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 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

*Work partially supported by ANR project ELICA - ANR-14-CE25-0005

To appear in EPTCS.

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: binary strings. 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] .

A long version, with details and proofs, can be found at
https://hal.archives-ouvertes.fr/hal-01934791/document

1 Recursion

1.1 Notations, and Recursion on Notations

Data structures considered are the set $\{\mathbf{0},\mathbf{1}\}^*$ of finite words over $\{\mathbf{0},\mathbf{1}\}$. 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 $a.\overline{x}$. 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.

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 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, $\texttt{Read}(\texttt{hd}(\texttt{Offset}))$ 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.

2 Primitive Recursion on Pointers, with Tropical Tiering

We define the set of primitive recursive functions on pointers, equipped with a non-size tiering discipline, called tropical tiering. The tropical tier (tropic) of the i^{th} variable of a function f is denoted $T_i(f)$, and takes values in $\mathbb{Z}\cup\{-\infty\}$. Tropical tiers are meant to denote an upper bound on the output size increase, with respect to the corresponding input size.

Basic pointer functions. Basic pointer functions are the following kind of functions:

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

$$\begin{array}{rclrclrcl} \texttt{hd}(a.\overline{x}) & = & a\ (T_1(\texttt{hd}) = -\infty) & \texttt{tl}(a.\overline{x}) & = & \overline{x}\ (T_1(\texttt{tl}) = -1) & \texttt{s}_0(\overline{x}) & = & 0.\overline{x}\ (T_1(\texttt{s}_0) = 1) \\ \texttt{hd}(\varepsilon) & = & \varepsilon & \texttt{tl}(\varepsilon) & = & \varepsilon & \texttt{s}_1(\overline{x}) & = & 1.\overline{x}\ (T_1(\texttt{s}_1) = 1) \end{array}$$

We also use 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, $\mathtt{s}(\overline{x}) = E(D(\overline{x})+1)$, with $T_1(\mathtt{s}) = 1$.

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

$$\mathtt{Pr}_i^n(\overline{x_1},\cdots,\overline{x_n}) = \overline{x_i}, \text{ with } T_{j\neq i}(\mathtt{Pr}_i^n) = -\infty,\ T_i(\mathtt{Pr}_i^n) = 0$$

3. and, finally, the `Offset` and `Read` constructs, as defined above, with tropic $-\infty$.

Composition. $f(\mathbf{x}) = g(h_1(\mathbf{x}),\cdots,h_n(\mathbf{x}))$, with $T_t(f) = \max_i\{T_i(g)+T_t(h_i)\}$, and

(Mutual) Primitive Recursion on Notations			**(Mutual) Primitive Recursion on Values**		
$\mathbf{f}(\varepsilon,\mathbf{y})$	=	$\mathbf{h}(\mathbf{y})$	$\mathbf{f}(\varepsilon,\mathbf{y})$	=	$\mathbf{h}(\mathbf{y})$
$\mathbf{f}(\mathtt{s}_a(\overline{x}),\mathbf{y})$	=	$\mathbf{g}_a(\overline{x},\mathbf{f}(\overline{x},\mathbf{y}),\mathbf{y})$	$\mathbf{f}(\mathtt{s}(\overline{x}),\mathbf{y})$	=	$\mathbf{g}(\overline{x},\mathbf{f}(\overline{x},\mathbf{y}),\mathbf{y})$.

The tropical tiering for primitive recursion follows two cases:

- For Primitive Recursion on Notations (respectively on Values)
  $T_2(\mathbf{g}_0) \leq 0$ and $T_2(\mathbf{g}_1) \leq 0$. (Respectively $T_2(\mathbf{g}) \leq 0$) In that case, we set
  1. $T_1(\mathbf{f}) = \max\{T_1(\mathbf{g}_0),T_1(\mathbf{g}_1),T_2(\mathbf{g}_0),T_2(\mathbf{g}_1)\}$, (resp. $T_1(\mathbf{f}) = \max\{T_1(\mathbf{g}),T_2(\mathbf{g})\}$) and,
  2. for all $t \geq 1$, $T_t(\mathbf{f}) = \max\{T_{t+1}(\mathbf{g}_0),T_{t+1}(\mathbf{g}_1),T_{t-1}(\mathbf{h}),T_2(\mathbf{g}_0),T_2(\mathbf{g}_1)\}$.
     (resp. $T_t(\mathbf{f}) = \max\{T_{t+1}(\mathbf{g}),T_{t-1}(\mathbf{h}),T_2(\mathbf{g})\}$.)
- For Primitive Recursion on Notations only
  the previous case above does not hold, $T_2(\mathbf{g}_0) \leq 1$, and $T_2(\mathbf{g}_1) \leq 1$. In that case, we also require that $T_1(\mathbf{g}_0) \leq 0$, $T_1(\mathbf{g}_1) \leq 0$, and, for all $t \geq 2$, $T_t(\mathbf{g}_0) = T_t(\mathbf{g}_1) = T_{t-2}(\mathbf{h}) = -\infty$. Then, we set $T_1(\mathbf{f}) = \max\{T_1(\mathbf{g}_0),T_1(\mathbf{g}_1),T_2(\mathbf{g}_0)-1,T_2(\mathbf{g}_1)-1,c_{\mathbf{h}}\}$, where $c_{\mathbf{h}}$ is a constant for $\mathbf{h}$ given in Proposition 2.1 below, and, for $t \geq 1$, $T_t(\mathbf{f}) = -\infty$.

Other cases than the two above do not enjoy tropical tiering.
In the absence of tropical tiering, the class of primitive recursive function on pointers, defined as the closure of the basic functions under composition and the two versions of primitive recursion above, coincide with the classical primitive recursive functions. Tropical tiering ensure that these functions, called L-primitve recursive functions on pointers, admit tropical interpretations, as per Proposition2.1, and therefore logarithmic space bounds (in the size of the computation input). They also capture the whole class of logarithmic space functions (with output of logarithmic length).

Proposition 2.1 *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

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

3 Embedding Tiering into the Syntax: Tropical Recursion

We are able to restrict the tier values to only four distinct values: $1,0,-1$ and $-\infty$, while retaining the same expressiveness. This allows us to refine the syntax and embed the tiering discipline into the syntax. Let us the $\wr$ separation 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, $f(\overline{x} \wr \overline{y} \wr \overline{z} \wr \overline{t}) = g(\wr \overline{y} \wr \overline{x}, \overline{z} \wr \overline{t})$. Tropical Recursion with substitution schemes defined below originate from [14, 13]. Our schemes follow the tropical tiering of the previous section. We define:

Basic tropical pointer functions (BF):

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

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

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

Tropical Recursion on Notations (TRN)- case 1 (TRN)

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

Tropical Recursion on Notations (TRN)- case 2

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

Tropical Recursion on Values (TRV)

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

Tropical Recursion with substitutions on Notations (TRSN)

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

Tropical Recursion with substitutions on Values (TRSV)

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

3.1 Results

The logarithmic space bound given by Proposition 2.1 allow us to use these four recursion schemes to give in a unified framework characterizations of the most relevant sub-polynomial complexity classes:

Theorem 3.1 *1. The closure of BF under TC, TRN and TRV is the class of logspace functions (FL)*

2. *The closure of BF under TC, TRN is the class of logspace/polylogtime functions*
3. *The closure of BF under TC, TRN, TRSN and TRSV is the class of polynomial time functions (FP)*
4. *The closure of BF under TC, TRN, and TRSN is the class of parallel polylogtime functions (FNC)*

Note however that in these characterizations, we require the output length of the functions to be logarithmically bounded. This constraint can be lifted by using a `Write` construct, symmetric to the `Read` construct, for writing the output bit by bit.

References

[1] Bill Allen (1991): *Arithmetizing Uniform NC*. *Ann. Pure Appl. Logic* 53(1), pp. 1–50, doi:10.1016/0168-0072(91)90057-S. Available at https://doi.org/10.1016/0168-0072(91)90057-S.

[2] Stephen Bellantoni & Stephen A. Cook (1992): *A New Recursion-Theoretic Characterization of the Polytime Functions*. *Computational Complexity* 2, pp. 97–110.

[3] Stephen A. Bloch (1994): *Function-Algebraic Characterizations of Log and Polylog Parallel Time*. *Computational Complexity* 4, pp. 175–205, doi:10.1007/BF01202288. Available at https://doi.org/10.1007/BF01202288.

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

[5] Ashok K. Chandra, Dexter Kozen & Larry J. Stockmeyer (1981): *Alternation*. *J. ACM* 28(1), pp. 114–133, doi:10.1145/322234.322243. Available at http://doi.acm.org/10.1145/322234.322243.

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

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

[8] Martin Hofmann (2003): *Linear types and non-size-increasing polynomial time computation*. *Inf. Comput.* 183(1), pp. 57–85. Available at http://dx.doi.org/10.1016/S0890-5401(03)00009-9.

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

[10] Satoru Kuroda (2004): *Recursion Schemata for Slowly Growing Depth Circuit Classes*. *Computational Complexity* 13(1-2), pp. 69–89, doi:10.1007/s00037-004-0184-4. Available at https://doi.org/10.1007/s00037-004-0184-4.

[11] Daniel Leivant (1991): *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*, IEEE Computer Society, pp. 2–11, doi:10.1109/LICS.1991.151625. Available at https://doi.org/10.1109/LICS.1991.151625.

[12] Daniel Leivant & Jean-Yves Marion (1994): *Ramified Recurrence and Computational Complexity II: Substitution and Poly-Space*. In Leszek Pacholski & Jerzy Tiuryn, editors: *CSL, Lecture Notes in Computer Science* 933, Springer, pp. 486–500.

[13] Daniel Leivant & Jean-Yves Marion (2000): *A characterization of alternating log time by ramified recurrence*. *Theor. Comput. Sci.* 236(1-2), pp. 193–208. Available at http://dx.doi.org/10.1016/S0304-3975(99)00209-1.

[14] Daniel Leivant & Jean-Yves Marion (2000): *Ramified Recurrence and Computational Complexity IV : Predicative Functionals and Poly-Space*. *Information and Computation*, p. 12 p. Available at https://hal.inria.fr/inria-00099077. To appear. Article dans revue scientifique avec comité de lecture.

[15] J. C. Lind (1974): *Computing in logarithmic space*. Technical Report, Massachusetts Institute of Technology.

[16] Walter L. Ruzzo (1981): *On Uniform Circuit Complexity*. *J. Comput. Syst. Sci.* 22(3), pp. 365–383, doi:10.1016/0022-0000(81)90038-6. Available at https://doi.org/10.1016/0022-0000(81)90038-6.