Pointers in Recursion: Exploring the Tropics

Paulin Jacobé de Naurois

Pointers in Recursion: Exploring the Tropics

Paulin Jacobé de Naurois
CNRS, Université Paris 13, Sorbonne Paris Cité, LIPN, UMR 7030, F-93430 Villetaneuse, France.
denaurois@lipn.univ-paris13.fr

Abstract. We translate the usual class of partial/primitive recursive functions to a pointer recursion framework, accessing actual input values via a pointer reading unit-cost function. These pointer recursive functions classes are proven equivalent to the usual partial/primitive recursive functions. Complexity-wise, this framework captures in a streamlined way most of the relevant sub-polynomial classes. Pointer recursion with the safe/normal tiering discipline of Bellantoni and Cook corresponds to polylogtime computation. We introduce a new, non-size increasing tiering discipline, called tropical tiering. Tropical tiering and pointer recursion, used with some of the most common recursion schemes, capture the classes logspace, logspace/polylogtime, ptime, and NC. Finally, in a fashion reminiscent of the safe recursive functions, tropical tiering is expressed directly in the syntax of the function algebras, yielding the tropical recursive function algebras.

Introduction

Characterizing complexity classes without explicit reference to the computational model used for defining these classes, and without explicit bounds on the resources allowed for the calculus, has been a long term goal of several lines of research in computer science. One rather successful such line of research is recursion theory. The foundational work here is the result of Cobham [7], who gave a characterization of polynomial time computable functions in terms of bounded recursion on notations - where, however, an explicit polynomial bound is used in the recursion scheme. Later on, Leivant [11] refined this approach with the notion of tiered recursion: explicit bounds are no longer needed in his recursion schemes. Instead, function arguments are annotated with a static, numeric denotation, a *tier*, and a tiering discipline is imposed upon the recursion scheme to enforce a polynomial time computation bound. A third important step in this line of research is the work of Bellantoni and Cook [2], whose safe recursion scheme uses only syntactical constraints akin to the use of only two tier values, to characterize, again, the class of polynomial time functions.

Cobham's approach has also later on been fruitfully extended to other, important complexity classes. Results relevant to our present work, using explicitly bounded recursion, are those of Lind [15] for logarithmic space, and Allen [1] and Clote [6] for small parallel classes.

Later on, Bellantoni and Cook's purely syntactical approach proved also useful for characterizing other complexity classes. Leivant and Marion [14, 13] used a predicative version of the safe recursion scheme to characterize alternating complexity classes, while Bloch [3], Bonfante et al [4] and Kuroda[10], gave characterizations of small, polylogtime, parallel complexity classes. An important feature of these results is that they use, either explicitly or not, a tree-recursion on the input. This tree-recursion is implicitly obtained in Bloch's work by the use of an extended set of basic functions, allowing for a dichotomy recursion on the input string, while it is made explicit in the recursion scheme in the two latter works. As a consequence, these characterizations all rely on the use of non-trivial basic functions, and non-trivial data structures. Moreover, the use of distinct basic function sets and data structures make it harder to express these charcterizations in a uniform framework.

Among all these previous works on sub-polynomial complexity classes, an identification is assumed between the argument of the functions of the algebra, on one hand, and the computation input on the other hand: an alternating, logspace computation on input $\overline{x}$ is denoted by a recursive function with argument $\overline{x}$. While this seems very natural for complexity classes above linear time, it actually yields a fair amount of technical subtleties

and difficulties for sub-linear complexity classes. Indeed, following Chandra et al. [5] seminal paper, sub-polynomial complexity classes need to be defined with a proper, subtler model than the one-tape Turing machine: the random access Turing machine (RATM), where computation input is accessed via a unit-cost pointer reading instruction. RATM input is thus accessed via a read-only instruction, and left untouched during the computation - a feature quite different to that of a recursive function argument. Our proposal here is to use a similar construct for reading the input in the setting of recursive functions: our functions will take as input pointers on the computation input, and one-bit pointer reading will be assumed to have unit cost. Actual computation input are thus implicit in our function algebras: the fuel of the computational machinery is only pointer arithmetics. This proposal takes inspiration partially from the Rational Bitwise Equations of [4].

Following this basic idea, we then introduce a new tiering discipline, called *tropical tiering*, to enforce a non-size increasing behavior on our recursive functions, with some inspirations taken from previous works of M. Hofmann [8, 9]. Tropical tiering induces a polynomial interpretation in the tropical ring of polynomials (hence its name), and yields a characterization of logarithmic space. The use of different, classical recursion schemes yield characterizations of other, sub-polynomial complexity classes such as polylogtime, NC, and the full polynomial time class. Following the approach of Bellantoni and Cook, we furthermore embed the tiering discipline directly in the syntax, with only finitely many different tier values - four tier values in our case, instead of only two tier values for the safe recursive functions, and provide purely syntactical characterizations of these complexity classes in a unified, simple framework. Compared to previous works, our framework uses a unique, and rather minimal set of unit-cost basic functions, computing indeed basic tasks, and a unique and also simple data structure. Furthermore, while the syntax of the tropical composition and recursion schemes may appear overwhelming at first sight, it has the nice feature, shared with the safe recursion functions of [2], of only adding a fine layer of syntactic sugar over the usual composition and primitive recursion schemes. Removing this sugar allows to retrieve the classical schemes. In that sense, we claim our approach to be simpler and than the previous ones of [3, 4, 10] .

The paper is organized as follows. Section 1 introduces the notations, and the framework of pointer recursion. Section 2 applies this framework to primitive recursion. Pointer partial/primitive recursive functions are proven to coincide with their classical counterparts in Theorem 2. Section 3 applies this framework to safe recursion on notations. Pointer safe recursive functions are proven to coincide with polylogtime computable functions in Theorem 3. Tropical tiering is defined in Section 4. Proposition 4 establishes the tropical interpretation induced by tropical tiering. Tropical recursive functions are then introduced in Subsection 4.3. Section 5 gives a sub-algebra of the former, capturing logspace/polylogtime computable functions in Theorem 9. Finally, Section 6 explores tropical recursion with substitutions, and provides a characterization of P in Theorem 11 and of NC in Theorem 13.

1 Recursion

1.1 Notations, and Recursion on Notations

Data structures considered in our paper are finite words over a finite alphabet. For the sake of simplicity, we consider the finite, boolean alphabet $\{\mathbf{0}, \mathbf{1}\}$. The set of finite words over $\{\mathbf{0}, \mathbf{1}\}$ is denoted as $\{\mathbf{0}, \mathbf{1}\}^*$.

Finite words over $\{\mathbf{0}, \mathbf{1}\}$ are denoted with overlined variables names, as in $\overline{x}$. Single values in $\{\mathbf{0}, \mathbf{1}\}$ are denoted as plain variables names, as in x. The empty word is denoted

by ε, while the dot symbol "." denotes the concatenation of two words as in $a.\overline{x}$, the finite word obtained by adding an a in front of the word $\overline{x}$. Finally, finite arrays of boolean words are denoted with bold variable names, as in $\mathbf{x} = (\overline{x_1}, \cdots, \overline{x_n})$. When defining schemes, we will often omit the length of the arrays at hand, when clear from context, and use bold variable names to simplify notations. Similarly, for mutual recursion schemes, finite arrays of mutually recursive functions are denoted by a single bold function name. In this case, the *width* of this function name is the size of the array of the mutually recursive functions.

Natural numbers are identified with finite words over $\{\mathbf{0}, \mathbf{1}\}$ via the usual binary encoding. Yet, in most of our function algebras, recursion is not performed on the numerical value of an integer, as in classical primitive recursion, but rather on its boolean encoding, that is, on the finite word over $\{\mathbf{0}, \mathbf{1}\}$ identified with it: this approach is denoted as *recursion on notations*.

1.2 Turing Machines with Random Access

When considering sub-polynomial complexity class, classical Turing Machines often fail to provide a suitable cost model. A crucial example is the class DLOGTIME: in logarithmic time, a classical Turing machine fails to read any further than the first $k.\log(n)$ input bits. In order to provide a suitable time complexity measure for sub-polynomial complexity classes, Chandra et al [5] introduced the Turing Machine with Random Access (RATM), whose definition follows.

▶ **Definition 1.** *RATM*
A Turing Machine with Random Access (RATM) is a Turing machine with no input head, one (or several) working tapes and a special pointer *tape, of logarithmic size, over a binary alphabet. The Machine has a special* Read *state such that, when the binary number on the pointer tape is k, the transition from the Read state consists in writing the k^{th} input symbol on the (first) working tape.*

1.3 Recursion on Pointers

In usual recursion theory, a function computes a value on its input, which is given explicitly as an argument. This, again, is the case in classical primitive recursion. While this is suitable for describing explicit computation on the input, as, for instance for single tape Turing Machines, this is not so for describing input-read-only computation models, as, for instance, RATMs. In order to propose a suitable recursion framework for input-read-only computation, we propose the following *pointer recursion* scheme, whose underlying idea is pretty similar to that of the RATM.

As above, recursion data is given by finite, binary words, and the usual recursion on notation techniques on these recursion data apply. The difference lies in the way the actual computation input is accessed: in our framework, we distinguish two notions, the *computation input*, and the *function input*: the former denotes the input of the RATM, while the latter denotes the input in the function algebra. For classical primitive recursive functions, the two coincide, up to the encoding of integer into binary strings. In our case, we assume an explicit encoding of the former into the latter, given by the two following constructs.

Let $\overline{w} = w_1.\cdots.w_n \in \{\mathbf{0}, \mathbf{1}\}^*$ be a computation input. To $\overline{w}$, we associate two constructs,

- the `Offset`: a finite word over $\{\mathbf{0}, \mathbf{1}\}$, encoding in binary the length n of $\overline{w}$, and
- the `Read` construct, a 1-ary function, such that, for any binary encoding $\overline{i}$ of an integer $0 < i \leq n$, $\texttt{Read}(\overline{i}) = w_i$, and, for any other value $\overline{v}$, $\texttt{Read}(\overline{v}) = \varepsilon$.

Then, for a given *computation input* $\overline{w}$, we fix accordingly the semantics of the `Read` and `Offset` constructs as above, and a *Pointer Recursive function* over $\overline{w}$ is evaluated with sole input the `Offset`, accessing computation input bits via the `Read` construct. For instance, under these conventions, `Read(hd(Offset))` outputs the first bit of the computational input $\overline{w}$. In some sense, the two constructs depend on $\overline{w}$, and can be understood as functions on $\overline{w}$. However, in our approach, it is important to forbid $\overline{w}$ from appearing explicitly as a function argument in the syntax of the function algebras we will define, and from playing any role in the composition and recursion schemes. Since $\overline{w}$ plays no role at the syntactical level - its only role is at the semantical level- we chose to remove it completely $\overline{w}$ from the syntactical definition of our functions algebras.

2 Pointers Primitive Recursion

Let us first detail our pointer recursive framework for the classical case of primitive recursion on notations.

2.0.0.1 Basic pointer functions.

Basic pointer functions are the following kind of functions:

1. Functions manipulating finite words over $\{\mathbf{0},\mathbf{1}\}$. For any $a \in \{\mathbf{0},\mathbf{1}\}, \overline{x} \in \{\mathbf{0},\mathbf{1}\}^*$,

$$\begin{array}{rclrclrcl} \mathtt{hd}(a.\overline{x}) & = & a & \mathtt{tl}(a.\overline{x}) & = & \overline{x} & \mathtt{s}_0(\overline{x}) & = & 0.\overline{x} \\ \mathtt{hd}(\varepsilon) & = & \varepsilon & \mathtt{tl}(\varepsilon) & = & \varepsilon & \mathtt{s}_1(\overline{x}) & = & 1.\overline{x} \end{array}$$

2. Projections. For any $n \in \mathbb{N}$, $1 \leq i \leq n$,

$$\mathtt{Pr}_i^n(\overline{x_1},\cdots,\overline{x_n}) = \overline{x_i}$$

3. and, finally, the `Offset` and `Read` constructs, as defined above.

2.0.0.2 Composition.

Given functions g, and $h_1,\cdots,h_n$, we define f by composition as

$$f(\mathbf{x}) = g(h_1(\mathbf{x}),\cdots,h_n(\mathbf{x})).$$

2.0.0.3 Primitive Recursion on Notations.

Let $\perp$ denote non-terminating computation. Given functions h, g_0 and g_1, we define f by primitive recursion on notations as

$$\begin{array}{rcl} f(\varepsilon,\mathbf{y}) & = & h(\mathbf{y}) \\ f(\mathtt{s}_a(\overline{x}),\mathbf{y}) & = & \begin{cases} g_a(\overline{x},f(\overline{x},\mathbf{y}),\mathbf{y}) & \text{if } f(\overline{x},\mathbf{y}) \neq \perp \\ \perp & \text{otherwise.} \end{cases} \end{array}$$

2.0.0.4 Minimization.

For a function s, denote by $s^{(n)}$ its n^{th} iterate. Then, given a function h, we define f by minimization on $\overline{x}$ as

$$\mu\overline{x}(h(\overline{x},\mathbf{y})) = \begin{cases} \bot & \text{if } \forall t \in \mathbb{N}, \mathtt{hd}(h(s_0^{(t)}(\varepsilon),\mathbf{y})) \neq \mathtt{s}_1(\varepsilon) \\ s_0^{(k)}(\varepsilon) & \text{where } k = \min\{t \ : \mathtt{hd}(h(s_0^{(t)}(\varepsilon),\mathbf{y})) = \mathtt{s}_1(\varepsilon)\} \quad \text{otherwise.} \end{cases}$$

In other words, a function f defined by minimization on h produces the shortest sequence of 0 symbols satisfying a simple condition on h , if it exists.

Let now PR_{not}^{point} be the closure of basic pointer functions under composition and primitive recursion on notations, and REC_{not}^{point} be the closure of basic pointer functions under composition, primitive recursion on notations, and minimization. Then, as expected,

▶ **Theorem 2.** *Modulo the binary encoding of natural integers, PR_{not}^{point} is the classical class of primitive recursive functions, and REC_{not}^{point} is the classical class of recursive functions.*

Proof. It is already well known that primitive recursive functions on notations are the classical primitive recursive functions, and recursive functions on notations are the classical recursive functions. Now, for one direction, it suffices to express the `Read` and `Offset` basic pointer functions as primitive recursive functions on the computation input. For the other direction, it suffices to reconstruct with pointer primitive recursion the computation input from the `Read` and `Offset` basic pointer functions.

3 Pointer Safe Recursion

We recall the tiering discipline of Bellantoni and Cook [2]: functions arguments are divided into two tiers, *normal* arguments and *safe* arguments. Notation-wise, both tiers are separated by a semicolon symbol in a block of arguments, the normal arguments being on the left, and the safe arguments on the right. We simply apply this tiering discipline to our pointer recursion framework.

3.0.0.1 Basic Pointer Safe Functions.

Basic pointer safe functions are the basic pointer functions of the previous section, all their arguments being considered safe.

3.0.0.2 Safe Composition.

Safe composition is somewhat similar to the previous composition scheme, with a tiering discipline, ensuring that safe arguments cannot be moved to a normal position in a function call. The reverse however is allowed.

$$f(\mathbf{x};\mathbf{y}) = g(h_1(\mathbf{x};),\cdots,h_m(\mathbf{x};);h_{m+1}(\mathbf{x};\mathbf{y}),\cdots,h_{m+n}(\mathbf{x};\mathbf{y})).$$

Calls to functions h_{m+i}, where safe arguments are used, are placed in safe position in the argument block of g. A special case of safe composition is $f(\overline{x};\overline{y}) = g(;\overline{x},\overline{y})$, where a normal argument $\overline{x}$ is used in safe position in a call. Hence, we liberally use normal arguments in safe position, when necessary.

3.0.0.3 Safe Recursion.

The recursion argument is normal. The recursive call is placed in safe position, a feature that prevents nesting recursive calls exponentially.

$$\begin{aligned} f(\varepsilon, \mathbf{y}; \mathbf{z}) &= h(\mathbf{y}; \mathbf{z}) \\ f(a.\overline{x}, \mathbf{y}; \mathbf{z}) &= g_a(\overline{x}, \mathbf{y}; f(\overline{x}, \mathbf{y}; \mathbf{z}), \mathbf{z}). \end{aligned}$$

Let now SR^{point}_{not} be the closure of the basic pointer safe functions under safe composition and safe recursion.

▶ **Theorem 3.** *SR^{point}_{not} is the class $DTIME(polylog)$ of functions computable in polylogarithmic time.*

Proof. The proof is essentially the same as for the classical result by Bellantoni and Cook [2]. Here however, it is crucial to use the RATM as computation model. Simulating a polylogtime RATM with safe recursion on pointers is very similar to simulating a polytime TM with safe recursion - instead of explicitly using the machine input as recursion data, we use the size of the input as recursion data, and access the input values via the `Read` construct, exactly as is done by the RATM model. The other direction is also similar: the tiering discipline of the safe recursion on pointers enforces a polylog bound on the size of the strings (since the initial recursion data - the `Offset` - has size logarithmic in the size n of the computation input), and thus a polylog bound on the computation time.

4 Tropical Tiering

We present here another, stricter tiering discipline, that we call *tropical Tiering*. The adjective "tropical" refers to the fact that this tiering induces a polynomial interpretation in the tropical ring of polynomials. This tiering discipline takes some inspiration from Hofmann's work on non-size increasing types [8], and pure pointer programs [9]. The idea however is to use here different tools than Hofmann's to achieve a similar goal of bounding the size of the function outputs. We provide here a non-size increasing discipline via the use of tiering, and use it in the setting of pointer recursion to capture not only pure pointer programs (Hoffman's class), but rather pointer programs with pointer arithmetics, which is in essence the whole class Logspace.

4.0.0.1 Basic Pointer Functions.

We add the following numerical successor basic function. Denote by $E : \mathbb{N} \to \{\mathbf{0}, \mathbf{1}\}^*$ the usual binary encoding of integers, and $D : \{\mathbf{0}, \mathbf{1}\}^* \to \mathbb{N}$ the decoding of binary strings to integers. Then,

$$\mathtt{s}(\overline{x}) = E(D(\overline{x}) + 1)$$

denotes the numerical successor on binary encodings, and, by convention, ε is the binary encoding of the integer 0.

4.0.0.2 Primitive Recursion on Values.

Primitive recursion on values is the usual primitive recursion, encoded into binary strings:

$$\begin{aligned} f(\varepsilon, \mathbf{y}) &= h(\mathbf{y}) \\ f(\mathtt{s}(\overline{x}), \mathbf{y}) &= g(\overline{x}, f(\overline{x}, \mathbf{y}), \mathbf{y}). \end{aligned}$$

4.1 Tropical Tier

As usual, tiering consists in assigning function variables to different classes, called tiers. In our setting, these tiers are identified by a numerical value, called *tropical tier*, or, shortly, *tropic*. The purpose of our tropical tiers is to enforce a strict control on the increase of the size of the binary strings during computation. Tropics take values in $\mathbb{Z} \cup \{-\infty\}$. The tropic of the i^{th} variable of a function f is denoted $T_i(f)$. The intended meaning of the tropics is to provide an upper bound on the linear growth of the function output size with respect to the corresponding input size, as per Proposition 4. Tropics are inductively defined as follows.

1. Basic pointer functions:

$$\begin{array}{rclcrclcrcl} T_{j\neq i}(\mathtt{Pr}_i^n) & = & -\infty & & T_1(\mathtt{hd}) & = & -\infty & & T_1(\mathtt{Read}) & = & -\infty \\ T_1(\mathtt{tl}) & = & -1 & & & & & & & & \\ T_i(\mathtt{Pr}_i^n) & = & 0 & & & & & & & & \\ T_1(\mathtt{s}_0) & = & 1 & & T_1(\mathtt{s}_1) & = & 1 & & T_1(\mathtt{s}) & = & 1 \end{array}$$

2. Composition:
$$T_t(f) = \max_i\{T_i(g) + T_t(h_i)\}.$$

3. Primitive recursion on notations. Two cases arise:
   - $T_2(g_0) \leq 0$ and $T_2(g_1) \leq 0$. In that case, we set
     a. $T_1(f) = \max\{T_1(g_0), T_1(g_1), T_2(g_0), T_2(g_1)\}$, and,
     b. for all $t \geq 1$,
        $T_t(f) = \max\{T_{t+1}(g_0), T_{t+1}(g_1), T_{t-1}(h), T_2(g_0), T_2(g_1)\}$.
   - the previous case above does not hold, $T_2(g_0) \leq 1$, and $T_2(g_1) \leq 1$. In that case, we also require that $T_1(g_0) \leq 0$, $T_1(g_1) \leq 0$, and, for all $t \geq 2$, $T_t(g_0) = T_t(g_1) = T_{t-2}(h) = -\infty$. Then, we set $T_1(f) = \max\{T_1(g_0), T_1(g_1), T_2(g_0) - 1, T_2(g_1) - 1, c_h\}$, where c_h is a constant for h given in Proposition 4 below, and, for $t \geq 1$, $T_t(f) = -\infty$.

   Other cases than the two above do not enjoy tropical tiering.
4. Primitive recursion on values. Only one case arises:
   - $T_2(g) \leq 0$. In that case, we set
     a. $T_1(f) = \max\{T_1(g), T_2(g)\}$, and,
     b. for all $t \geq 1$, $T_t(f) = \max\{T_{t+1}(g), T_{t-1}(h), T_2(g)\}$.

   Again, other cases than the one above do not enjoy tropical tiering.

Furthermore, when using tropical tiering, we use mutual recursion schemes. For $\mathbf{f} = (f_1, \cdots, f_n)$, mutual primitive recursion (on values) is classically defined as follows,

$$\begin{array}{rcl} \mathbf{f}(\varepsilon, \mathbf{y}) & = & \mathbf{h}(\mathbf{y}) \\ \mathbf{f}(\mathtt{s}(\overline{x}), \mathbf{y}) & = & \begin{cases} \mathbf{g}(\overline{x}, \mathbf{f}(\overline{x}, \mathbf{y}), \mathbf{y}) & \text{if } \forall i\ (f_i(\overline{x}, \mathbf{y}) \neq \bot) \\ \bot & \text{otherwise.} \end{cases} \end{array}$$

and similarly for mutual primitive recursion on notations. Tropical tiering is then extended to mutual primitive recursion in a straightforward manner.

We define the set of L-primitive pointer recursive functions as the closure of the basic pointer functions of Sections 2 and 4 under composition, (mutual) primitive recursion on notations and (mutual) primitive recursion on values, with tropical tiering.

4.2 Tropical Interpretation

Tropical tiering induces a non-size increasing discipline. More formally,

▶ **Proposition 4.** *The tropical tiering of a L-primitive recursive function f induces a polynomial interpretation of f on the tropical ring of polynomials, as follows.*

For any L-primitve recursive function f with n arguments, there exists a constant $c_f \geq 0$ such that

$$|f(\overline{x_1}, \cdots, \overline{x_n})\ | \leq \max_t\{T_t(f) + |\overline{x_t}|, c_f\}.$$

PROOF. The proof is given for non-mutual recursion schemes, by induction on the definition tree. Mutual recursion schemes follow the same pattern.

1. For basic pointer functions, the result holds immediately.
2. Let f be defined by composition, and assume that the result holds for the functions g, $h_1, \cdots, h_n$. Then, for any $i = 1, \cdots, n$, $|h_i(\mathbf{x})\ | \leq \max_t\{T_t(h_i) + |\overline{x_t}|, c_{h_i}\}$. Moreover, there exists by induction c_g such that $|g(h_1(\mathbf{x}), \cdots, h_n(\mathbf{x}))\ | \leq \max_i\{T_i(g) + |h_i(\mathbf{x})\ |, c_g\}$. Composing the inequalities above yields $|g(h_1(\mathbf{x}), \cdots, h_n(\mathbf{x}))\ | \leq \max_i\{T_i(g) + \max_t\{T_t(h_i) + |\overline{x_i}|, c_{h_i}\}, c_g\} = \max_t\{T_t(f) + |\overline{x_t}|, \max_i\{c_{f_i}, c_g\}\}$.
3. Let f be defined by primitive recursion on notations, and assume that the first case holds. Let $f(a.\overline{x}, \mathbf{y}) = g_a(\overline{x}, f(\overline{x}, \mathbf{y}), \mathbf{y})$, for $a \in \{0, 1\}$, and assume $T_2(g_0) \leq 0$ and $T_2(g_1) \leq 0$. We apply the tropical interpretation on g, and we show by induction the result for f on the length of $a.\overline{x}$.
   a. If $\max_{\overline{x}, f(\overline{x}, \mathbf{y}), t}\{|\overline{x}| + T_1(g_a), |f(\overline{x}, \mathbf{y})| + T_2(g_a), |\overline{y_t}| + T_{t+2}(g_a), c_{g_a}\} = |\overline{x}| + T_1(g_a)$: $|f(a.\overline{x}, \mathbf{y})| \leq |\overline{x}| + T_1(g_a) \leq |\overline{x}| + T_1(f)$, and the result holds.
   b. If $\max_{\overline{x}, f(\overline{x}, \mathbf{y}), t}\{|\overline{x}| + T_1(g_a), |f(\overline{x}, \mathbf{y})| + T_2(g_a), |\overline{y_t}| + T_{t+2}(g_a), c_{g_a}\} = |f(\overline{x}, \mathbf{y})| + T_2(g_a)$: Since $T_2(g_a) \leq 0$, $|f(a.\overline{x}, \mathbf{y})| \leq |f(\overline{x}, \mathbf{y})|$, and the induction hypothesis applies.
   c. If $\max_{\overline{x}, f(\overline{x}, \mathbf{y}), t}\{|\overline{x}| + T_1(g_a), |f(\overline{x}, \mathbf{y})| + T_2(g_a), |\overline{y_t}| + T_{t+2}(g_a), c_{g_a}\} = |\overline{y_t}| + T_{t+2}(g_a)$ for some t: the result applies immediately by structural induction on g_a.
   d. If $\max_{\overline{x}, f(\overline{x}, \mathbf{y}), t}\{|\overline{x}| + T_1(g_a), |f(\overline{x}, \mathbf{y})| + T_2(g_a), |\overline{y_t}| + T_{t+2}(g_a), c_{g_a}\} = c_{g_a}$, the result holds immediately.
   e. The base case $f(\epsilon, \mathbf{y})$ is immediate.
4. Let f be defined by primitive recursion on notations, and assume now that the second of the two corresponding cases holds. Let $f(a.\overline{x}, \mathbf{y}) = g_a(\overline{x}, f(\overline{x}, \mathbf{y}), \mathbf{y})$, for $a \in \{0, 1\}$. Since the first case does not hold, $T_2(g_0) = 1$ or $T_2(g_1) = 1$: assume that $T_2(g_0) = 1$ (the other case being symmetric). Assume also that, $T_1(g_0) \leq 0$ and $T_1(g_1) \leq 0$, and for all $t \geq 2$, $T_t(g_0) = T_t(g_1) = T_{t-2}(h) = -\infty$. Then, we set $T_1(f) = \max\{0, c_h\}$. We apply the tropical interpretation on g, and prove by induction on the length of $a.\overline{x}$ that $|f(a.\overline{x}, \mathbf{y})| \leq |a.\overline{x}| + \max\{c_{g_1}, c_{g_2}, c_h\}$.
   a. If $\max_{t \geq 2}\{|\overline{x}| + T_1(g_a), |f(\overline{x}, \mathbf{y})| + T_2(g_a), |\overline{y_t}| + T_t(g_a), c_{g_a}\} = |\overline{x}| + T_1(g_a)$. Since $T_1(g_a) \leq 0$ and $T_1(f) \geq 0$, $|f(a.\overline{x}, \mathbf{y})| \leq |\overline{x}| \leq T_1(f) + |\overline{x}|$, and the result holds.
   b. If $\max_{t \geq 2}\{|\overline{x}| + T_1(g_a), |f(\overline{x}, \mathbf{y})| + T_2(g_a), |\overline{y_t}| + T_t(g_a), c_{g_a}\} = |f(\overline{x})| + T_2(g_a)$. Since $T_2(g_a) \leq 1$, $|f(a.\overline{x})| \leq 1 + |f(\overline{x})|$, and the induction hypothesis allows to conclude.
   c. If $\max_{t \geq 2}\{|\overline{x}| + T_1(g_a), |f(\overline{x}, \mathbf{y})| + T_2(g_a), |\overline{y_t}| + T_t(g_a), c_{g_a}\} = c_{g_a}$, the result holds immediately.
   d. The case $\max_{t \geq 2}\{|\overline{x}| + T_1(g_a), |f(\overline{x}, \mathbf{y})| + T_2(g_a), |\overline{y_t}| + T_t(g_a), c_{g_a}\} = |\overline{y_t}| + T_t(g_a)$ is impossible since $T_t(g_a) = -\infty$ for $t \geq 2$.
   e. The base case $f(\epsilon, \mathbf{y})$ is immediate.
5. Let now assume f is define by primitive recursion on values. Then, the only possible case is similar to the first case of primitive recursion on notation.

The proof by induction above emphasizes the critical difference between recursion on notation and recursion on values: the second case of the safe recursion on notations correspond to the linear, non-size increasing scanning of the input, as in, for instance,

$$f(a.\overline{x}) = s_a(f(\overline{x})).$$

This, of course, is only possible in recursion on notation, where the height of the recursive calls stack is precisely the length of the scanned input. Recursion on values fails to perform this linear scanning, since, for a given recursive argument $\overline{x}$, the number of recursive calls is then exponential in $|\overline{x}|$.

Proposition 4 proves that the tropical tiering of a function yields actually a tropical polynomial interpretation for the function symbols: The right hand side of the Lemma inequality is indeed a tropical interpretation. Moreover, this interpretation is directly given by the syntax.

Furthermore, the proof also highlights why we use mutual recursion schemes instead of more simple, non-mutual ones: non-size increasing discipline forbids the use of multiplicative constants in the size of the strings. So, in order to capture a computational space of size $k.\log(n)$, we need to use k binary strings of length $\log(n)$, defined by mutual recursion.

▶ **Corollary 5.** *L-primitive pointer recursive functions are computable in logarithmic space.*

Proof.

Proposition 4 ensures that the size of all binary strings is logarithmically bounded. A structural induction on the definition of f yields the result. The only critical case is that of a recursive construct. When evaluating a recursive construct, one needs simply to store all non-recursive arguments (the $\overline{y_i}$'s) in a shared memory, keep a shared counter for keeping track of the recursive argument $\overline{x}$, and use a simple `while` loop to compute successively all intermediate recursive calls leading to $f(\overline{x}, \mathbf{y})$. All these shared values have logarithmic size. The induction hypothesis ensures then that, at each step in the `while` loop, all computations take logarithmic space. The two other cases, composition and basic functions, are straightforward.

In the following section, we prove the converse: logarithmic space functions can be computed by a sub-algebra of the L-primitive pointer recursive functions.

4.3 Tropical Recursion

In this section we restrict our tropical tiering approach to only four possible tier values: 1, 0, -1 and $-\infty$. While doing so, we still retrain the same expressiveness. The rules for tiering are adapted accordingly. More importantly, the use of only four tier values allows us to denote these tropics directly in the syntax, in an approach similar to that of Bellantoni and Cook, by adding purely syntactical features to the composition and primitive recursion schemes. Let us take as separator symbol the following $\wr$ symbol, with leftmost variables having the highest tier. As with safe recursive functions, we allow the use of a high tier variable in a low tier position, as in, for instance,

$$f(\overline{x} \wr \overline{y} \wr \overline{z} \wr \overline{t}) = g(\ \wr \overline{y} \wr \overline{x}, \overline{z} \wr \overline{t}).$$

Our tropical recursive functions are then as follows.

4.3.0.1 Basic tropical pointer functions.

Basic tropical pointer functions are the following.

$$\begin{array}{rclrcl}
\mathtt{hd}(\wr\ \wr\ \wr\, a.\overline{x}) & = & a & \mathtt{tl}(\wr\ \wr\, a.\overline{x}\,\wr) & = & \overline{x} \\
\mathtt{hd}(\wr\ \wr\ \wr\, \varepsilon) & = & \varepsilon & \mathtt{tl}(\wr\ \wr\, \varepsilon\,\wr) & = & \varepsilon \\
\mathtt{s}_0(\overline{x}\,\wr\ \wr\ \wr) & = & 0.\overline{x} & \mathtt{s}_1(\overline{x}\,\wr\ \wr\ \wr) & = & 1.\overline{x} \\
\mathtt{s}(\overline{x}\,\wr\ \wr\ \wr) & = & E(D(\overline{x})+1) & \mathtt{Read}(\wr\ \wr\ \wr\,\overline{x}) & = & a \in \{0,1\} \\
 & & & \mathtt{Pr}^n_i(\wr\,\overline{x_i}\,\wr\ \wr\,\overline{x_1},\cdots,\overline{x_{i-1}},\overline{x_{i+1}},\cdots,\overline{x_n}) & = & \overline{x_i}
\end{array}$$

4.3.0.2 Tropical composition.

Define $\mathbf{t} = \mathbf{t}_1, \mathbf{t}_2, \mathbf{t}_3, \mathbf{t}_4$. The tropical composition scheme is then

$$\begin{array}{rcl}
f(\mathbf{x}\wr\mathbf{y}\wr\mathbf{z}\wr\mathbf{t}) & = & g(h_1(\wr\,\mathbf{x}\wr\mathbf{y}\wr\mathbf{t}),\cdots,h_a(\wr\,\mathbf{x}\wr\mathbf{y}\wr\mathbf{t})\wr \\
 & & h_{a+1}(\mathbf{x}\wr\mathbf{y}\wr\mathbf{z}\wr\mathbf{t}),\cdots,h_b(\mathbf{x}\wr\mathbf{y}\wr\mathbf{z}\wr\mathbf{t})\wr \\
 & & h_{b+1}(\mathbf{y}\wr\mathbf{z}\wr\ \wr\mathbf{t}),\cdots,h_c(\mathbf{y}\wr\mathbf{z}\wr\ \wr\mathbf{t})\wr \\
 & & h_{c+1}(\mathbf{t}_1\wr\mathbf{t}_2\wr\mathbf{t}_3\wr\mathbf{t}_4),\cdots,h_d(\mathbf{t}_1\wr\mathbf{t}_2\wr\mathbf{t}_3\wr\mathbf{t}_4))
\end{array}$$

4.3.0.3 Tropical Recursion on Notations - case 1.

$$\begin{array}{rcl}
\mathbf{f}(\mathbf{x}\wr\varepsilon,\mathbf{y}\wr\mathbf{z}\wr\mathbf{t}) & = & \mathbf{h}(\mathbf{x}\wr\mathbf{y}\wr\mathbf{z}\wr\mathbf{t}) \\
\mathbf{f}(\mathbf{x}\wr\mathtt{s}_a(\overline{r}\wr\ \wr\ \wr),\mathbf{y}\wr\mathbf{z}\wr\mathbf{t}) & = & \mathbf{g}_a(\mathbf{x}\wr\overline{r},\mathbf{f}(\mathbf{x}\wr\overline{r},\mathbf{y}\wr\mathbf{z}\wr\mathbf{t}),\mathbf{y}\wr\mathbf{z}\wr\mathbf{t})
\end{array}$$

4.3.0.4 Tropical Recursion on Notations - case 2. (Linear scanning)

$$\begin{array}{rcl}
\mathbf{f}(\wr\,\varepsilon\wr\ \wr\mathbf{t}) & = & \varepsilon \\
\mathbf{f}(\wr\,\mathtt{s}_a(\overline{r}\wr\ \wr\ \wr)\wr\ \wr\mathbf{t}) & = & \mathbf{g}_a(\mathbf{f}(\wr\,\overline{r}\wr\ \wr\mathbf{t})\wr\,\overline{r}\wr\ \wr\mathbf{t})
\end{array}$$

4.3.0.5 Tropical Recursion on Values.

$$\begin{array}{rcl}
\mathbf{f}(\mathbf{x}\wr\varepsilon,\mathbf{y}\wr\mathbf{z}\wr\mathbf{t}) & = & \mathbf{h}(\mathbf{x}\wr\mathbf{y}\wr\mathbf{z}\wr\mathbf{t}) \\
\mathbf{f}(\mathbf{x}\wr\mathtt{s}(\overline{r}\wr\ \wr\ \wr),\mathbf{y}\wr\mathbf{z}\wr\mathbf{t}) & = & \mathbf{g}(\mathbf{x}\wr\overline{r},\mathbf{f}(\mathbf{x}\wr\overline{r},\mathbf{y}\wr\mathbf{z}\wr\mathbf{t}),\mathbf{y}\wr\mathbf{z}\wr\mathbf{t})
\end{array}$$

As above, we use the mutual version of these recursion schemes, with the same tiering discipline. Note that, unlike previous characterizations of sub-polynomial complexity classes [3, 4, 10], our tropical composition and recursion schemes are only syntactical refinements of the usual composition and primitive recursion schemes - removing the syntactical sugar yields indeed the classical schemes.

▶ **Definition 6.** *L-tropical functions*
The class of L-tropical functions is the closure of our basic tropical pointer functions, under tropical composition, tropical mutual recursion on notations, and tropical mutual recursion on values.

The restriction of only four tier values suffices to capture the computational power of RATMs. More precisely,

▶ **Theorem 7.** *The class of L-tropical functions is the class of functions computable in logarithmic space, with logarithmic size output.*

Proof. L-tropical functions are L- primitive pointer recursive functions with tropics 1, 0, -1 and $-\infty$. Following Corollary 5, they are computable in logspace. The converse follows from the simulation of a logarithmic space RATM, given in the Appendix. The simulation works as follows.

4.3.0.6 Encoding the machine configurations.

Assume the machine M works in space $k\lceil\log(n+1)\rceil$. A configuration of M is then encoded by $2k+3$ binary strings of length less than $\lceil\log(n+1)\rceil$:

1. one string, of constant length, encodes the machine state,
2. one string, of length $\lceil\log(n+1)\rceil$, encodes the pointer tape,
3. one string, of length $\lceil\log(n+1)\rceil$, encodes the head of the pointer tape. It contains **0** symbols everywhere, but on the position of the head (where it contains a **1**).
4. k strings, of length $\lceil\log(n+1)\rceil$, encode the content of the work tape, and
5. k strings, of length $\lceil\log(n+1)\rceil$, encode the position of the work tape head, with (as for the pointer tape) **0** everywhere but on the position of the head.

4.3.0.7 Reading and Updating a configuration.

Linear scanning of the recursive argument in tropical recursion, corresponding to case 2 of the definition of tropical recursion on notations, is used to read and to update the encoding of the configuration. In order to do so, one defines L-tropical functions for

1. encoding boolean values `true` and `false`, boolean connectives, and `if then else` constructs,
2. scanning an input string until a **1** is found, and computing the corresponding prefix sequence,
3. computing left and right extractions of sub-strings of a string, for a given prefix,
4. replacing exactly one bit in a binary string, whose position is given by a prefix of the string.

These functions are given in the Appendix. With all these simple bricks, and especially with the in-place one-bit replacement, one is then able to read a configuration, and to update it, with L-tropical functions. None of these L-tropical functions uses recursion on values.

4.3.0.8 Computing the Transition map of the Machine.

Given the functions above, the transition map **Next** of the machine is then computed by a simple L-tropical function of width $(2k+3)$: For a recursive argument $\overline{s}$ of size $\lceil\log(n+1)\rceil$, $\mathbf{Next}(\wr\,\overline{s}, \mathbf{c}\,\wr\,\wr)$ computes the configuration reached from $\mathbf{c}$ in one transition step.

4.3.0.9 Simulating the RATM.

The simulation of the RATM is then obtained by iterating its transition map a suitable number of times. The time upper bound is here obtained by nesting k tropical recursive functions on values: on an input of size $\lceil \log(n+1) \rceil$, the unfolding of these recursive calls takes time n^k. At each recursive step, this function needs to apply the transition map. The transition map having width $(2k+3)$, we use here a mutual recursion scheme, of width $(2k+3)$. Again, for a recursive argument $\overline{s}$ of size $\lceil \log(n+1) \rceil$, we define

$$\begin{aligned}
\mathbf{Step}_1(\wr\, \varepsilon, \overline{s}, \mathbf{c} \,\wr\, \wr) &= \mathbf{c} \\
\mathbf{Step}_1(\wr\, \mathbf{s}(\overline{t} \,\wr\, \wr\, \wr), \overline{s}, \mathbf{c} \,\wr\, \wr) &= \mathbf{Next}(\wr\, \overline{s}, \mathtt{Step_1}(\wr\, \overline{\mathbf{t}}, \overline{\mathbf{n}}, \mathbf{c} \,\wr\, \wr) \,\wr\, \wr) \\
\mathbf{Step}_2(\wr\, \varepsilon, \overline{s}, \mathbf{c} \,\wr\, \wr) &= \mathbf{c} \\
\mathbf{Step}_2(\wr\, \mathbf{s}(\overline{t} \,\wr\, \wr\, \wr), \overline{s}, \mathbf{c} \,\wr\, \wr) &= \mathbf{Step}_1(\wr\, \overline{s}, \overline{s}, \mathbf{Step}_2(\wr\, \overline{t}, \overline{s}, \mathbf{c} \,\wr\, \wr) \,\wr\, \wr) \\
&\vdots \\
\mathbf{Step}_k(\wr\, \varepsilon, \overline{s}, \mathbf{c} \,\wr\, \wr) &= \mathbf{c} \\
\mathbf{Step}_k(\wr\, \mathbf{s}(\overline{t} \,\wr\, \wr\, \wr), \overline{s}, \mathbf{c} \,\wr\, \wr) &= \mathbf{Step}_{k-1}(\wr\, \overline{s}, \overline{s}, \mathbf{Step}_k(\wr\, \overline{t}, \overline{s}, \mathbf{c} \,\wr\, \wr) \,\wr\, \wr).
\end{aligned}$$

Replacing $\overline{s}$ by the $\mathtt{Offset}$ in the above gives the correct bounds. Finally, one simply needs to use simple L-tropical functions for computing the initial configuration, and reading the final configuration.

5 Logarithmic Space, Polylogarithmic Time

▶ **Definition 8.** *LP-tropical functions*
The class of LP-tropical functions is the closure of our basic tropical pointer functions, under tropical composition and tropical mutual recursion on notations.

▶ **Theorem 9.** *The class of LP-tropical functions is the class of functions computable in logarithmic space, polylogarithmic time, with logarithmic size output.*

Proof. Mutual recursion on notations, with recursive arguments of logarithmic size, are computable in polylogarithmic time, following similar arguments as in the proof of Theorem 7. The converse follows from the simulation in the proof of Theorem 7 above, where mutual recursion on values for the functions $\mathtt{Step}_i$ is replaced by mutual recursion on notations.

6 Alternation

In this section we extend the approach of Leivant and Marion [12] to our setting. Let us define a similar tropical recursion on notations with substitutions. Note that the tropical tiering discipline prevents using substitutions in case 2 of the tropical recursion on notations. Substitutions are therefore only defined for case 1 of this recursion scheme.

6.0.0.1 Tropical Recursion with substitutions on Notations.

Given functions h, g_0, g_1, k_1 and k_2,

$$\begin{aligned}
\mathbf{f}(\mathbf{x} \wr \varepsilon, \overline{u}, \mathbf{y} \wr \mathbf{z} \wr \mathbf{t}) &= \mathbf{h}(\mathbf{x} \wr \overline{u}, \mathbf{y} \wr \mathbf{z} \wr \mathbf{t}) \\
\mathbf{f}(\mathbf{x} \wr \mathbf{s}_a(\overline{r} \wr\, \wr\, \wr), \mathbf{y} \wr \overline{u}, \mathbf{z} \wr \mathbf{t}) &= \mathbf{g}_a(\mathbf{x} \wr \overline{r}, \mathbf{f}(\mathbf{x} \wr \overline{r}, k_1(\wr\, \overline{u} \wr\, \wr), \mathbf{y} \wr \mathbf{z} \wr \mathbf{t}), \\
&\qquad \mathbf{f}(\mathbf{x} \wr \overline{r}, k_2(\wr\, \overline{u} \wr\, \wr), \mathbf{y} \wr \mathbf{z} \wr \mathbf{t}), \mathbf{y} \wr \mathbf{z} \wr \mathbf{t}) \ .
\end{aligned}$$

6.0.0.2 Tropical Recursion with substitutions on Values.

Given functions h, g, k_1 and k_2,

$$\begin{aligned}\mathbf{f}(\mathbf{x} \wr \varepsilon, \overline{u}, \mathbf{y} \wr \mathbf{z} \wr \mathbf{t}) &= \mathbf{h}(\mathbf{x} \wr \overline{u}, \mathbf{y} \wr \mathbf{z} \wr \mathbf{t}) \\ \mathbf{f}(\mathbf{x} \wr \mathbf{s}(\overline{r} \wr \ \wr \ ), \mathbf{y} \wr \overline{u}, \mathbf{z} \wr \mathbf{t}) &= \mathbf{g}(\mathbf{x} \wr \overline{r}, f(\mathbf{x} \wr \overline{r}, k_1(\wr \overline{u} \wr \ \wr \ ), \mathbf{y} \wr \mathbf{z} \wr \mathbf{t}), \\ &\quad \mathbf{f}(\mathbf{x} \wr \overline{r}, k_2(\wr \overline{u} \wr \ \wr \ ), \mathbf{y} \wr \mathbf{z} \wr \mathbf{t}), \mathbf{y} \wr \mathbf{z} \wr \mathbf{t}) \ .\end{aligned}$$

Again, as above, we assume these recursion schemes to be mutual.

▶ **Definition 10.** *P-tropical functions*
The class of P-tropical functions is the closure of our basic tropical pointer functions, under tropical composition, tropical recursion on notations and on values, and tropical recursion with substitutions on notations and on values.

▶ **Theorem 11.** *The class of P-tropical functions with binary output is the class P.*

Proof. The result follows from *Alogspace* $= P$ [5], and Theorem 7. Substitutions in the tropical recursion scheme on notations amounts to alternation. Restriction to decision classes instead of function classes comes from the use of alternating Turing machines, which compute only decision problems.

Let us first see how to simulate a logspace alternating machine with P-tropical functions. Recall the notations and functions of the proof of Theorem 7. Since we now need to simulate a non-deterministic, alternating machine, we assume without loss of generality that we now have two kinds of machine states:

- non-deterministic universal
- non-deterministic existential

and that non-deterministic transitions have at most two branches. Therefore, we also assume that we have one predicate that determines the kind of a state in a configuration $\mathbf{c}$: $\mathtt{IsUniversal}(\wr \overline{s}, \mathbf{c} \wr \ \wr \ )$. This predicate is assumed to output `false` or `true`.

We also assume that we have two transition maps, $\mathtt{Next}_0(\wr \overline{s}, \mathbf{c} \wr \ \wr \ )$, and $\mathtt{Next}_1(\wr \overline{s}, \mathbf{c} \wr \ \wr \ )$, for computing both branches of non-deterministic transitions. For deterministic transitions, we assume both branches are the same. Finally, we also assume we have a predicate $\mathtt{isPositive}(\wr \overline{s}, \mathbf{c} \wr \ \wr \ )$, which returns `true` if the configuration $\mathbf{c}$ is final and accepting, and `false` otherwise.

We define now, with substitutions, the following:

$$\begin{aligned}\mathtt{Accept}(\wr \varepsilon, \overline{s}, \mathbf{c} \wr \ \wr \ ) &= \mathtt{isPositive}(\wr \overline{s}, \mathbf{c} \wr \ \wr \ ) \\ \mathtt{Accept}(\wr \mathbf{s}(\overline{t}), \overline{s}, \mathbf{c} \wr \ \wr \ )) &= \mathtt{match\ IsUniversal}(\wr \overline{s}, \mathbf{c} \wr \ \wr \ )\ \mathtt{with} \\ &\ |\mathtt{true} -> \mathtt{AND}\ (\wr \mathtt{Accept}(\wr \overline{t}, \mathtt{Next}_0(\wr \overline{s}, \mathbf{c} \wr \ \wr \ ), \mathbf{c} \wr \ \wr \ ), \\ &\qquad\qquad \mathtt{Accept}(\wr \overline{t}, \mathtt{Next}_1(\wr \overline{s}, \mathbf{c} \wr \ \wr \ )) \wr \ \wr \ ) \\ &\ |\mathtt{false} -> \mathtt{OR}\ (\wr \mathtt{Accept}(\wr \overline{t}, \mathtt{Next}_0(\wr \overline{s}, \mathbf{c} \wr \ \wr \ )), \mathbf{c} \wr \ \wr \ ), \\ &\qquad\qquad \mathtt{Accept}(\wr \overline{t}, \mathtt{Next}_1(\wr \overline{s}, \mathbf{c} \wr \ \wr \ )) \wr \ \wr \ ) \ .\end{aligned}$$

Then, for $\overline{t}$ and $\overline{s}$ large enough, and an initial configuration $\mathbf{c}$, $\mathtt{Accept}(\wr \overline{t}, \overline{s}, \mathbf{c} \wr \ \wr \ )$ outputs the result of the computation of the machine. Finally, nesting up to k layers of such recursion on values schemes allows, as in the proof of Theorem 7, to simulate a polynomial computation time.

The other direction is pretty straightforward: For any instance of a recursion scheme with substitutions, for any given values $\overline{r}$, $\overline{u}$, $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{z}$, each bit of $\mathbf{g}_a(\mathbf{x} \wr \overline{r}, \mathbf{f}(\mathbf{x} \wr \overline{r}, k_1(\wr \overline{u} \wr \ \wr \ ), \mathbf{y} \wr \mathbf{z} \wr \mathbf{t}), \mathbf{f}(\mathbf{x} \wr \overline{r}, k_2(\wr \overline{u} \wr \ \wr \ ), \mathbf{y} \wr \mathbf{z} \wr \mathbf{t}), \mathbf{y} \wr \mathbf{z} \wr \mathbf{t})$ is a boolean function of the bits of $\mathbf{f}(\mathbf{x} \wr \overline{r}, k_1(\wr \overline{u} \wr \ \wr \ ), \mathbf{y} \wr \mathbf{z} \wr \mathbf{t})$ and $\mathbf{f}(\mathbf{x} \wr \overline{r}, k_2(\wr \overline{u} \wr \ \wr \ ), \mathbf{y} \wr \mathbf{z} \wr \mathbf{t})$. Hence, it

can be computed by an alternating procedure. The space bound follows from the bound on the size of the strings, provided by the tiering discipline.

▶ **Definition 12.** *NC-tropical functions*
The class of NC-tropical functions is the closure of our basic pointer tropical functions, under tropical composition, tropical recursion on notations and tropical recursion with substitutions on notations.

▶ **Theorem 13.** *The class of NC-tropical functions with binary output is NC.*

Proof. The result follows from $A(logspace, polylogtime) = NC$ [16], and Theorem 7. Substitutions in the tropical recursion scheme on notations amounts to alternation. The proof is similar to that of Theorem 11, where additionally,

- The time bound on the computation of the machine needs only to be polylogarithmic, instead of polynomial. As in Theorem 9, tropical recursion on notations suffices to obtain this bound, and tropical recursion on values is no longer needed.
- For the other direction, any bit of $\mathbf{g}_a(\mathbf{x} \wr \overline{r}, \mathbf{f}(\mathbf{x} \wr \overline{r}, k_1(\wr \overline{u} \wr\ \wr), \mathbf{y} \wr \mathbf{z} \wr \mathbf{t}), \mathbf{f}(\mathbf{x} \wr \overline{r}, k_2(\wr \overline{u} \wr\ \wr), \mathbf{y} \wr \mathbf{z} \wr \mathbf{t}), \mathbf{y} \wr \mathbf{z} \wr \mathbf{t})$ is again a boolean function of the bits of $\mathbf{f}(\mathbf{x} \wr \overline{r}, k_1(\wr \overline{u} \wr\ \wr), \mathbf{y} \wr \mathbf{z} \wr \mathbf{t})$ and $\mathbf{f}(\mathbf{x} \wr \overline{r}, k_2(\wr \overline{u} \wr\ \wr), \mathbf{y} \wr \mathbf{z} \wr \mathbf{t})$. Here, this boolean function can be computed by a boolean circuit of polylogarithmic depth, hence, by an alternating procedure in polylogarihtmic time. The arguments behind this remark are the same as the ones in the proof of $A(logspace, polylogtime) = NC$.

7 Concluding Remarks

Theorems 7, 9, 11, and 13 rely on mutual recursive schemes. As stated above, we use these mutual schemes to express a space computation of size $k \log(n)$ for any constant k, with binary strings of length at most $\log(n) + c$. If we were to use only non-mutual recursion schemes, we would need to have longer binary strings. This can be achieved by taking as input to our functions, not simply the `Offset`, but some larger string $\#^k(\texttt{Offset})$, where $\#^k$ is a function that appends k copies of its argument.

It also remains to be checked wether one can refine Theorem 13 to provide characterizations of the classes NC^i as in [13]. A first step in this direction is to define a recursion rank, accounting for the nesting of recursion schemes: then, check wether NC-tropical functions of rank i are computable in NC^i. Conversely, check also whether the simulation of Theorem 7 induces a fixed overhead, and wether NC^i can be encoded by NC-tropical functions of rank $i + c$ for some constant c small enough.

Finally, note that we characterize logarithmic space functions with logarithmically long output (Theorem 9), and NC functions with one-bit output (Theorem 13). As usual, polynomially long outputs for these classes can be retrieved via a pointer access: it suffices to parameterize these functions with an additional, logarithmically long input, denoting the output bit one wants to compute. In order to retrieve functions with polynomially long output, this approach could also be added to the syntax, with a `Write` construct similar to our `Read` construct, for writing the output.

References

1 Bill Allen. Arithmetizing uniform NC. *Ann. Pure Appl. Logic*, 53(1):1–50, 1991. URL: `https://doi.org/10.1016/0168-0072(91)90057-S`, `doi:10.1016/0168-0072(91)90057-S`.
2 Stephen Bellantoni and Stephen A. Cook. A new recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2:97–110, 1992.

3 Stephen A. Bloch. Function-algebraic characterizations of log and polylog parallel time. *Computational Complexity*, 4:175–205, 1994. URL: https://doi.org/10.1007/BF01202288, doi:10.1007/BF01202288.

4 Guillaume Bonfante, Reinhard Kahle, Jean-Yves Marion, and Isabel Oitavem. Two function algebras defining functions in NC^k boolean circuits. *Inf. Comput.*, 248:82–103, 2016. URL: https://doi.org/10.1016/j.ic.2015.12.009, doi:10.1016/j.ic.2015.12.009.

5 Ashok K. Chandra, Dexter Kozen, and Larry J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, 1981. URL: http://doi.acm.org/10.1145/322234.322243, doi:10.1145/322234.322243.

6 P. Clote. Sequential, machine-independent characterizations of the parallel complexity classes ALOGTIME, AC^k, NC^k and NC. *Feasible Mathematics, Birkhaüser, 49-69*, 1989.

7 A. Cobham. The intrinsic computational difficulty of functions. In Y. Bar-Hillel, editor, *Proceedings of the International Conference on Logic, Methodology, and Philosophy of Science*, pages 24–30. North-Holland, Amsterdam, 1962.

8 Martin Hofmann. Linear types and non-size-increasing polynomial time computation. *Inf. Comput.*, 183(1):57–85, 2003.

9 Martin Hofmann and Ulrich Schöpp. Pure pointer programs with iteration. *ACM Trans. Comput. Log.*, 11(4):26:1–26:23, 2010. URL: http://doi.acm.org/10.1145/1805950.1805956, doi:10.1145/1805950.1805956.

10 Satoru Kuroda. Recursion schemata for slowly growing depth circuit classes. *Computational Complexity*, 13(1-2):69–89, 2004. URL: https://doi.org/10.1007/s00037-004-0184-4, doi:10.1007/s00037-004-0184-4.

11 Daniel Leivant. A foundational delineation of computational feasiblity. In *Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July 15-18, 1991*, pages 2–11. IEEE Computer Society, 1991. URL: https://doi.org/10.1109/LICS.1991.151625, doi:10.1109/LICS.1991.151625.

12 Daniel Leivant and Jean-Yves Marion. Ramified recurrence and computational complexity ii: Substitution and poly-space. In Leszek Pacholski and Jerzy Tiuryn, editors, *CSL*, volume 933 of *Lecture Notes in Computer Science*, pages 486–500. Springer, 1994.

13 Daniel Leivant and Jean-Yves Marion. A characterization of alternating log time by ramified recurrence. *Theor. Comput. Sci.*, 236(1-2):193–208, 2000.

14 Daniel Leivant and Jean-Yves Marion. Ramified Recurrence and Computational Complexity IV : Predicative Functionals and Poly-Space. *Information and Computation*, page 12 p, 2000. to appear. Article dans revue scientifique avec comité de lecture. URL: https://hal.inria.fr/inria-00099077.

15 J. C. Lind. Computing in logarithmic space. Technical report, Massachusetts Institute of Technology, 1974.

16 Walter L. Ruzzo. On uniform circuit complexity. *J. Comput. Syst. Sci.*, 22(3):365–383, 1981. URL: https://doi.org/10.1016/0022-0000(81)90038-6, doi:10.1016/0022-0000(81)90038-6.

Appendix

In this section we provide the definitions of the L-tropical functions used in the proof of Theorem 7, and the subsequent theorems. We also provide more details on the RATM simulation performed in this proof.

Some Assumptions on the RATM being simulated

Let f be a function computable in deterministic space $k\log(n)$, with output of size $k\log(n)$, computed by a RATM M. We assume the following.

- The machine M uses one pointer tape, of size $\lceil \log(n+1) \rceil$, and exactly one computation tape.
- For every input $\overline{x}$ of length n, the machine uses exactly $k.\lceil \log(n+1) \rceil$ cells on the computation tape.
- At the start of the computation, the computation tape is as follows.
  1. The computation tape is on a cell containing the **0** symbol, followed by $k.\lceil \log(n+1) \rceil - 1$ **0** cells on the right.
  2. The cells on the left of the computation head, and the cells on the right of the $k.\lceil \log(n+1) \rceil$ **0** symbols, contain only blank symbols.
- Moreover, during the computation, the following holds.
  1. The computation head never goes on any cell on the left of its initial position.
  2. The machine never writes a blank symbol.
- The same assumptions are made for the pointer tape.

It is easy to check that these assumptions are benign. They enable us to ignore the blank symbol in the simulation, and have a strict correspondence between the binary symbols of the RATM and those of the L-tropical algebra.

Encoding Machine Configurations

We need to encode the four following datas:

1. Machine state. Assume M has S states, with initial state numbered 0 and final state numbered 1. A machine state t is encoded by a binary string of length $\lceil \log(S+1) \rceil$, consisting in the binary encoding of t, padded with **0** symbols if necessary.
2. Computation tape. We only need to encode the $k.\lceil \log(n+1) \rceil$ cells on the right of the initial head position (including itself). These cells never contain a blank symbol during the computation, we can therefore encode them in binary. We encode them in a k-tuple of binary sequences of length $\lceil \log(n+1) \rceil$.
3. Computation head. The position of the computation head is encoded by a binary string of length $k.\lceil \log(n+1) \rceil$, with a **1** symbol on the position of the head, and **0** symbols everywhere else. This binary sequence is given by a k-tuple of binary strings of length $\lceil \log(n+1) \rceil$.
4. Pointer tape. The pointer tape has size $\lceil \log(n+1) \rceil$: it is therefore encoded by a binary string of length $\lceil \log(n+1) \rceil$.
5. Pointer tape head. The position of the pointer tape head is encoded by a binary string of length $\lceil \log(n+1) \rceil$, with a **1** symbol on the position of the head, and **0** symbols everywhere else.

The encoding of the machine configuration is then given by the $(2k+3)$-tuple of the above binary strings.

Notation-wise, in our simulation, we use the variable name $\overline{s}$ for recursion schemes on the size of the input: that is, such recursion schemes are meant to be initially called with an argument $\overline{s} = \texttt{Offset}$ of length $L(n)$. Similarly, we use the variable name $\overline{t}$ for recursion schemes on the computation time.

L-tropical Functions for in-place Read/write Instructions

7.0.0.1 Boolean values and connective

We encode booleans `false` and `true` with $\mathtt{s}_0(\varepsilon \wr \ \wr \ \wr)$ and $\mathtt{s}_1(\varepsilon \wr \ \wr \ \wr)$ respectively. We define the following `match` construct

$$\begin{array}{lll} \mathtt{match} & \overline{x} & \mathtt{with} \\ & | & \mathtt{s}_0(\overline{r} \wr \ \wr \ \wr) -> A \\ & | & \mathtt{s}_1(\overline{r} \wr \ \wr \ \wr) -> B \\ & | & \varepsilon -> C \end{array}$$

as the following degenerate tropical recursion on notations.

$$\begin{array}{rcl} \mathtt{match}(\wr \mathtt{s}_0(\overline{r} \wr \ \wr \ \wr), \overline{a}, \overline{b}, \overline{c} \wr \ \wr) & = & \overline{a} \\ \mathtt{match}(\wr \mathtt{s}_1(\overline{r} \wr \ \wr \ \wr), \overline{a}, \overline{b}, \overline{c} \wr \ \wr) & = & \overline{b} \\ \mathtt{match}(\wr \varepsilon, \overline{a}, \overline{b}, \overline{c} \wr \ \wr) & = & \overline{c} \end{array}$$

Then, `if then esle`, and `AND` and `OR` boolean functions are obtained by trivial applications of the `match` construct above. We also use a function `isempty`, for testing if a string equals ε.

7.0.0.2 1-bit concatenation

Adding one-bit in first position.

$$\begin{array}{lll} \mathtt{1BC}(\overline{y} \wr \overline{x} \wr \ \wr) = \mathtt{match} & \overline{x} & \mathtt{with} \\ & | & \mathtt{s}_0(\overline{t} \wr \ \wr \ \wr) -> \mathtt{s}_0(\overline{y} \wr \ \wr \ \wr) \\ & | & \mathtt{s}_1(\overline{t} \wr \ \wr \ \wr) -> \mathtt{s}_1(\overline{y} \wr \ \wr \ \wr) \\ & | & \varepsilon -> \overline{y} \end{array}$$

For notational purposes we sometimes use $\mathtt{hd}(\wr \ \wr \ \wr \overline{x}).\overline{y}$ instead.

7.0.0.3 Tail extraction

Extracting the tail of a string, for a given prefix length.

$$\begin{array}{rcl} \mathtt{Te}(\wr \mathtt{s}_a(\overline{x} \wr \ \wr \ \wr) \wr \overline{e} \wr) & = & \mathtt{tl}(\wr \ \wr \mathtt{Te}(\wr \overline{x} \wr \overline{e} \wr) \wr) \\ \mathtt{Te}(\wr \varepsilon \wr \overline{e} \wr) & = & \overline{e} \end{array}$$

7.0.0.4 Bit extraction

Extracting one bit of a string, for a given prefix length.

$$\mathtt{Be}(\wr \ \wr \ \wr \overline{x}, \overline{e}) = \mathtt{hd}(\wr \ \wr \ \wr \mathtt{Te}(\wr \overline{x} \wr \overline{e} \wr))$$

7.0.0.5 Head extraction

Extracting the head a string, for a given prefix length.

$$\begin{aligned}\texttt{He}(\wr\, \texttt{s}_a(\wr\ \wr\ \wr\, \overline{x})\wr\ \wr\, \overline{e}) &= \texttt{Be}(\wr\ \wr\ \wr\, \texttt{s}_a(\wr\ \wr\ \wr\, \overline{x}), e).\texttt{He}(\wr\, \overline{x}\wr\ \wr\, \overline{e})\\ \texttt{He}(\wr\, \varepsilon\wr\ \wr\, \overline{e}) &= \varepsilon\end{aligned}$$

7.0.0.6 Prefix length Computation

Extract the initial subsequence of **0** only symbols, followed by the first **1**. This function is used for computing the prefix length corresponding to the position of the head in our encoding of the tapes of the RATM.

$$\begin{aligned}\texttt{Prefix}(\wr\, \varepsilon\wr\ \wr\,) &= \varepsilon\\ \texttt{Prefix}(\wr\, \texttt{s}_0(\wr\ \wr\ \wr\, \overline{x})\wr\ \wr\,) &= \texttt{s}_0(\texttt{Prefix}(\wr\, \overline{x}\wr\ \wr\,)\wr\ \wr\ \wr\,)\\ \texttt{Prefix}(\wr\, \texttt{s}_1(\wr\ \wr\ \wr\, \overline{x})\wr\ \wr\,) &= \texttt{s}_1(\varepsilon\wr\ \wr\ \wr\,)\end{aligned}$$

7.0.0.7 Length Comparison

A predicate for comparing string lengths

$$\texttt{SameLength}(\wr\, \overline{x}, \overline{y}\wr\ \wr\,) =$$

$$\texttt{AND}(\wr\, \texttt{isempty}(\wr\, \texttt{Te}(\wr\, \overline{x}\wr\ \wr\, \overline{y})\wr\ \wr\,), \texttt{isempty}(\wr\, \texttt{Te}(\wr\, \overline{y}\wr\ \wr\, \overline{x})\wr\ \wr\,)\wr\ \wr\,).$$

7.0.0.8 One bit replacement

Replacing exactly one bit in a string $\overline{e}$ by the first bit of $\overline{b}$, for a given prefix length $\overline{x}$.

$$\begin{aligned}\texttt{Cb}(\wr\, \texttt{s}_a(\overline{x}\wr\ \wr\ \wr\,)\wr\ \wr\, \overline{y}, \overline{e}, \overline{b}) &=\\ &\quad \texttt{if SameLength}(\wr\, \texttt{s}_a(\overline{x}\wr\ \wr\ \wr\,), \overline{y}\wr\ \wr\,)\\ &\quad\ \texttt{then hd}(\wr\ \wr\ \wr\, \overline{b}).\texttt{Cb}(\wr\, \overline{x}\wr\ \wr\, \overline{y}, \overline{e}, \overline{b})\\ &\texttt{else Be}(\wr\ \wr\ \wr\, \texttt{Te}(\wr\, \texttt{s}_a(\overline{x}\wr\ \wr\ \wr\,)\wr\, \overline{e}\wr\,), \overline{e}).\texttt{Cb}(\wr\, \overline{x}\wr\ \wr\, \overline{y}, \overline{e}, \overline{b})\\ \texttt{Cb}(\wr\, \varepsilon\wr\ \wr\, \overline{y}, \overline{e}, \overline{b}) &= \varepsilon\end{aligned}$$

and

$$\texttt{ChBit}(\wr\, \overline{s}\wr\ \wr\, \overline{x}, \overline{e}, \overline{b}) = \texttt{Cb}(\wr\, \overline{s}\wr\ \wr\, \texttt{Te}(\wr\, \overline{x}\wr\ \wr\, \overline{e}), \overline{e}, \overline{b})$$

for any $\overline{s}$ with $|\overline{s}| = |\overline{e}|$.

Reading and Updating a Configuration

The `Prefix` function above computes the prefix corresponding the position of the head of the pointer and of the computation heads in our encoding. Used in conjunction with the boolean constructs on the k strings encoding the computation tape, and in conjunction with the bit extraction function `Be` above, it allows to read the current symbol on the computation

tape, and on the pointer tape, of the encoding of the RATM. Updating these two symbols is performed with the `ChBit` in-place one-bit replacement function.

Similarly, moving the heads of these two tapes can easily be performed with this `ChBit`, in conjunction with the `tl` and $\mathtt{s}_1$ basic tropical functions.

Let us now describe how we can read and update the machine state: This machine state is encoded in binary by a string of length $\lceil \log(S+1) \rceil$, where S is the number of the states of M. The length of this string is fixed, and does not depend on the input. Therefore, we can safely assume that we have a fixed decision tree of depth $\lceil \log(S+1) \rceil$, for reading each bit of this string. The leaves of this decision tree are in one-to-one correspondence with the states of M. This decision tree can moreover be encoded with basic tropical functions and tropical composition only. Similarly, overwriting the machine state can be done with basic tropical functions and tropical composition only.

Finally, when in an input reading state, the input tape symbol is obtained simply by using the basic tropical function `Read`, with the pointer tape as argument.

The transition map **Next** of the RATM is then obtained by a boolean composition of the above functions. Similarly, computing an encoding of the initial configuration, and reading a final configuration, is simple.