. The full power of recursion is recovered by adding an additional stack or array data structure.

In fact, these models not only compute the same functions as recursive and tail recursive programs, but they do so about as efficiently according to natural measures of complexity (see Section 2.3). Hence our separation results are robust, and not an artifact of our model.

Organization of This Paper In Section 2, we review recursive programs, tail recursion, and complexity measures on recursive programs. Section 3.1 contains the bulk of our technical work and concludes with the abstract theorem about tail recursive programs over the natural numbers which we will use to prove our lower bound, and Section 3.2 applies that theorem to an explicit example. We then make concluding remarks and state some open problems.

2 Recursion over Abstract Structures

In this section we will precisely define our basic objects of study, namely *recursive programs* and *recursive functions over first-order structures*. We follow a simplied version of the exposition in Moschovakis, [7, 8]. First, we review a few preliminaries.

Pointed Structures We require that our structures be *pointed*, i.e., that they contain distinct constants 0 and 1. This simplifies the development of the theory considerably. By identifying boolean values with $\{0, 1\}$ we forego predicates in favor of functions into $\{0, 1\}$, and we do not need to keep track of the *sorts* of terms, which in our setting generalize both terms and atomic formulas in first-order logic.

Expansions If $\mathbf{A}$ is a structure and f is a function on A, then $(\mathbf{A}, f)$ denotes the structure with f as an additional primitive.

In fact, all the structures we will work with will be some expansion of what we have decided to call (for better or for worse) *Predecessor arithmetic,* namely

$$\mathbf{N}_{Pd} := (\mathbb{N}, 0, 1, Pd, eq_0)$$

where

$$eq_0(n) = \begin{cases} 0 & \text{if} n = 0 \\ 1 & \text{otherwise} \end{cases}.$$

2.1 Syntax and Semantics of Recursive Programs

Recursive Programs are the syntactic objects that define recursive functions. *Terms* are the building blocks of recursive programs, and extend terms in first-order logic by function-valued variables and conditional statements.

Terms and Their Semantics

Definition 1 A Φ-*term* M is generated by the grammar

$$M := \mathtt{x} \mid \phi(M_1, \ldots, M_n) \mid \mathtt{p}(M_1, \ldots, M_n) \mid \text{ if } M_0 \text{ then } M_1 \text{ else } M_2$$

where $\mathtt{x}$ is a variable, $\mathtt{p}$ is a function-valued variable or *recursive variable*, and $\phi \in \Phi$. Without loss of generality, assume that variables and function-valued variables are all of the form $\mathtt{x}_i$ or $\mathtt{p}_i$ for $i \in \mathbb{N}$.

A Φ-term without recursive variables is called *explicit*, and an explicit Φ-term without conditionals is called *algebraic*. An algebraic Φ-term is simply a term in the first-order setting; it denotes a function over a Φ-structure $\mathbf{A}$.

Within an explicit term, a conditional

$$\text{if } M_0 \text{ then } M_1 \text{ else } M_2$$

denotes M_1 if the denotation of M_0 is 0 and M_2 if the denotation of M_0 is not 0. Explicit terms also denote functions, and such functions are themselves called *explicit*.

A non-explicit term with recursive variables becomes explicit when we substitute in explicit functions (or partial functions) for those variables. Let the denotation of a non-explicit term be this functional that takes tuples of partial functions to a partial function.

If $\bar{\mathtt{x}}$ and $\bar{\mathtt{p}}$ are tuples of variables and recursive variables respectively, and if for some term M all of its variables and recursive variables occur in $\bar{\mathtt{x}}$ and $\bar{\mathtt{p}}$, then (following Moschovakis) we write $M(\bar{\mathtt{x}}, \bar{\mathtt{p}})$ and call it a *full extended term*.

The advantage of this notation is that it makes substitution of terms very easy to write. For example, given some elements $\bar{\mathtt{x}}$ from a Φ-structure, we may form the *term with parameters* $M(\bar{x}, \bar{\mathtt{p}})$. Or, we might substitute additional Φ-terms for the variables $\bar{\mathtt{x}}$.

We trust that this method of notating substitution is natural to our readers and does not require further explanation.

Anatomy of a Deterministic Recursive Program

Definition 2 A Φ-*recursive program* is a $(K+1)$-tuple of Φ-terms

$$E = (E_0(\bar{\mathtt{x}}_0, \bar{\mathtt{p}}), E_1(\bar{\mathtt{x}}_1, \bar{\mathtt{p}}), \ldots, E_K(\bar{\mathtt{x}}_K, \bar{\mathtt{p}}))$$

where $\bar{\mathtt{p}} = (\mathtt{p}_1, \ldots, \mathtt{p}_K)$ and the arity of $\mathtt{p}_i$ is $|\bar{\mathtt{x}}_i|$ for $1 \le i \le K$.

We shall typically write a program E as

$$E_0(\bar{\mathtt{x}}_0, \bar{\mathtt{p}}) \text{ where}$$
$$\{\mathtt{p}_1(\bar{\mathtt{x}}_1) = E_1(\bar{\mathtt{x}}_1, \bar{\mathtt{p}})$$
$$\vdots$$
$$\mathtt{p}_K(\bar{\mathtt{x}}_K) = E_k(\bar{\mathtt{x}}_K, \bar{\mathtt{p}})\}.$$

The term E_0 is called the *head* of E. The system of equations below the head is the *body* of E.

Denotational Semantics Given a Φ-structure $\mathbf{A}$, a Φ-recursive program E computes a partial function over its domain A:

A *solution* to the body of a recursive equation is a tuple of partial functions $\bar{p}$ on A that is fixed by the functionals denoted by E_i, i.e.,

$$\mathbf{A} \models p_i(\bar{x}) = E_i(\bar{x}, \bar{p})$$

for all $\bar{x}$ and each $1 \leq i \leq K$. [1] Such a solution must always exist (among partial functions!) and moreover there must be a *least* solution $\bar{p}_0$ which is extended by all the others.

Finally, the partial function computed by E over $\mathbf{A}$ is defined to be the partial function denoted by $E_0(\cdot, \bar{p}_0)$; i.e., the partial function obtained when the least solution $\bar{p}_0$ to the body is fed into the head E_0.

For more detail and proofs, see Chapter 2A of Moschovakis [7].

Definition 3 For a Φ-structure $\mathbf{A}$, a partial function is *recursive* if it is computable by some Φ-recursive program E over $\mathbf{A}$. Let $\mathrm{rec}(\mathbf{A})$ be the set of all recursive partial functions over $\mathbf{A}$.

Informal Operational Semantics It will be helpful to informally describe how a human might evaluate a recursive program on a given input. This will also motivate the definition of the *sequential logical complexity* in Section 2.3.

We give a procedure to evaluate a term relative to a fixed Φ-structure $\mathbf{A}$ and Φ-recursive program

$$E = (E_0(\bar{\mathrm{x}}_0, \bar{\mathrm{p}}), E_1(\bar{\mathrm{x}}_1, \bar{\mathrm{p}}), \ldots, E_K(\bar{\mathrm{x}}_K, \bar{\mathrm{p}}))$$

Given a term $M(\bar{x}, \bar{\mathrm{p}})$, proceed according to the following rules:

(Base) If M is a single variable, then $M(\bar{x}, \bar{\mathrm{p}})$ is a single element of A. Return that element.

(Conditional) If $M \equiv$ if M_0 then M_1 else M_2 then evaluate $M_0(\bar{x}, \bar{\mathrm{p}})$. If that is zero, evaluate $M_1(\bar{x}, \bar{\mathrm{p}})$, otherwise evaluate $M_2(\bar{x}, \bar{\mathrm{p}})$, and return the result.

(Primitive-call) If $M \equiv \phi(M_1, \ldots, M_n)$, evaluate $M_i(\bar{x}, \bar{\mathrm{p}})$ for $1 \leq i \leq n$, apply ϕ, and return the result.

(Recursive-call) $M \equiv \mathrm{p}_i(M_1, \ldots, M_n)$, evaluate $M_i(\bar{x}, \bar{\mathrm{p}})$ to obtain y_i for $1 \leq i \leq n$, evaluate $E_i(\bar{y}, \bar{\mathrm{p}})$, and return the result.

For a given input $\bar{x}$, our initial term will be $E_0(\bar{x}, \bar{\mathrm{p}})$. When we have evaluated it, the result will either be an element A, or the procedure will not terminate. This procedure defines a partial function which is identical to the one given by its denotational semantics.

[1] In equality of partial functions, if one side diverges, the other must as well.

2.2 Tail Recursion

Recursive programs which are *tail recursive* express exactly the iterative algorithms over a given structure. The partial functions they compute form an extremely robust class, admitting (as we have seen) several different equivalent definitions.

Tail recursive functions are closed under composition and, more generally, expansions: if f is tail recursive over $(\mathbf{A}, g)$ and g is tail recursive over $\mathbf{A}$, then f is tail recursive over $\mathbf{A}$.

Definition 4 A Φ-recursive program E is *tail recursive* in case it is of the form

$$f(\bar{x}_0) = \mathrm{p}(G(\bar{x}_0)) \text{ where}$$
$$\left\{\mathrm{p}(\bar{x}_1) = \text{if } \tau(\bar{x}_1) \text{ then } o(\bar{x}_1) \text{ else } \mathrm{p}(F(\bar{x}_1))\right\}$$

where G is an explicit term of arity $n = |\bar{x}_0|$ and co-arity [2] $k = |\bar{x}_1|$, F is an explicit Φ-term of arity and co-arity k, and τ and o are explicit terms of arity k.

Definition 5 A partial function $f : A^n \rightharpoonup A$ is *tail recursive* in case it is computable by some tail recursive program over $\mathbf{A}$. The set of all tail recursive partial functions is denoted tail($\mathbf{A}$).

2.3 Complexity Measures on Recursive Programs

We measure the complexity of a recursive program on a particular input by the number of *logical steps* performed in the execution, namely, the total number of times we have to apply a primitive, parse a conditional, or make a recursive call. Following Moschovakis, we call this the *sequential logical complexity.* [3]

For a Φ-structure $\mathbf{A}$ and a Φ-recursive program E, let $M = M(\bar{x}, \bar{\mathrm{p}})$ be a term-with-parameters. The sequential logical calls complexity $L^s(M) = L^s(\mathbf{A}, E, M)$, which measures the number of logical steps it takes to evaluate M, is specified by the following rules. These are easily verified by the informal operational semantics above.

1. If M is a single parameter x, $L^s(M) = 0$.
2. If $M \equiv \phi(M_1, \ldots, M_n)$, then $L^s(M) = 1 + L^s(M_1) + \cdots + L^s(M_n)$.
3. If $M \equiv p_i(M_1, \ldots, M_n)$, then $L^s(M) = 1 + L^s(M_1) + \cdots + L^s(M_n) + L^s(E_i(\bar{M}_1, \ldots, \bar{M}_n))$, where M_i is the denotation of $M_i(\bar{x}, \bar{\mathrm{p}})$.
4. If $M \equiv$ if M_0 then M_1 else M_2, then

$$L^s(M) = \begin{cases} 1 + L^s(M_0) + L^s(M_1) & \text{if} \bar{M}_0 = 0 \\ 1 + L^s(M_0) + L^s(M_2) & \text{otherwise} \end{cases}.$$

[2] A term of co-arity k is simply a k-tuple of terms.

[3] The other natural sequential measure of complexity for recursive programs is the *sequential number of calls complexity,* which measures the number of calls to the primitives during execution. This complexity measure is bounded above and (surprisingly) below by a linear function of the logical complexity (see Chapter 3 of Moschovakis [7]). This tight relationship ensures that our results will apply to both measures.

The number of logical steps it takes to evaluate E on input $\bar{x}$ for a program E is $L^s(\mathbf{A}, E, E_0(\bar{x}, \bar{\mathtt{p}}))$, $E_0(\bar{x}, \bar{\mathtt{p}})$ being of course the first term in the computation. [4] Therefore, define

$$l^s(\mathbf{A}, E, \bar{x}) := L^s(\mathbf{A}, E, E_0(\bar{x}, \bar{\mathtt{p}})).$$

This will be our fundamental measure of complexity.

Sequential Logical Complexity of Tail Recursive Programs If we have a tail recursive program E given by

$$f(\bar{\mathtt{x}}_0) = \mathtt{p}(G(\bar{\mathtt{x}}_0)) \text{ where}$$

$$\left\{\mathtt{p}(\bar{\mathtt{x}}_1) = \text{ if } \tau(\bar{\mathtt{x}}_1) \text{ then } o(\bar{\mathtt{x}}_1) \text{ else } \mathtt{p}(F(\bar{\mathtt{x}}_1))\right\},$$

then the sequential logical complexity of this program on input $\bar{x}$ is linearly related to the number of times the recursive variable $\mathtt{p}$ gets called. We state the following (easy) lemma without proof:

Lemma 1 *For a given $\bar{x}$, let n be the least k such that $\tau(F^k(G(\bar{x}))) = 0$. Then*

$$n \leq l^s(\mathbf{A}, E, \bar{x}) \leq Cn + D$$

for some constants C and D independent of $\bar{x}$.

Tail Recursion, Equivalent Models, and Their Complexity We hypothesize that in any reasonable computing formalism over abstract structures, there will be a natural way of counting the number of logical steps.

For example, a program schema may be thought of as a flowchart whose edges are labelled by *assignment statements,* and whose nodes are labelled by conditionals. Given some input, the computation is modelled by a path through the flowchart. The number of steps in the computation can be thought of as the length of that path.

Under this assumption, our thesis becomes:

> Let $\mathcal{M}$ be a model of computation for abstract structures that computes exactly the tail recursive functions over any structure. Then the number of logical steps it takes to evaluate a function f on input $\bar{x}$ over a structure $\mathbf{A}$ using some instance of $\mathcal{M}$ is bounded below by some linear function of $l^s(\mathbf{A}, E, \bar{x})$, for some tail recursive program E computing f.

This thesis becomes a theorem when we substitute in any concrete model of tail recursive computation, for example finite algorithmic procedures, program schemata, or while-programs. This shows that the lower bounds we demonstrate for tail recursive programs apply more broadly.

[4] L^s diverges exactly when E does.

3 Gaps in Complexity

On the structure $(\mathbf{N}_{Pd}, \gamma)$ the total function

$$f(n, x) = \gamma^{2^n}(x) \tag{1}$$

can be computed by the recursive program E^R:

$$f(n, x) = p(n, x) \text{ where}$$
$$p(n, x) = \begin{cases} \gamma(x) & \text{if } n = 0 \\ p(n-1, p(n-1, x)) & \text{otherwise.} \end{cases}$$

It is easy to show:

Lemma 2 *$l^s((\mathbf{N}_{Pd}, \gamma), E^R, n, x) = O(2^n)$. In other words, there are constants c and d such that E takes at most $d + c2^n$ logical steps on the input (n, x).*

For any function $g : \mathbb{N} \to \mathbb{N}$, we will define γ such that for infinitely many n, there is an x such that computing $f(n, x)$ takes at least $g(n)$ logical steps by any tail recursive program. This will give a positive answer to our main question, i.e., that even when all recursive functions are tail recursive, there can be arbitrarily large gaps in complexity between the two.

3.1 Combinatorics of $(\mathbf{N}_{Pd}, \gamma)$-Tail Recursion

For this section fix a natural number k and assume that γ is non-decreasing.

Definition 6 For tuples $\bar{m}, \bar{n} \in \mathbb{N}^k$ and $a \in \mathbb{N}$, define $\bar{m} \sim_a \bar{n}$ in case $\min(m_i, a) = \min(n_i, a)$ for each $1 \leq i \leq k$.

In other words, $\bar{m} \sim_a \bar{n}$ if the two tuples "agree on parameters less than a." It is clearly an equivalence relation.

It is (reasonably) straightforward to show that:

Lemma 3 *For some $f : \mathbb{N} \to \mathbb{N}$, and $u, v, w, w' \in \mathbb{N}$, suppose $w' - w$ is less than or equal to $f(u) - u$ and $f(v) - v$. Then*

$$\min(u, w) = \min(v, w) \implies \min(f(u), w') = \min(f(v), w').$$

3.1.1 $\sim_a$-Equivalence and $(\mathbf{N}_{Pd}, \gamma)$-Terms

If we imagine for a moment that a were infinity, the relation $\sim_a$ would simply become equality. In this case, the following two statements would be trivially true:

$$\bar{m} \sim_a \bar{n} \implies F(\bar{m}) \sim_a F(\bar{n})$$

if F were a term of arity and co-arity k, and

$$\bar{m} \sim_a \bar{n} \implies F(\bar{m}) = 0 \iff F(\bar{n}) = 0$$

if F were a term of arity k. In the real world, we can still recover analogues to these statements, which are stated in Corollary 1.

Lemma 4 *Suppose F is an explicit $(\mathbf{N}_{Pd}, \gamma)$-term of arity k. Then there is a constant b depending only on F such that for all $a > b$,*

$$\bar{m} \sim_a \bar{n} \implies \min(F(\bar{m}), a - b) = \min(F(\bar{n}), a - b). \tag{2}$$

Proof We may suppose that F is a term in the variables x_1 through x_k, in which case $F(\bar{m})$ is the value of the term obtained when m_i is substituted for x_i. The proof is by induction on the construction of F.

If F is a constant 0 or 1, then the conclusion of equation (2) is trivially satisifed for any b and $a > b$, so we may take b to be 0. If F is the variable x_i, then $F(\bar{m}) = m_i$ and $F(\bar{n}) = n_i$, and $\bar{m} \sim_a \bar{n} \implies \min(m_i, a) = \min(n_i, a)$ by definition, so we may also take b to be 0.

Suppose that F is $\varphi(F')$, where φ is γ or Pd. Then by induction there is some b' such that for all $a > b'$, $\bar{m} \sim_a \bar{n} \implies \min(F'(\bar{m}), a - b') = \min(F'(\bar{n}), a - b')$.

Now suppose that $a > b' + 1$. I claim that

$$\min(F'(\bar{m}), a - b') = \min(F'(\bar{n}), a - b')$$

$$\implies \min(\varphi(F'(\bar{m})), a - (b' + 1)) = \min(\varphi(F'(\bar{n})), a - (b' + 1)) \tag{3}$$

This follows from Lemma 3 with $f = \varphi$, $w' = a - (b' + 1)$, $u = F'(\bar{m})$, and $v = F'(\bar{n})$. The hypotheses of Lemma 3 are satisfied as

$$(a - (b' + 1)) - (a - b') = -1 \leq \varphi(x) - x$$

for any x, when φ is either Pd or γ. □

Hence for $a > b' + 1$, if $\bar{m} \sim_a \bar{n}$, then by induction we have $\min(F'(\bar{m}), a - b') = \min(F'(\bar{n}), a - b')$, and by equation (3) we have $\min(F(\bar{m}), a - (b' + 1)) = \min(F(\bar{n}), a - (b' + 1))$. Hence we may take $b = b' + 1$.

Suppose that F is $eq_0(F')$. By induction, there is some b' such that for all $a > b'$, $\bar{m} \sim_a \bar{n} \implies \min(F'(\bar{m}), a - b') = \min(F'(\bar{n}), a - b')$. Fix $a > b'$ and assume $\bar{m} \sim_a \bar{n}$. Then $\min(F'(\bar{m}), a - b') = \min(F'(\bar{n}), a - b')$, and since $a - b' \geq 1$, we may decrease along the second coordinate (in fact this is a consequence of Lemma 3) to obtain $\min(F'(\bar{m}), 1) = \min(F'(\bar{n}), 1)$. But this says exactly that $eq_0(F'(\bar{m})) = eq_0(F'(\bar{n}))$, and hence (trivially) that $\min(F(\bar{m}), a - b) = \min(F(\bar{n}), a - b)$. We have shown that

$$\forall a > b\, \bar{m} \sim_a \bar{n} \implies \min(F(\bar{m}), a - b') = \min(F(\bar{n}), a - b'),$$

so we may take $b = b'$.

Finally, suppose that F is the conditional statement "if F_0 then F_1 else F_2." By induction, there are b_0, b_1, and b_2, such that for $i \in \{0, 1, 2\}$, if $a > b_i$ and $\bar{m} \sim_a \bar{n}$, we have $\min(F_i(\bar{m}), a - b_i) = \min(F_i(\bar{n}), a - b_i)$.

Let b be $\max\{b_0, b_1, b_2\}$ and suppose $a > b$ and $\bar{m} \sim_a \bar{n}$. We then want to show that $\min(F(\bar{m}), a - b) = \min(F(\bar{n}), a - b)$.

Since $a > b_0$, $\min(F_0(\bar{m}), a - b_0) = \min(F_0(\bar{n}), a - b_0)$. Since $a - b_0 \geq 1$, we can decrease the second coordinate to obtain $\min(F_0(\bar{m}), 1) = \min(F_0(\bar{n}), 1)$, and hence $F_0(\bar{m}) = 0 \iff F_0(\bar{n}) = 0$.

Suppose that $F_0(\bar{m}) = F_0(\bar{n}) = 0$. Then $F(\bar{m}) = F_1(\bar{m})$ and $F(\bar{n}) = F_1(\bar{n})$. Since $a > b_1$, $\min(F_1(\bar{m}), a - b_1) = \min(F_1(\bar{n}), a - b_1)$, so $\min(F(\bar{m}), a - b_1) = \min(F(\bar{n}), a - b_1)$. Finally, since $b \geq b_1$, $a - b \leq a - b_1$, and decreasing along the second coordinate, we get $\min(F(\bar{m}), a - b) = \min(F(\bar{n}), a - b)$.

The case that $F_0(\bar{m}), F_0(\bar{n}) > 0$ follows similar lines.

Corollary 1 *Suppose F is an explicit $(\mathbf{N}_{Pd}, \gamma)$-term of arity k. Then there is a constant b depending only on F such that for all $a > b$,*

$$\bar{m} \sim_a \bar{n} \implies eq_0(F(\bar{m})) = eq_0(F(\bar{n})).$$

Suppose G is an explicit $(\mathbf{N}_{Pd}, \gamma)$-term of arity and co-arity k. Then there is a constant b depending only on F such that for all $a > b$,

$$\bar{m} \sim_a \bar{n} \implies G(\bar{m}) \sim_{a-b} G(\bar{n}).$$

Proof By Lemma 4, there is some b such that if $\bar{m} \sim_a \bar{n}$ and $a > b$, $\min(F(\bar{m}), a - b) = \min(F(\bar{n}), a - b)$. As noted in the proof of that lemma, this statement when $a = b + 1$ is equivalent to $eq_0(F(\bar{m})) = eq_0(F(\bar{n}))$.

To prove the second statement, suppose that $G = (G_1, \ldots, G_k)$, where each G_i is a term of arity k. Then by Lemma 4, for each $1 \leq i \leq k$ there is a b_i such that if $a > b_i$ and $\bar{m} \sim_a \bar{n}$, $\min(G_i(\bar{m}), a - b_i) = \min(G_i(\bar{n}), a - b_i)$.

Let $b = \max\{b_i : 1 \leq i \leq k\}$, and suppose $a > b$ and $\bar{m} \sim_a \bar{n}$. Then for each i, $\min(G_i(\bar{m}), a - b_i) = \min(G_i(\bar{n}), a - b_i)$, and hence, decreasing along the second coordinate, $\min(G_i(\bar{m}), a-b) = \min(G_i(\bar{n}), a-b)$. But this is exactly the statement that $G(\bar{m}) \sim_{a-b} G(\bar{n})$. □

3.1.2 $\sim_a$-equivalence and $(\mathbf{N}_{Pd}, \gamma)$-tail recursion

For the remainder of Section 3.1, fix a tail recursive program E

$$f(\mathtt{x}_1, \ldots, \mathtt{x}_n) = \mathtt{p}(G(\mathtt{x}_1, \ldots, \mathtt{x}_n)) \text{ where}$$
$$\{\mathtt{p}(\bar{\mathtt{x}}) = \text{if } \tau(\bar{\mathtt{x}}) \text{ then } o(\bar{\mathtt{x}}) \text{ else } \mathtt{p}(F(\bar{\mathtt{x}}))\} \tag{4}$$

where $\bar{\mathtt{x}} = (\mathtt{x}_1, \ldots, \mathtt{x}_k)$, G, τ, o, and F are explicit $(\mathbf{N}_{Pd}, \gamma)$-terms, and G and F have co-arity k.

What we will do is examine the tuples of the form $F^j(G(\bar{x}))$ such that $\tau(F^j(G(\bar{x}))) \neq 0$, which are the tuples we evaluate $\mathtt{p}$ upon in the course of the computation on input $\bar{x}$. If some tuple is repeated, i.e., $F^i(G(\bar{x})) = F^j(G(\bar{x}))$ for some $i < j$, then the computation fails to halt.

In this section we obtain an analogue of this result when equality is replaced by $\sim_a$-equivalence.

By Corollary 1 applied to τ and F, there are constants b and c such that for any $a > b, c$,

$$\bar{m} \sim_a \bar{n} \implies eq_0(\tau(\bar{m})) = eq_0(\tau(\bar{n})) \tag{5}$$

and

$$\bar{m} \sim_a \bar{n} \implies F(\bar{m}) \sim_{a-b} F(\bar{n}). \tag{6}$$

Corollary 2 *Fix $\bar{m}$ and $\bar{n}$, and suppose $\bar{m} \sim_{ab+c} \bar{n}$ for some $a > 0$. Then $\tau(F^j(\bar{m})) = 0 \iff \tau(F^j(\bar{n})) = 0$ for $0 \le j < a$.*

Proof Fix $0 \le j < a$. Then $F^j(\bar{m}) \sim_{(a-j)b+c} F^j(\bar{n})$: when $j = 0$, it is equivalent to $\bar{m} \sim_{ab+c} \bar{n}$, and when $j > 0$, it follows from equation (6). (We take F^0 to be the identity function.)

By equation (5), since $(a - j)b + c > c$, we have that $eq_0(\tau(F^j(\bar{m}))) = eq_0(\tau(F^j(\bar{n})))$, or in other words $\tau(F^j(\bar{m})) = 0 \iff \tau(F^j(\bar{n})) = 0$. □

Definition 7 For $\bar{m} \in \mathbb{N}^k$ and $u < v \in \mathbb{N}$, define

$$\mathfrak{G}(\bar{m}, u, v) \iff \{m_1, \ldots, m_k\} \cap (u, v) = \emptyset.$$

(Here (u, v) is an interval, i.e. the set $\{n \in \mathbb{N} : u < n < v\}$.)

Then we have immediately that if $\mathfrak{G}(\bar{m}, u, v)$, $\mathfrak{G}(\bar{n}, u, v)$ and $\bar{m} \sim_u \bar{n}$ then $\bar{m} \sim_v \bar{n}$.

Lemma 5 *Fix $\bar{m}$. Suppose that $\bar{m} \sim_{ab+c} F^i(\bar{m})$ for some $a, i > 0$. Then the least j such that $\tau(F^j(\bar{m})) = 0$, if it exists, cannot be in the interval $[i, i + a)$.*

Proof By Corollary 2, for $0 \le j < a$,

$$\tau(F^j(\bar{m})) = 0 \iff \tau(F^{i+j}(\bar{m})) = 0.$$

Therefore if for some $i \le j < a + i$, $\tau(F^j(\bar{m})) = 0$, then $\tau(F^{j-i}(\bar{m})) = 0$. □

Lemma 6 *Fix $\bar{m}$. Suppose that there exists such a j such that $\tau(F^j(\bar{m})) = 0$, and let j_0 be the least such j. Suppose that $\mathfrak{G}(F^j(\bar{m}), a, a'b + c)$ for some $a > 0$, $a' \ge a$, and for all $j \le (a + 1)^k$. Then j_0 cannot be in the interval $((a + 1)^k, a')$.*

Proof We first claim that $F^{n_1}(\bar{m}) \sim_a F^{n_2}(\bar{m})$ for some $0 \le n_1 < n_2 \le (a + 1)^k$. Otherwise each tuple $\{F^j(\bar{m}) : 0 \le j \le (a+1)^k\}$ is in its own $\sim_a$-equivalence class. However, there are only $(a + 1)^k$ different $\sim_a$-equivalence classes: for each of the k indices, there are $a + 1$ choices, one for each $\{0, 1, \ldots, a - 1\}$, and one for numbers $\ge a$.

Hence $F^{n_1}(\bar{m}) \sim_a F^{n_2}(\bar{m})$ for some $0 \le n_1 < n_2 \le (a + 1)^k$. By assumption, $\mathfrak{G}(F^{n_1}(\bar{m}), a, a'b + c)$ and $\mathfrak{G}(F^{n_2}(\bar{m}), a, a'b + c)$, so $F^{n_1}(\bar{m}) \sim_{a'b+c} F^{n_2}(\bar{m})$. By Lemma 5, the least j such that $\tau(F^{n_1+j}(\bar{m})) = 0$ cannot be in the interval $[n_2 - n_1, n_2 - n_1 + a')$. But this least j is $j_0 - n_1$; therefore, $j_0 - n_1 \notin [n_2 - n_1, n_2 - n_1 + a')$ and $j_0 \notin [n_2, n_2 + a')$. Since $((a + 1)^k, a') \subset [n_2, n_2 + a')$, j_0 is not contained in $((a + 1)^k, a')$. □

This result may seem technical but the philosophy behind it is quite clear:

Suppose the elements of a tuple $\bar{m}$ contain very big or very small numbers. Then the computation of $\mathfrak{p}$ *on* $\bar{m}$ *halts in either a very short or very long amount of time.*

3.2 An Explicit Complexity Gap

We will now apply the results of Section 3.1 to any tail recursive program computing the function f of equation (1). We shall show that, for any increasing function g, we can define γ such that any tail recursive program E computing f over $(\mathbf{N}_{Pd}, \gamma)$ makes at least $g(n)$ logical steps on infinitely many inputs (n, x). (Recall that only $O(2^n)$ logical steps were required by a general recursive program, for any γ.)

We first describe how to obtain γ from g. Namely, let $\gamma : \mathbb{N} \to \mathbb{N}$ be any strictly increasing function for which for every function $L : x \mapsto ux + v$ for $u, v \in \mathbb{N}$, the interval $(n, 2^n + L(g(n)))$ is disjoint with the range of γ for arbitrarily large n. The fact that there exists such a function requires proof.

Lemma 7 *For every increasing function $g : \mathbb{N} \to \mathbb{N}$ there is an increasing function $\gamma : \mathbb{N} \to \mathbb{N}$ such that for all $u, v \in \mathbb{N}$ there exist infinitely many n such that $(n, 2^n + g(n)u + v)$ is disjoint with the range of γ.*

Proof We define γ by recursion. Let $\gamma(0)$ be 0, and define

$$\gamma(i+1) := 1 + \max\{2^{\gamma(i)+1} + g(\gamma(i)+1)u + v \mid u, v \leq i\},$$

so that for $n = \gamma(i) + 1$, $\gamma(i) < n < 2^n + g(n)u + v < \gamma(i+1)$ for all $u, v \leq i$. Clearly γ is increasing.

Now fix $u, v \in \mathbb{N}$. Then for all $i \geq u, v$ if $n = \gamma(i) + 1$, then $(n, 2^n + g(n)u + v)$ is disjoint from the range of γ. This is because $\gamma(i) < n, 2^n + g(n)u + v < \gamma(i+1)$, and γ is increasing so there is no j such that $\gamma(i) < \gamma(j) < \gamma(i+1)$. □

Having defined γ from g, we suppose that E is a tail recursive program computing f over $(\mathbf{N}_{Pd}, \gamma)$, and we now describe the infinite family of inputs on which E runs slowly. Let $L(n)$ be the linear function $nb + c$, where b and c are obtained from E as in Section 3.1.2. By definition, $(n, 2^n + L(g(n)))$ is disjoint with the range of γ for arbitrarily large n. Let $n_1 < n_2 < \ldots$ be an infinite increasing sequence witnessing this, and let $x_i := 2^{n_i} + L(g(n_i))$. The pairs (n_i, x_i) for $i \in \mathbb{N}$ will be the infinitely many inputs on which E will perform at least $g(n_i)$ logical steps.

For all i, j, let $\bar{m}_{i,j} := F^j(G(n_i, x_i))$. The $\bar{m}_{i,j}$ are the tuples that $\mathfrak{p}$ gets called upon in the course of the computation on input (n_i, x_i). Then

Lemma 8 *For $0 \leq j < 2^{n_i}$, $\mathfrak{G}(\bar{m}_{i,j}, n_i, L(g(n_i)))$.*

Proof Let $\bar{m}_{i,j} = (m_1^{(i,j)}, \ldots, m_k^{(i,j)})$. Let m range over $m_\ell^{(i,j)}$ for $i \in \mathbb{N}$, $0 \leq j < 2^{n_i}$ and $1 \leq \ell \leq k$. Then $m = b(y)$ where b is an algebraic $\{Pd, \gamma, eq_0\}$-term of

length less than 2^{n_i} and y is 0, 1, n_i, or x_i. I claim that all such m are contained in the set

$$[0, n_i] \cup [x_i - 2^{n_i}, x_i] \cup \bigcup_{\vartheta \in \gamma[\mathbb{N}]} [\vartheta - 2^{n_i}, \vartheta].$$

In case b is a $\{Pd\}$-term, this is obvious. Otherwise $b = Pd^j \circ \gamma \circ b'$ or $b = Pd^j \circ eq_0 \circ b'$ for some $j < 2^{n_i}$ and algebraic $\{Pd, \gamma, eq_0\}$-term b'. Define $m' := b'(y)$. In the first case, $m = \gamma(m') - j$, so $m \in [\vartheta - 2^{n_i}, \vartheta]$ for some ϑ in the range of γ. In the second case, m is 0 or 1.

Since $x_i - 2^{n_i} = L(g(n_i))$, each parameter m is contained in the set

$$[0, n_i] \cup [L(g(n_i)), L(g(n_i)) + 2^{n_i}] \cup \bigcup_{\vartheta \in \gamma[\mathbb{N}]} [\vartheta - 2^{n_i}, \vartheta].$$

However, if we take the intersection with $(n_i, L(g(n_i)))$,

$$[0, n_i] \cap (n_i, L(g(n_i))) = \emptyset$$

$$[L(g(n_i)), L(g(n_i)) + 2^{n_i}] \cap (n_i, L(g(n_i))) = \emptyset.$$

If $z \in (n_i, L(g(n_i))) \cap [\vartheta - 2^{n_i}, \vartheta]$ for some $\vartheta \in \gamma[\mathbb{N}]$, then

$$n_i < z \leq \vartheta \leq z + 2^{n_i} < L(g(n_i)) + 2^{n_i}.$$

This contradicts the assumption that the range of γ and $(n_i, L(g(n_i)) + 2^{n_i})$ are disjoint for all n_i. □

We can now prove our main theorem. The proof uses the notion of *value-depth complexity* (see Chapter 4 in Moschovakis [7]). In our case, the value-depth complexity of a natural number y given natural numbers $(x_1, \ldots, x_k)$ is the length of the shortest term T such that $(\mathbf{N}_{Pd}, \gamma) \models y = T(x_i)$ for some $1 \leq i \leq k$. The value-depth complexity of y given $\bar{x}$ is a lower bound for the number of logical steps a program E takes to compute input $\bar{x}$ if $(\mathbf{N}_{Pd}, \gamma) \models E(\bar{x}) = y$. [5]

Theorem 1 *For sufficiently large i, the number of sequential logical calls made by the tail recursive program E on input (n_i, x_i) is at least $g(n_i)$. I.e.,*

$$l^s((\mathbf{N}_{Pd}, \gamma), E, n_i, x_i) \geq g(n_i).$$

Proof By Lemma 8 and the definition of L, for all i,

$$\mathfrak{G}(\bar{m}_{i,j}, n_i, g(n_i)b + c)$$

for $0 \leq j < 2^{n_i}$, and hence for $0 \leq j \leq (n_i + 1)^k$ if i is sufficiently large. If $j_{0,i}$ is the least j such that $\tau(\bar{m}_{i,j}) = 0$, we can apply Lemma 6 to conclude that $j_{0,i}$ is not in the interval $((n_i + 1)^k, g(n_i))$ for sufficiently large i.

Moreover, $j_{0,i}$ grows exponentially in n_i. This is because the value-depth complexity of $\gamma^{2^{n_i}}(x_i)$ given n_i and x_i is 2^{n_i}, and so 2^{n_i} is a lower bound for the number

[5] The intuition here is that E needs to "construct" the term defining y from $\bar{x}$ in the course of its computation, and it can increase the length of this term by at most one symbol per logical step.

of calls to γ. But $j_{0,i}$ is bounded below by a linear function of the number of logical steps by Lemma 1.

Therefore, for sufficiently large i, $j_{0,i} > (n_i+2)^k$, so $j_{0,i} \geq g(n_i)$. In other words, on input (n_i, x_i), the tail recursion (4) makes at least $g(n_i)$ recursive calls before halting. But by Lemma 1, this shows that the number of logical steps is also at least $g(n_i)$. □

4 Conclusion

For each increasing function g, we have exhibited a structure $(\mathbf{N}_{Pd}, \gamma)$ and a γ-recursive function f such that for each tail recursive program E computing f, there is an infinite family of inputs $\{(n_i, x_i) : i \in \mathbb{N}\}$ such that $n_i < n_{i+1}$ for each i and the number of logical steps in the computation $E(n_i, x_i)$ is at least $g(n_i)$. On the other hand, the recursive program E always makes $O(2^n)$ logical steps on all inputs (n, x), which is optimal within a linear factor.

Hence we have found structures with arbitrarily large complexity gaps. On the other hand, with an increasing function γ we can compute the successor by a tail recursion, so in terms of computability there is no difference between recursion and tail recursion over $(\mathbf{N}_{Pd}, \gamma)$. Therefore, we have a positive answer to our original question.

We might imagine some ways to strengthen this specific result. For example, we believe that there should be a difference in complexity when we count just the number of times γ is called, instead of the total number of logical steps. This would be somewhat cleaner. It would also be nice to show that there is an infinite family on inputs on which *all* tail recursive programs run slowly, instead of having the family of inputs depend on the program.

5 Future Work

More generally, there would be several additional types of functions where it would be desirable to find complexity gaps. Namely, we would like to find gaps among unary functions and relations, i.e., functions into {0, 1}. The usual reductions (using pairing functions and the graph relation) fail to preseve complexity measures in the right way when applied to this example.

What would be most interesting is a difference in complexity over a standard structure of the natural numbers, like $\mathbf{N}_u$ or the related structure of *binary string arithmetic*. Since Turing machines and other concrete machine models of computation are naturally expressed by tail recursive programs over these structures, such a complexity difference might imply that such machines cannot faithfully express all algorithms without *some* loss in overhead. This has foundational importance, since the standard way to formalize the complexity of algorithms uses Turing machines.

A final open question is whether we can find a complexity difference lurking among usual computational problems–searching, sorting, and the like. For example, among pure "comparison sorts," the ones which are naturally expressed by

iterative algorithms like *selection sort* and *insertion sort* seem to use $O(n^2)$ comparisons on lists of length n, whereas a general recursive algorithm like *mergesort* uses $O(n \log n)$ comparisons.

We might conjecture that in a sufficiently abstract model of sorting, we might be able to show a gap in the number of comparsions between recursion and tail recursion. Unfortunately, this is false as stated, but we wonder whether there is anything in this spirit.

Acknowledgments This article developed out of the author's Ph.D. thesis, which was completed under the helpful guidance of Yiannis N. Moschovakis.

References

1. Friedman, Mansfield: Algorithmic procedures. Trans. Am. Math. Soc. **332**, 297–312 (1992)
2. Sheila, A.: Greibach. Theory of program structures: Schemes, Semantics, Verification. Springer, Verlag (1975)
3. Stoltenberg-Hansen, V., Moldestad, J., Tucker, J.V.: Finite algorithmic procedures and computation theories. Math. Scand. **46**, 77–94 (1980)
4. Kfoury, A.J., Stolboushkin, A.P.: An infinite pebble game and applications. Inf Comput **136**, 53–66 (1997)
5. Lynch, N.A., Blum, E.K.: A difference in expressive power between flowcharts and recursion schemes. Math. Syst Theory **12**(1), 205–211 (1979)
6. McCarthy, J.: A basis for a mathematical theory of computation. In: proceedings of the Western Joint Computer Conference, pp. 225–238 (1961)
7. Moschovakis, Y.N.: Recursion and complexity. Lecture notes, UCLA (2015)
8. Moschovakis, Y.N., van den Dries, L.: Arithmetic complexity. ACM Trans. Comput. Log. **10** (2009)
9. Paterson, M.S., Hewitt, C.E.: Comparative schematology. In: Proc. Rec. ACM Conference on Concurrent Systems and Parallel Computation, pp. 119–127 (1970)
10. Tiuryn, J.: A simplified proof of ddl $<$ dl. Inf. Comput. **81**(1), 1–12 (1989)
11. Tucker, J.V., Zucker, J.I.: Computable functions and semicomputable sets on many-sorted algebras, chapter 5, pages 397–525 Oxford University Press (2000)