Recursion schemata for NC^k

Guillaume Bonfante, Reinhard Kahle, Jean-Yves Marion, and Isabel Oitavem

Loria - INPL, 615, rue du Jardin Botanique, BP-101, 54602 Villers-lès-Nancy, France
CENTRIA, UNL and DM, Universidade de Coimbra, Apartado 3008, 3001-454 Coimbra, Portugal
UNL and CMAF, Universidade de Lisboa, Av. Prof. Gama Pinto, 2, 1649-003 Lisboa, Portugal
E-mail: {Jean-Yves.Marion|Guillaume.Bonfante}@loria.fr; kahle@mat.uc.pt; isarocha@ptmat.fc.ul.pt

Abstract

We give a recursion-theoretic characterization of the complexity classes NC^k *for* $k \geq 1$*. In the spirit of implicit computational complexity, it uses no explicit bounds in the recursion and also no separation of variables is needed. It is based on three recursion schemes, one corresponds to time (time iteration), one to space allocation (explicit structural recursion) and one to internal computations (mutual in place recursion).*

1 Introduction

Since the seminal works of Simmons [22], of Leivant [14, 15], of Bellantoni and Cook [3], and of Girard [11], *implicit computational complexity* has provided models over infinite domains of major complexity classes which are independent from the notion of time or of space related to machines. These studies have nowadays at least two twin directions. The first direction concerns the characterization of complexity classes by mean of logics or of recursion schemas. A motivation is to have a mathematical logic model of bounded resource computations. The second direction is more practical and aims to analyze and certify resources, which are necessary for a program execution. One of the major challenges here is to capture a broad class of useful programs whose complexity is bounded. There are several approaches [1, 13, 19, 7]. In fact, the first direction can be seen as a guideline of the second approach.

This paper falls in the first direction. We give a recursion-theoretic characterization of each class NC^k by mean of a function algebra INC^k based on tree recursion. We demonstrate that $\mathsf{INC}^k = \mathrm{NC}^k$ for $k \geq 1$.

Research supported by the project *Teorias e linguagens de programação para computações com recursos limitados* within the *Programa PESSOA 2005–2006* of *GRICES - EGIDE* and partly by CMAF via the FCT project POCI/MAT/61720/2004 and by DM, FCT-UNL.

The classes NC^k were firstly described based on circuits. NC^k is the class of functions accepted by uniform boolean circuit families of depth $O(\log^k n)$ and polynomial size with bounded fan-in gates, where n is the length of the input—see, for instance, [2] or [12].

Note that to simulate functions which are in NC^k, one may consider the equivalent definition due to Ruzzo. In [21], Ruzzo identifies NC^k with the classes of languages recognized by alternating Turing machines (in short ATMs) in time $O(\log^k n)$ and space $O(\log n)$. We write $\mathrm{NC}^k = \mathrm{ATM}(O(\log^k n), O(\log n))$, for $k \geq 1$.

Characterizations of the classes NC^k in terms of function algebra have been already proposed. Let us mention here the work of Clote [9] based on two recurrence schemas. However, one of them, weak bounded recursion on notation, asks for an a priori bound à la Cobham [10]. Based on the idea of ramification, Bloch gives a characterization of NC using a divide and conquer schema [5]. Our schema have the divide and conquer shape, but they do not involve a tiering discipline.

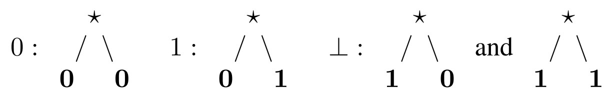
In fact, the main difficulty in this characterization of NC^k relies on the double constraint about time and space. Other previous characterizations based on tree recursion fail to exactly capture for this reason. In 1998, Leivant [16] characterized NC using a hierarchy of classes RSR, such that $\mathsf{RSR}_k \subseteq \mathrm{NC}^k \subseteq \mathsf{RSR}_{k+2}$ for $k \geq 2$. In the sequence of [4] and [20], this result was refined in [6] by defining term systems T^k such that $T^k \subseteq \mathrm{NC}^k \subseteq T^{k+1}$ for $k \geq 2$. Both approaches are defined in a sorted context, either with safe/normal arguments or with tiered recursion.

We define INC^k as classes of functions, over the tree algebra $\mathbb{T}$, closed under composition and three schemes over $\mathbb{T}$: time iteration, explicit structural recursion and mutual in place recursion. No explicit bounds are used in the schemes and also no separation of variables is needed. The mutual in place recursion scheme, one main point of our contribution, is related to previous work of Leivant and Marion, see [17]. The absence of tiering mechanism is related to [18].

2. Preliminaries

Let $\mathbb{W}$ be the set of words over $\{0,1\}$. We denote by ϵ the empty word and by $\mathbb{W}_i$ the subset of $\mathbb{W}$ of words of length exactly i. We consider the tree algebra $\mathbb{T}$, generated by two 0-ary constructors $\mathbf{0}, \mathbf{1}$ and a binary constructor $\star$. $\mathbb{T}$ can be seen as the set of binary trees where leaves are labeled by $\{\mathbf{0}, \mathbf{1}\}$. $\mathsf{S}(t)$ denotes the size of a tree, $\mathsf{H}(t)$ corresponds to the usual notion of height. We say that a tree t is *perfect balanced* (or shorter balanced) if it has $2^{\mathsf{H}(t)}$ leaves each labeled zero or one.

To represent binary trees labeled by the constants $\{0, 1, \perp\}$, we use the following encoding, where $\perp$ is, in fact, encoded by two different trees. Here, 0 serves as false, 1 as true and $\perp$ as the undefined.



For convenience, in the following, we call these trees *value trees* or, in short, *values*.

We use the $\overline{(\,\cdot\,)}$ notation as a shorthand for finite sequences. These can be nested such as in $\bar{\sigma}(\bar{u})$ which denotes a sequence $\sigma_1(u_1, \ldots, u_{k_1}), \ldots, \sigma_n(u_1, \ldots, u_{k_n})$.

Definition 1. *Let us consider the set of functions, called the* basic functions, $\mathcal{B} = \{\mathbf{0}, \mathbf{1}, \star, (\pi^j_i)_{i \le j}, \mathsf{cond}, \mathsf{d}_0, \mathsf{d}_1\}$ *where* $\mathbf{0}, \mathbf{1}$ *and* $\star$ *are the constructors of the algebra* $\mathbb{T}$, d_0 *and* d_1 *are the destructors of* $\mathbb{T}$, cond *is a conditional and* π^j_i *are the usual projections. Destructors and conditional are defined as follows:*

$$\begin{aligned}
\mathsf{d}_0(\mathbf{0}) = \mathsf{d}_1(\mathbf{0}) &= \mathbf{0} \\
\mathsf{d}_0(\mathbf{1}) = \mathsf{d}_1(\mathbf{1}) &= \mathbf{1} \\
\mathsf{d}_0(t_0 \star t_1) &= t_0 \\
\mathsf{d}_1(t_0 \star t_1) &= t_1 \\
\mathsf{cond}(\mathbf{0}, x_0, x_1, x_\star) &= x_0 \\
\mathsf{cond}(\mathbf{1}, x_0, x_1, x_\star) &= x_1 \\
\mathsf{cond}(t_0 \star t_1, x_0, x_1, x_\star) &= x_\star
\end{aligned}$$

The set of basic functions closed by composition is called the set of explicitly defined functions. *If the output of a function is* $\mathbf{0}$ *or* $\mathbf{1}$, *then we say that the function is* boolean. *If the definition of a function does not use* $\star$, *the function is said to be* $\star$*-free.*

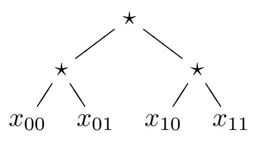
As a shorthand notation, we use $\mathsf{d}_{b_1 \cdots b_k}$ for the function $\mathsf{d}_{b_k} \circ \cdots \circ \mathsf{d}_{b_1}$.

Given a non-empty (enumerable) set of variables $\mathcal{X}$, we denote by $\mathcal{T}(\star, \mathcal{X})$ the term-algebra of binary trees whose leaves are labeled by variables from $\mathcal{X}$. If t, u denote some terms and x is a variable, the term $t[x \leftarrow u]$ denotes the substitution of x by u in t. Then, $t[x \leftarrow u, y \leftarrow v] = t[x \leftarrow u][y \leftarrow v]$. All along, we take care to avoid clashes of variables. When we have a collection I of variable substitutions, we use the notation $t[(x_w \leftarrow u_w)_{w \in I}]$. Again, we will avoid conflicts of variables.

We now define some convenient notations, used extensively all along the paper. Given a set of variables $\mathcal{X} = (x_w)_{w \in \mathbb{W}}$, we define a family of balanced trees $(\mathsf{t}_i)_{i \in \mathbb{N}}$ in $\mathcal{T}(\star, \mathcal{X})$ where each leaf is labeled by a distinct variable:

$$\begin{aligned}
\mathsf{t}_0 &= x_\epsilon \\
\mathsf{t}_{i+1} &= \mathsf{t}_i[(x_w \leftarrow x_{0w})_{w \in \mathbb{W}_i}] \star \mathsf{t}_i[(x_w \leftarrow x_{1w})_{w \in \mathbb{W}_i}]
\end{aligned}$$

As we see, the index of the variables indicate the path from the root to it. For example, t_2 can be represented as follows:



Then, given a family of functions $(f_w)_{w \in \mathbb{W}_2}$, the term $\mathsf{t}_2[(x_w \leftarrow f_w(x_w))_{w \in \mathbb{W}_2}]$ may be used as an abbreviation for $(f_{00}(x_{00}) \star f_{01}(x_{01})) \star (f_{10}(x_{10}) \star f_{11}(x_{11}))$.

This notation is particularly useful if, for example, one wants to describe schemes such as:

$$\begin{aligned}
f((x_{00} \star x_{01}) \star (x_{10} \star x_{11})) = \\
(f(x_{00}) \star f(x_{01})) \star (f(x_{10}) \star f(x_{11}))
\end{aligned}$$

which is rewritten to:

$$f(\mathsf{t}_2) = \mathsf{t}_2[(x_w \leftarrow f(x_w))_{w \in \mathbb{W}_2}]$$

3. The classes INC^k

In this section, for each $k \ge 1$, we describe a class of function definitions, INC^k, closed under composition and three recursion schemes.

Definition 2. INC^k *is the closure of* $\mathcal{B} = \{\mathbf{0}, \mathbf{1}, \star, (\pi^j_i)_{i \le j}, \mathsf{cond}, \mathsf{d}_0, \mathsf{d}_1\}$ *under composition, mutual in place recursion (*MIP*), explicit structural recursion, and time iteration (*TI*) for k.*

The mentioned schemes are described below.

3.1. Mutual in place recursion

The first recursion scheme, *mutual in place recursion*, is the key element of our characterization. It will be used later for the simulation of alternating Turing machines where *valued configuration trees* are updated according to the computation steps performed by an ATM. Mutual in place recursion allows us to perform such updates at relatively low computational cost (cf. Lemma 17).

Definition 3. *The functions $(f_i)_{i\in I}$ (with the set I finite) are defined by* mutual in place recursion *(*MIP*) if they are defined by a set of equations, with $i,j,l \in I$ and $c \in \{\mathbf{0},\mathbf{1}\}$, of the form*

$$f_i(t_0 \star t_1, \bar{u}) = f_j(t_0, \bar{\sigma}_{i,0}(t_0 \star t_1, \bar{u})) \star f_l(t_1, \bar{\sigma}_{i,1}(t_0 \star t_1, \bar{u})) \quad (1)$$

$$f_i(c, \bar{u}) = g_i(c, \bar{u}) \quad (2)$$

where $\bar{\sigma}_{i,0}$ and $\bar{\sigma}_{i,1}$ are sequences of $\star$-free explicitely defined functions and the functions g_i are explicitely defined boolean functions.

(1) *is called the i-*recursion-equation*, (2) the i-*base-equation*. To avoid some non relevant semantics issues, we suppose that for a given i, there is exactly one i-recursion-equation and one i-base-equation.*

Notice that the first argument is shared by the entire set of mutually defined functions as recursion argument. While for the others, copies, switch and visit can be performed freely.

As a consequence, for any such function f, one may observe that $f(t,\bar{x})$ is a tree with the exact shape of t but, possibly, with different leaves.

Example 1. *Given two perfectly balanced trees of common height, using $\mathbf{0}$ as false and $\mathbf{1}$ as true, we can compute the bit-or of their labels using* MIP*-recursion, as follows:*

$$\mathsf{or}(t_0 \star t_1, u) = \mathsf{or}(t_0, \mathsf{d}_0(u)) \star \mathsf{or}(t_1, \mathsf{d}_1(u))$$
$$\mathsf{or}(c, u) = \mathsf{cond}(u, c, \mathbf{1}, \mathbf{1}), \qquad c \in \{\mathbf{0},\mathbf{1}\}$$

Taking the convention that $b \vee \bot = \bot \vee b = b \wedge \bot = \bot \wedge b = \bot$, one may observe that the or function defined above copes with the encoding of $\{0,1,\bot\}$-trees. For the conjunction, we also use MIP-recursion:

$$\begin{aligned} \mathsf{and}(t_0 \star t_1, u) &= \\ \mathsf{and}_0(t_0 \star t_1, u) &= \\ \mathsf{and}_1(t_0 \star t_1, u) &= \mathsf{and}_0(t_0, \mathsf{d}_0(u)) \star \mathsf{and}_1(t_1, \mathsf{d}_1(u)) \\ \mathsf{and}_0(c, u) &= \mathsf{cond}(u, c, \mathbf{1}, \mathbf{1}), \qquad c \in \{\mathbf{0},\mathbf{1}\} \\ \mathsf{and}_1(c, u) &= \mathsf{cond}(u, \mathbf{0}, c, \mathbf{1}), \qquad c \in \{\mathbf{0},\mathbf{1}\} \end{aligned}$$

We now give a lemma which allows us to define a family of MIP-definable functions in terms of the shorthand notation defined above.

Lemma 4. *We suppose given a (finite) family $(n_i)_{i\in I}$ of integers, and a family $(f_i)_{i\in I}$ of functions satisfying equations of the form:*

$$f_i(\mathsf{t}_{n_i}, \bar{u}) = \mathsf{t}_{n_i}[(x_w \leftarrow f_{\mathbb{m}(i,w)}(x_w, \bar{\sigma}_{i,w}(\mathsf{t}_{n_i}, \bar{u})))_{w\in\mathbb{W}_{n_i}}], \quad (3)$$

$$f_i(\mathsf{t}_m[(x_w \leftarrow c_w)_{w\in\mathbb{W}_m}], \bar{u}) = \mathsf{t}_m[(x_w \leftarrow g_{i,w}(c_w, \bar{u}))_{w\in\mathbb{W}_m}], 0 \le m < n_i, \quad (4)$$

where $\mathbb{m}$ is a finite mapping $I \times \mathbb{W}_n \to I$, $c_w \in \{\mathbf{0},\mathbf{1}\}$, $\bar{\sigma}_{i,w}$ are vectors of $\star$-free explicitly defined functions, and $(g_{i,w})_{i\in I, w\in\mathbb{W}}$ are explicitely defined boolean functions. Then, the functions $(f_i)_{i\in I}$ are MIP*-definable.*

Proof. In an equation such as Equation (3), we call n_i the level of the definition of f_i. The proof is by induction on the maximal level of the functions $N = \max_{i\in I} n_i$.

If $N = 1$, then the equations correspond to usual MIP-equations.

Suppose now $N > 1$. For all the indices i such that f_i has level N, we replace its definitional equations by:

$$\begin{aligned} f_i(t_0 \star t_1, \bar{u}) &= f_{i\bullet 0}(t_0, t_0 \star t_1, \bar{u}) \star f_{i\bullet 1}(t_1, t_0 \star t_1, \bar{u}) \\ f_{i\bullet w}(t_0 \star t_1, t_\epsilon, \bar{u}) &= f_{i\bullet w0}(t_0, t_\epsilon, \bar{u}) \star f_{i\bullet w1}(t_1, t_\epsilon, \bar{u}), \\ &\qquad (1 < |w| < N-1) \\ f_{i\bullet w}(t_0 \star t_1, t_\epsilon, \bar{u}) &= \\ &f_{i,w0}(t_0, \bar{\sigma}_{i,w0}(t_\epsilon, \bar{u})) \star f_{i,w1}(t_1, \bar{\sigma}_{i,w1}(t_\epsilon, \bar{u})), \\ &\qquad (|w| = N-1) \\ f_{i\bullet w}(c, t_\epsilon, \bar{u}) &= g_{i,w}(c, \bar{u}), \qquad (1 < |w| < N) \\ f_i(c, \bar{u}) &= g_{i,\epsilon}(c, \bar{u}) \end{aligned}$$

where the indices $i \bullet w$ are fresh. One may observe that the level of each of these functions is 1. So that we transformed a system of equation of level N to a system of equation of level strictly smaller than N. We end by induction. □

The definition of and above can be rewritten—without explicit use of auxiliary functions—according to Lemma 4 as follows:

$$\mathsf{and}(\mathsf{t}_2, u) = \mathsf{t}_2[(x_w \leftarrow \mathsf{and}(x_w, \mathsf{d}_w(u)))_{w\in\mathbb{W}_2}]$$
$$\mathsf{and}(c_0 \star c_1, u) = \mathsf{cond}(\mathsf{d}_0(u), c_0, \mathbf{1}, \mathbf{1}) \star \mathsf{cond}(\mathsf{d}_1(u), \mathbf{0}, c_1, \mathbf{1})$$

We end by a Lemma and a Remark used in the proof of Proposition 12. The Lemma says that MIP-recursion is insensitive to copying, erasing and visit of non recursive arguments.

Lemma 5. *Suppose that $f \in (f_i)_{i\in I}$ is defined by* MIP*-recursion. Then, any function $g(t,\bar{u}) = f(t, \bar{\sigma}(t,\bar{u}))$ where the $\bar{\sigma}$ are $\star$-free explicitly defined functions can be defined by* MIP*-recursion.*

Remark 6. *Suppose that the family $(f_i)_{i\in I}$ is defined by* MIP*-recursion. Suppose that f is defined by the two equations*

$$f(t,\bar{u}) = f_j(t_0,\bar{\sigma_0}(t,\bar{u})) \star f_k(t_1,\bar{\sigma_1}(t,\bar{u}))$$
$$f(c,\bar{u}) = g(c,\bar{u})$$

where $j,k \in I$, the sequences σ_0,σ_1 are $\star$-free explicitly defined functions and g is a boolean explicitly defined functions. Then, f is defined by MIP*-recursion.*

The same conclusion applies with equations of the form:

$$f(\mathsf{t}_n,\bar{u}) = \mathsf{t}_n[(x_w \leftarrow f_{\mathtt{m}(w)}(x_w,\bar{\sigma}_w(\mathsf{t}_n,\bar{u})))_{w\in\mathbb{W}_n}],$$
$$f(\mathsf{t}_m[(x_w \leftarrow c_w)_{w\in\mathbb{W}_m}],\bar{u}) = \mathsf{t}_m[(x_w \leftarrow g_{i,w}(c_w,\bar{u}))_{w\in\mathbb{W}_m}], 0 \le m < n,$$

where $\mathtt{m}$ is a finite mapping $\mathbb{W}_n \to I$, $c_w \in \{\mathbf{0},\mathbf{1}\}$, $\bar{\sigma}_w$ are vectors of $\star$-free explicitly defined functions, and $(g_w)_{i\in I,w\in\mathbb{W}}$ are explicitely defined boolean functions.

3.2. Explicit structural recursion

The recursion scheme defined here corresponds to the space aspect of functions definable in NC^k. It will be used to construct trees of height $O(\log n)$, cf. the following lemma.

Definition 7. Explicit structural recursion *is the following schema, with $c \in \{\mathbf{0},\mathbf{1}\}$:*

$$f(t_0 \star t_1,\bar{u}) = h(f(t_0,\bar{u}), f(t_1,\bar{u}))$$
$$f(c,\bar{u}) = g(c,\bar{u})$$

where h and g are explicitely defined.

Lemma 8. *Given constants α_1 and α_0, one may define in INC^k a function which maps any perfect balanced t tree to a perfect balanced tree of height $\mathsf{H}(f(t)) = \alpha_1\mathsf{H}(t)+\alpha_0$.*

Proof. The proof is immediate taking f defined by explicit structural recursion with $h = h_{\alpha_1}$ and $g = h_{\alpha_0}(\mathbf{1},\mathbf{1})$ where $h_1(w_0,w_1) = w_0 \star w_1$ and $h_i(w_0,w_1) = h_{i-1}(w_0,w_1) \star h_{i-1}(w_0,w_1)$ for $i > 1$. □

3.3. Time iteration

The following scheme allows us to iterate MIP-definable functions. It serves to capture the time aspect of functions definable in NC^k and depends, obviously, on the parameter k.

Definition 9. *Given $k \ge 1$, we define* time iteration *(*TI*), for k, as follows, with $c_1,\dots,c_k \in \{\mathbf{0},\mathbf{1}\}$:*

$$f(t'_1 \star t''_1, t_2,\dots,t_k,s,\bar{u}) = h(f(t'_1,t_2,\dots,t_k,s,\bar{u}),\bar{u})$$
$$f(c_1, t'_2 \star t''_2, t_3,\dots,t_k,s,\bar{u}) = f(s,t'_2,t_3\dots,t_k,s,\bar{u})$$
$$\vdots$$
$$f(c_1,\dots,c_{i-1},t'_i \star t''_i,t_{i+1},\dots,t_k,s,\bar{u}) = f(c_1,\dots,c_{i-2},s,t'_i,t_{i+1},\dots,t_k,s,\bar{u})$$
$$\vdots$$
$$f(c_1,\dots,c_k,s,\bar{u}) = g(s,\bar{u})$$

where the function h is MIP*-definable (but no condition is imposed on g).*

Lemma 10. *Given a* MIP*-definable function h, a function g in INC^k and constants β_1 and β_0, one may define in INC^k, by time iteration, a function f such that for all perfect balanced tree t*

$$f(t,\bar{u}) = \underbrace{h(\dots h}_{\beta_1(\mathsf{H}(t))^k+\beta_0 \text{ times}}(g(t,\bar{u}),\bar{u})\dots).$$

Proof. Let us assume $k=2$ (the other cases are analogous). In this case, the time iteration scheme has the shape

$$f(t'_1 \star t''_1,t_2,s,\bar{u}) = h(f(t'_1,t_2,s,\bar{u}),\bar{u})$$
$$f(c_1,t'_2 \star t''_2,s,\bar{u}) = f(s,t'_2,s,\bar{u})$$
$$f(c_1,c_2,s,\bar{u}) = g(s,\bar{u})$$

It is trivial to prove by induction on $\mathsf{H}(t)$ that

$$f(t,t,t,\bar{u}) = \underbrace{h(\dots h}_{(\mathsf{H}(t))^2 \text{ times}}(g(t,\bar{u}),\bar{u})\dots),$$

where g can be any function previously defined in INC^k.

By composition, $f_0(t,\bar{u}) = \underbrace{h(\dots h}_{\beta_0 \text{ times}}(g(t,\bar{u}),\bar{u})\dots)$ is in INC^k. Then, considering $f_i(t,t,t,\bar{u})$ by (TI) based on h and $g = f_{i-1}$, it is clear that f_{β_1} is the desired function. □

We state now our main theorem:

Theorem 11. *For $k \ge 1$, functions in INC^k are exactly functions computed by circuits in* NC^k.

The proof of the theorem is a direct consequence of Proposition 12 and Proposition 13 coming in the two next Sections.

4. Simulation of alternating Turing machines

The concept of an alternating Turing Machine was introduced by Chandra, Kozen and Stockmeyer as a generalization of the non-deterministic Turing machine concept, see [8].

Here we consider alternating random access Turing machines (ARMs) as described in [16], by Leivant. This means that, here, the operational semantics of a ARM, M, is described as a two stage process: firstly, generating an input-independent computation tree; secondly, evaluating that computation tree for a given input.

A binary tree T of configurations is a *computation tree (of M)* if each non-leaf of T spawns its children configurations. A computation tree of M is generated as follows: when in a configuration with an action state, depending on the state and on the bits at the top of the stacks, it spawns a pair of successor configurations. We assume that the machine works with two stacks. Moreover, we assume that one of the successor configurations, let us say the first one, lets the stacks unchanged, and that the stacks of the second configuration can only be changed by pushing or poping one letter at the top of one stack, or/and changing the internal state of the machine. We will be interested in computation trees which have the initial configuration of M as a root. Each computation tree, T, maps binary representation of integers (inputs) to a value in $\{0, 1, \perp\}$, where bottom denotes "undefined". This map is defined according to points 1 and 2, below.

1. If T is a single configuration with state q then:
   (a) if q is an accepting [rejecting] state, the returned value is 1 [respectively, 0];
   (b) if q is an action state, the returned value is $\perp$;
   (c) if q is an reading state then the returned value is 1 or 0 depending on whether the information which results from concatenating the stacks points (notion to be made explicit later on), in the input, to 1 or not.
2. If T is not a single configuration, then the root configuration has a conjunctive or a disjunctive state. We define the value returned by T to be the conjunction, respectively the disjunction, of the values returned by the immediate subtrees.

Conjunctive and disjunctive states may diverge, indicated by the "undetermined value", $\perp$.

The inputs of the machines are binary numbers, which are inputed to terms as minimal-size perfectly balanced trees, with leaves representing the binary number (with high-order zeros used to pad the leftmost branches of the tree). We say that a binary string *points to* the bit "i", of the input, if it determines a path, in the tree representing the input, which leads to the bit "i". An output consisting of just a leaf 1 is interpreted as "true" or "accept"; any other output tree is interpreted as "false" or "reject". In this way, each term recognizes/accepts a language.

Proposition 12. *Given $k \geq 1$ and constants $\alpha_1, \alpha_0, \beta_1, \beta_0$, any* ARM *working in space $\alpha_1 \log(n) + \alpha_0$ and time $\beta_1 \log^k(n) + \beta_0$, where n is the length of the input, can be simulated in* INC^k.

Proof (sketch). Let us consider such a machine. We can describe the transition function δ for action states as follows. $\delta(q, a, b) = (q', q'', \mathsf{pop}_i)$ with $i \in \{1, 2\}$ means that being in state q with top bits being a and b, the first successor configuration lets the stacks unchanged and has state q', and the second successor has state q'' and pops the stack i. When we write $\delta(q, a, b) = (q', q'', \mathsf{push}_i(c))$, with $i \in \{1, 2\}$ and $c \in \mathbb{W}_1$, it is like above but we push the letter c on the top of the stack i.

Take $d = \lceil \log(|Q|) \rceil$, where Q is the finite set of states, then one may attribute to each state in Q a word $w \in \mathbb{W}_d$. We take the convention that the initial state q_0 has encoding $\underbrace{0 \cdots 0}_{d \text{ letters}}$. From now on, the distinction between the state and its associated word is omitted.

Suppose we are given two stacks $s_1 = a_1 a_2 \cdots a_i$ and $s_2 = b_1 b_2 \cdots b_j$ over $\{0, 1\}$ of length less or equal than $\alpha_1 \cdot \log(n) + \alpha_0$. These stacks will be now encoded as a word which will later form (a part of) a path in a configuration tree. We use an extra character # to separate the content of the two stacks and to pad all paths to a fixed length. Using the encoding l with $\mathsf{l}(\mathtt{0}) = 10$, $\mathsf{l}(\mathtt{1}) = 11$ and $\mathsf{l}(\#) = 00$, we encode the two stacks as above by a word

$$\mathsf{P}(s_1, s_2) = \\ \mathsf{l}(a_1)\mathsf{l}(b_1)\mathsf{l}(a_2) \cdots \mathsf{l}(a_i)\mathsf{l}(\#)\mathsf{l}(b_2) \cdots \mathsf{l}(b_j)\mathsf{l}(\#)\mathsf{l}(\#) \cdots \mathsf{l}(\#)$$

in such a way that this word has length exactly $2(\alpha_1 \cdot \log(n) + \alpha_0 + 1)$. The "+1" origins from the extra character # which separates the two (tails of the) stacks.

For convenience we use typewriter font for the encoding given by l: taking $a_1, \ldots, b_1, \ldots$ as above, the encoding is then written as:

$$\mathsf{P}(s_1, s_2) = \mathtt{a}_1\, \mathtt{b}_1\, \mathtt{a}_2\, \mathtt{a}_3 \cdots \mathtt{a}_\mathtt{i}\, \#\, \mathtt{b}_2\, \mathtt{b}_3 \cdots \mathtt{b}_\mathtt{j}\, \#\, \# \cdots \#.$$

A *configuration tree* is a perfectly balanced tree of height $d + 2(\alpha_1 \cdot \log(n) + \alpha_0 + 1)$. It is the "container" for all configurations. A full path in a configuration tree can be decomposed as a word qw with $q \in \mathbb{W}_d$ and $w \in \mathbb{W}_{2(\alpha_1 \cdot \log(n) + \alpha_0 + 1)}$. A *configuration path* is one for which q is an encoding of a state and w is an encoding of the stacks. Note that not all paths of a configuration tree are configuration path; but all possible configurations are represented by a configuration path in the configuration tree. A valued configuration tree is a configuration tree in which leaves are values $0, 1, \perp$. With respect to our encoding of values, it is a tree of height $d + 2(\alpha_1 \cdot \log(n) + \alpha_0 + 1) + 1$. Notice that given a valued configuration tree t, the value corresponding to the configuration (q, s_1, s_2) is obtained, in t, by $\mathsf{d}_{q\mathsf{P}(s_1,s_2)}(t)$.

We describe now the process of the computation. The initial valued configuration tree has all leaves labeled by $\perp$, i.e., $\mathbf{1} \star \mathbf{1}$ (this tree can be defined by explicit structural recursion, cf. Lemma 8). The strategy will be to update the leaves of the initial valued configuration tree, according to the procedure described in items 1 and 2 above, as many times as the running time of the machine. For the update one must use a MIP-function which is then, by Lemma 10, iterated using time iteration. After this process, the output can be obtained following the path for the initial configuration $(q_0, \epsilon, \epsilon)$.This finishes the proof. Now, we just have a closer look how the update is implemented by a MIP-function.

The update function next takes as input the currently computed valued configuration tree and the input tree. It returns the next configuration tree. Actually, the function can be considered as a sort of case distinction function just calling auxiliary functions. That is:

$$\begin{aligned}&\mathsf{next}(\mathsf{t}_{d+4}, y) = \\ &\mathsf{t}_{d+4}[(x_{q\mathtt{ab}} \leftarrow \mathsf{next}_{q,a,b}(x_{q\mathtt{ab}}, \mathsf{t}_{d+4}, y))_{q\in\mathbb{W}_d, \mathtt{a}\in\mathbb{W}_2, \mathtt{b}\in\mathbb{W}_2}]\end{aligned}$$

where $\mathsf{next}_{q,a,b}$ are the auxiliary functions. By Remark 6, it is sufficient to prove that these auxiliary function can be defined by MIP-recursion[1]. The role of these functions is to update the part of the configuration tree they correspond to. We have to distinguish the following cases. 1) q is an accepting state; 2) q is a rejecting state; 3) q is a reading state; 4) q is a $\vee$ (disjunctive) state; 5) q is a $\wedge$ (conjunctive) state. Case 4) and 5) both split in four sub-cases, depending on the action corresponding to q.

The update of the part of a valued configuration tree t is comparatively easy for the cases of *accepting* and *rejecting* states. For them we define auxiliary functions accept (resp. reject) taking as input a tree t whose leaves are labeled by values in $\{0, 1, \perp\}$ which returns t with leaves relabeled by 1 (resp. 0).

$$\begin{aligned}\mathsf{accept}(\mathsf{t}_2) &= \mathsf{t}_2[(x_w \leftarrow \mathsf{accept}(x_w))_{w\in\mathbb{W}_2}]\\ \mathsf{accept}(c_0 \star c_1) &= \mathbf{0} \star \mathbf{1}\\ \mathsf{accept}(c) &= c\end{aligned}$$

$$\begin{aligned}\mathsf{reject}(\mathsf{t}_2) &= \mathsf{t}_2[(x_w \leftarrow \mathsf{reject}(x_w))_{w\in\mathbb{W}_2}]\\ \mathsf{reject}(c_0 \star c_1) &= \mathbf{0} \star \mathbf{0}\\ \mathsf{reject}(c) &= c\end{aligned}$$

Formally, we define $\mathsf{next}_{q,a,b}(x, t, y) = \mathsf{accept}(x)$ if q is an accepting state and $\mathsf{next}_{q,a,b}(x, t, y) = \mathsf{reject}(x)$ if q is a rejecting state. We use Lemma 5 to show that $\mathsf{next}_{q,a,b}$ is defined by MIP-recursion.

For the *reading* cases, the situation is a little bit more complicated, since we have to take into account the coding of the first bit b_1 of the second stack between the bits of the first stack. The auxiliary function read uses as input $\mathsf{d}_q(t)$ and y, the minimal size perfectly balanced tree coding the input of the machine (with high-order zeros used to pad the leftmost branches of the tree). It returns $\mathbf{0}\star\mathbf{1}$ (i.e. the valued tree corresponding to 1) if the string $a_1 \cdots a_i b_1 \cdots b_j$ points to the bit 1 in y, and $\mathbf{0} \star \mathbf{0}$ otherwise.

The formal definition of the read function is a technical exercise which is left out here. It uses MIP-recursion in an essential way. This includes the use of auxiliary functions $\bar{\sigma}$ which are $\star$-free explicitely defined.

The hard cases are the disjunctive and conjunctive states. In these cases, to compute the value of a configuration, we need the value of its two successor configurations. The key point is that the transformation of a configuration $(q, a_1 \cdots a_i, b_1 \cdots b_j)$ to its successors is entirely determined by the state q and the two top bits a_1 and b_1 so that $\mathsf{next}_{q,a_1,b_1}$ "knows" exactly which transformation it must implement. Here, our encoding of the stacks come into play.

We have to distinguish the four cases where we push or pop an element on one of the two stacks:

1. $\delta(q, \mathrm{a}_1, \mathrm{b}_1) = (q', q'', \mathsf{push}_1(\mathrm{a}_0))$;
2. $\delta(q, \mathrm{a}_1, \mathrm{b}_1) = (q', q'', \mathsf{pop}_1)$;
3. $\delta(q, \mathrm{a}_1, \mathrm{b}_1) = (q', q'', \mathsf{push}_2(\mathrm{b}_0))$;
4. $\delta(q, \mathrm{a}_1, \mathrm{b}_1) = (q', q'', \mathsf{pop}_2)$.

By assumption, the stacks of q' are the same as for q, so that its encoding is

$$q'\, \mathtt{a}_1\, \mathtt{b}_1\, \mathtt{a}_2\, \mathtt{a}_3 \cdots \mathtt{a}_\mathtt{i}\ \#\, \mathtt{b}_2\, \mathtt{b}_3 \cdots \mathtt{b}_\mathtt{j}\ \#\ \# \cdots \#$$

but for q'' we get the four different configurations:

1. $q''\, \mathtt{a}_0\, \mathtt{b}_1\, \mathtt{a}_1\ \mathtt{a}_2\ \mathtt{a}_3 \cdots \mathtt{a}_\mathtt{i}\ \#\ \mathtt{b}_2\ \mathtt{b}_3 \cdots \mathtt{b}_\mathtt{j}\ \# \cdots \#$
2. $q''\, \mathtt{a}_2\, \mathtt{b}_1\, \mathtt{a}_3 \cdots\ \mathtt{a}_\mathtt{i}\ \ \#\ \mathtt{b}_2\, \mathtt{b}_3 \cdots \mathtt{b}_\mathtt{j}\ \#\ \ \#\ \# \cdots \#$
3. $q''\, \mathtt{a}_1\, \mathtt{b}_0\, \mathtt{a}_2\ \mathtt{a}_3\ \cdots\ \mathtt{a}_\mathtt{i}\ \#\ \mathtt{b}_1\ \mathtt{b}_2\ \mathtt{b}_3 \cdots \mathtt{b}_\mathtt{j}\ \# \cdots \#$
4. $q''\, \mathtt{a}_1\, \mathtt{b}_2\, \mathtt{a}_2\ \mathtt{a}_3\ \cdots\ \mathtt{a}_\mathtt{i}\ \#\ \mathtt{b}_3 \cdots \mathtt{b}_\mathtt{j}\ \#\ \ \#\ \# \cdots \#$

[1]Actually, wrt the simulation, Equations for $m < d + 4$ play no role, so that we do not write them explicitly.

For every $\vee$ and $\wedge$ state q we have to define a function $\mathsf{next}_{q,a_1,b_1}$. Given t the presently computed valued configuration tree, $\mathsf{next}_{q,a_1,b_1}(\mathsf{d}_{q\mathtt{a_1}\mathtt{b_1}}(t), t, y)$ is intended to update the values of the corresponding part of the configuration tree t. As for accepting and rejecting states, we will use auxiliary functions $\mathsf{next}_{\circ,1}$, $\mathsf{next}_{\circ,2,b_1}$, $\mathsf{next}_{\circ,3,b_1}$, and $\mathsf{next}_{\circ,4}$, which correspond to the four cases mentioned above (and where $\circ$ is $\wedge$ or $\vee$ according to the state coded by q). Then we use Lemma 5 to keep the functions $\mathsf{next}_{q,a_1,b_1}$ defined by MIP-recursion.

We come back now to the definition of the four auxiliary functions $\mathsf{next}_{\circ,1}$, $\mathsf{next}_{\circ,2,b_1}$, $\mathsf{next}_{\circ,3,b_1}$, and $\mathsf{next}_{\circ,4}$. In the second and third case, we have to define two variants for the two different possible values of $\mathtt{b}_1$ since the definition of the function depends on it (for $\mathsf{next}_{\circ,3,b_1}$ this should be obvious because we have to "copy" b_1 to the tail of the second stack).

The principle of the definition of the auxiliary functions is to follow in parallel the paths of the two successor configurations. To do that, we essentially use substitution of parameters, in the mutual in place recursion scheme. The recurrence argument is there as a "witness" of the currently computed configuration tree.

For the first case, we define $\mathsf{next}_{q,a_1,b_1}(x,t,y) = \mathsf{next}_{\circ,1}(x, \mathsf{d}_{q'a_1b_1}(t), \mathsf{d}_{q''a_0b_1a_1}(t))$ and use Lemma 5. With respect to our encoding, in our simulation, observe that $\mathsf{next}_{\circ,1}(x,u,v)$ is fed with $\mathsf{d}_{qa_1b_1}(t), \mathsf{d}_{q'a_1b_1}(t), \mathsf{d}_{qa_0b_1a_1}(t)$ where t is the currently computed valued configuration tree. So that the height of the last argument is one less than the others. In this case, we can go in parallel, with the only *previso* that the second stack is one letter shorter. Equations below cope with that technical point — we play with the fact that the height of x is $4(\alpha_1 \cdot \log(n) + \alpha_0 - 1) + 1$. Formally we define $\mathsf{next}_{\circ,1}$ as:

$$\begin{aligned}
&\mathsf{next}_{\circ,1}(\mathsf{t}_2, u, v) = \\
&\quad \mathsf{t}_2[(x_w \leftarrow \mathsf{next}_{\circ,1'}(x_w, \mathsf{d}_w(u), \mathsf{d}_w(v)))_{w\in\mathbb{W}_2}] \\
&\mathsf{next}_{\circ,1'}(\mathsf{t}_4, u, v) = \\
&\quad \mathsf{t}_4[(x_w \leftarrow \mathsf{next}_{\circ,1'}(x_w, \mathsf{d}_w(u), \mathsf{d}_w(v)))_{w\in\mathbb{W}_4}] \\
&\mathsf{next}_{\circ,1'}(\mathsf{t}_3[x_w \leftarrow c_w], u, v) = \\
&\quad \mathsf{t}_3[(x_{w0} \leftarrow \mathsf{cond}(\mathsf{d}_{w0}(u), \mathsf{d}_0(v), \mathbf{0}, \mathbf{0})_{w\in\mathbb{W}_2}] \\
&\qquad [(x_{w1} \leftarrow \mathsf{d}_{w1}(u) \circ \mathsf{d}_1(v))_{w\in\mathbb{W}_2}]
\end{aligned}$$

where the c_w are to be taken in $\{\mathbf{0},\mathbf{1}\}$. We do not mention here the equations for trees smaller than 2 for $\mathsf{next}_{\circ,1}$ and 3 for $\mathsf{next}_{\circ,1'}$ since these are never used in the simulation.

For the other case, for the path corresponding to q' we follow always $\mathtt{a}_1\,\mathtt{b}_1$. For the path corresponding to q'' we have the following cases:

2. do not follow any bit;
3. we already follow $\mathtt{a}_1\,\mathtt{b}_0$;
4. in this case we use four copies of the configuration tree and follow in each of them $\mathtt{a}_1\,0\,0$, $\mathtt{a}_1\,0\,1$, $\mathtt{a}_1\,1\,0$, and $\mathtt{a}_1\,1\,1$, respectively.

   Thus, not knowing at this moment which path we have to follow for $\mathtt{b}_2$, we just include all four possibilities, i.e., $0\,0$, $0\,1$, $1\,0$, and $1\,1$.

As a consequence, we have to follow the following paths for q' and q'' to get the leaves which should be combined according to $\circ$:

2. $\mathtt{a}_2\,\mathtt{a}_3\,\cdots\,\mathtt{a}_\mathtt{i}\,\#\,\mathtt{b}_2\,\mathtt{b}_3\,\cdots\,\mathtt{b}_\mathtt{j}\,\#\,\#\,\cdots\,\#$ and
   $\mathtt{a}_2\,\mathtt{b}_1\,\mathtt{a}_3\,\cdots\,\mathtt{a}_\mathtt{i}\,\#\,\mathtt{b}_2\,\mathtt{b}_3\,\cdots\,\mathtt{b}_\mathtt{j}\,\#\,\#\,\#\,\cdots\,\#$

   In this case, we have to follow $\mathtt{a}_2$ in the q' path and $\mathtt{a}_2\,\mathtt{b}_1$ in the q'' path and the case can be reduced to the first one. But to follow $\mathtt{b}_1$ it is needed to have the function parameterized by this letter.

   Thus, for $\mathsf{next}_{\wedge,2,b_1}$ we have:

   $$\begin{aligned}&\mathsf{next}_{\wedge,2,b_1}(\mathsf{t}_2, u, v) = \\ &\quad \mathsf{t}_2[(x_{a_2} \leftarrow \mathsf{next}_{\wedge,1}(x_{a_2}, \mathsf{d}_{a_2}(u), \mathsf{d}_{a_2b_1}(v)))_{a_2\in\mathbb{W}_2}]\end{aligned}$$

3. $\mathtt{a}_2\,\mathtt{a}_3\,\cdots\,\mathtt{a}_\mathtt{i}\,\#\,\mathtt{b}_2\,\mathtt{b}_3\,\cdots\,\mathtt{b}_\mathtt{j}\,\#\,\#\,\cdots\,\#$ and
   $\mathtt{a}_2\,\mathtt{a}_3\,\cdots\,\mathtt{a}_\mathtt{i}\,\#\,\mathtt{b}_1\,\mathtt{b}_2\,\mathtt{b}_3\,\cdots\,\mathtt{b}_\mathtt{j}\,\#\,\cdots\,\#$

   When we build the tree for this situation, we know that at each level the "left-left branch", i.e., the one which follows $0\,0$, encodes at that level the separation character $\#$. Thus, at this stage we have to "pop" the b_1 from the second stack, and can continue afterwards as in case 1.

   The "left-right branch" is actually not used by our encodings (since there is no character encoded by $0\,1$); thus we can do what we like, and we just follow the treatment of the "left-left branch". For the "right-left" and "right-right" branches, which are encodings of an a_i, we just follow both trees in parallel.

   So, $\mathsf{next}_{\wedge,3,b_1}$ is defined as:

   $$\begin{aligned}&\mathsf{next}_{\wedge,3,b_1}(\mathsf{t}_2, u, v) = \\ &\quad (\mathsf{next}_{\wedge,1}(x_{00}, \mathsf{d}_{00}(u), \mathsf{d}_{00b_1}(v)) \star \\ &\qquad \mathsf{next}_{\wedge,1}(x_{01}, \mathsf{d}_{01}(u), \mathsf{d}_{01b_1}(v))) \star \\ &\quad (\mathsf{next}_{\wedge,3,b_1}(x_{10}, \mathsf{d}_{10}(u), \mathsf{d}_{10}(v)) \star \\ &\qquad \mathsf{next}_{\wedge,3,b_1}(x_{11}, \mathsf{d}_{11}(u), \mathsf{d}_{11}(v)))\end{aligned}$$

4. $\mathtt{a}_2\,\mathtt{a}_3\,\cdots\,\mathtt{a}_\mathtt{i}\,\#\,\mathtt{b}_2\,\mathtt{b}_3\,\cdots\,\mathtt{b}_\mathtt{j}\,\#\,\#\,\cdots\,\#$ and
   —for four copies of—
   $\mathtt{a}_2\,\mathtt{a}_3\,\cdots\,\mathtt{a}_\mathtt{i}\,\#\,\mathtt{b}_3\,\cdots\,\mathtt{b}_\mathtt{j}\,\#\,\#\,\#\,\cdots\,\#$

   In this case, $\mathsf{next}_{\wedge,4}$ needs to use four auxiliary functions, $\mathsf{next}_{\wedge,4,00}$, $\mathsf{next}_{\wedge,4,01}$, $\mathsf{next}_{\wedge,4,10}$, and $\mathsf{next}_{\wedge,4,11}$ which treat the four leaves of a tree t_2 according to the branches.

The definitions are a little bit more involved as in the previous cases, but they still follow the same idea, and therefore, they are omitted here. □

5. Compilation of recursive definitions to circuit

This section is devoted to the proof of the Proposition:

Proposition 13. *For $k \geq 1$, any function in INC^k is computable in* NC^k.

We begin with some observations. All along, n denotes the size of the input. First, to simulate theoretic functions in INC^k, we will forget the tree structure and make the computations on the words made by the leaves. Actually, since the trees are always full balanced binary trees, we could restrict our attention to input of size 2^k for some k.

Second, functions defined by explicit structural recursion can be computed by NC^1 circuits. This is a direct consequence of the fact that explicit structural recursion is a particular case of LRRS-recursion as defined in Leivant and Marion [17]

Third, by induction on the definition of functions, one proves the key Lemma:

Lemma 14. *Given a function $f \in \mathsf{INC}^k$, there are (finitely many)* MIP*-functions $h_1, \dots, h_m$ and polynomials $P_1, \dots, P_m$ of degree smaller than k such that $f(\bar{t}, \bar{u}) = h_1^{P_1(\log(n))}(\cdots h_m^{P_m(\log(n))}(g(\bar{u}))\dots)$ where g is defined by structural recursion.*

Now, the compilation of functions to circuits relies on three main ingredients. First point, we show that each function h_i as above can be computed by a circuit:

1. of fixed height with respect to the input (the height depends only on the definition of the functions),
2. with a linear number of gates with respect to the size of the first input of the circuit (corresponding to the recurrence argument),
3. with the number of output bits equal to the number of input bits of its first argument.

According to 1), we note H the maximal height of the circuits corresponding to the h_i's.

Second point, since there are $\sum_{i=1..m} P_i(\log(n))$ applications of such h_i, we get a circuit of height bounded by $H \times \sum_{i=1..m} P_i(\log(n)) = O(\log^k(n))$. That is a circuit of height compatible with NC^k. Observe that we have to add as a first layer a circuit that computes g. According to our second remark, this circuit has a height bounded by $O(log(n))$, so that the height of the whole circuit is of the order $O(\log^k(n))$.

Third point, the circuits corresponding to g, being in NC^1, have a polynomial number of gates with respect to n and a polynomial number of output bits with respect to n. Observe that the output of g is exactly the recurrence argument of some h_i whose output is itself the first argument of the next h_i, and so on. So that according to item 3) of the first point, the size of the input argument of each of the h_i is exactly the size of the output of g. Consequently, according to item 2) above, the number of circuit gates is polynomial.

Since all constructions are uniform, we get the expected result.

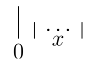
5.1. NC^0 circuits for mutual recursion

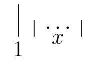
In this section, we prove that functions defined by mutual in place recursion can be computed by NC^0 circuits with a linear number of gates wrt the size of the first argument. Since MIP-functions keep the shape of their first argument, we essentially have to build a circuit for each bit of this argument.

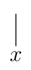
Lemma 15. *Any explicitely defined boolean function can be defined without use of $\star$.*

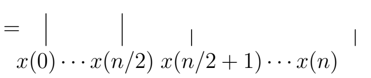
Lemma 16. *Explicitly defined boolean functions are in* NC^0.

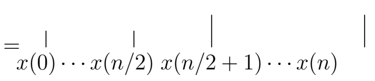
Proof. Consider the following circuits. To stress the fact that circuits are uniform, we put the size of the arguments into the brackets. n correspond to the size of x, n_0 to the size of x_0 and so on. $x(k)$ for $k \in \mathbb{N}$ corresponds to the k-th bit of the input x. The "long" wires correspond to the outputs. Shorter ones are simply forgotten.

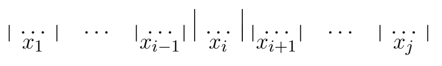
$C_0[n]$:


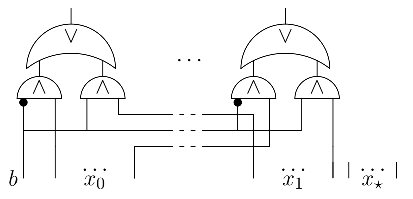
$C_1[n]$:


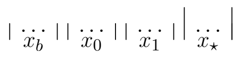
$C_{\mathtt{d}_0}[1] = C_{\mathtt{d}_1}[1] =$


$C_{\mathtt{d}_0}[2+n] =$


$C_{\mathtt{d}_1}[2+n] =$


$C_{\pi_i^j}[n_1, \cdots, n_j]$:


$C_{\texttt{cond}}[1, n_0, n_0, n_\star] =$ 

$C_{\texttt{cond}}[2 + n_b, n_0, n_1, n_\star] =$ 

We see that composing the previous cells, with help of Lemma 15, we can build a circuit of fixed height (wrt to the size of input) for any explicitly defined boolean function. Observe that the constructions are clearly uniform. □

Lemma 17. *Any* MIP*-function can be computed by a circuit of fixed height wrt the size of the input.*

Proof. Let us consider a set $(f_i)_{i\in I}$ of MIP-functions. That is, we have a set of equations of the form:

$$f_i(t_0 \star t_1, \bar{u}) = f_{p(i,0)}(t_0, \bar{\sigma}_{i,0}(\bar{u})) \star f_{p(i,1)}(t_1, \bar{\sigma}_{i,1}(\bar{u}))$$
$$f_i(c, \bar{u}) = g_i(c, \bar{u})$$

where $p(i, b) \in I$ is an explicit (finite) mapping of the indices, $\bar{\sigma}_{i,0}$ and $\bar{\sigma}_{i,1}$ are vectors of $\star$-free explicitely defined functions and the functions $g_{i,c}$ (and consequently the g_i) are explicitly defined boolean functions.

First, observe that any of these explicitly defined functions g_i can be computed by some circuit B_i of fixed height as seen in Lemma 16. Since I is finite, we call M the maximal height of these circuits $(B_i)_{i\in I}$.

Suppose we want to compute $f_i(t, \bar{x})$ for some t and $\bar{x}$ which have both size smaller than n. Remember that the shape of the output is exactly the shape of the recurrence argument t. So, to any k-th bit of the recurrence argument t, we will associate a circuit computing the corresponding output bit, call this circuit C_k. Actually, we will take for each k, $C_k \in \{B_i : i \in I\}$. Putting all the circuits $(C_k)_k$ in parallel, we get a circuit that computes all the output bits of f_i, and moreover, this circuit has a height bounded by M. So, the last point is to show that for each k, we may compute uniformly the index i of the circuit B_i corresponding to C_k and the inputs of the circuit C_k.

To denote the k-th bit of the input, consider its binary encoding where we take the path in the full binary tree t ending at this k-th bit. Call this path w. Notice first that w itself has logarithmic size wrt n, the size of t. Next, observe that any sub-tree of the inputs can be represented in logarithmic size by means of its path. Since all along the computations, the arguments $\bar{u}$ are sub-trees of the input, we can accordingly represent them within the space bound.

To represent the value of a subterm of some input, we use the following data structure. Consider the record type $\texttt{st} = \{\texttt{r}; \texttt{w}; \texttt{h}\}$. The field `r` says to which input the value corresponds to. `r = 0` corresponds to t, `r = 1` correspond to x_1 and so on. `w` gives the path to the value (in that input). For convenience, we keep its height `h`. In summary $\{$`r=i;w=w';h=m`$\}$ corresponds to the subtree $\mathrm{d}_{w'}(u_i)$ (where we take the convention that $t = u_0$). We use the '.' notation to refer to a field of a record. We consider then the data structure $\texttt{val} = \texttt{st} + \{0, 1\}$. Variables `u, v` coming next will be of that "type".

To compute the function $(\sigma_{i,b})_{i\in I, b\in\{0,1\}}$ appearing in the definition of the $(f_i)_{i\in I}$, we compose the programs:

```
zero(u){
  return 0;
}

one(u){
  return 1;
}

pi_i_j(u_1,...u_j){
  return u_i;
}

d0(u){
  if(u == 0 || u == 1 || u.h = 0)  return u;
   else  return [r=u.r;w=u.w 0;h= u.h-1];
}

d1(u){
  if(u == 0 || u == 1 || u.h == 0) return u;
  else return [r=u.r;w=u.w 1;h= u.h-1];
}

cond(u_b,u_0,u_1,u_s){
  if (u_b == 0 ||
     (u_b.h == 0 && last-bit(u_b.w) == 0))
     return u_0;
  elseif (u_b == 1||
         (u_b.h == 0 && last-bit(u_b.w) == 1))
             return u_1;
  else  return u_s;
}
```

Then we compute the values of i and the $\bar{u}$ in $g_i(c, \bar{u})$ corresponding to the computation of the k-th bits of the output. Take $d + 1$ the maximal arity of functions in $(f_i)_{i\in I}$. To simplify the writing, we take it (wlog) as a common arity for all functions.

```
G(i,w,u_0,...,u_d){
 //u_0 corresponds to t,
   if(w == epsilon) {
   return(i,u_0,...,u_d);
   }
   else{
   a := pop(w); //get the first letter of w
   w := tail(w); //remove the first letter to w
   switch(i,a){//i in I, a in {0,1}
```

```
        case (i1,0):
        v_0 = d_0(u_0);
        foreach 1 <= k <= d:
              v_k = sigma_i1_0_k(u_0,...,u_d);
              //use the sigma defined above
              next_i = p_i1_0;
              //the map p is hard-encoded
        break;
        .
        .
        .
        case (im,1):
        v_0 = d_1(u_0);
        foreach 1 <= k <= d:
              v_k = sigma_im_1_k(u_0,...,u_d);
              next_i = p_im_1;
        break;
       }
       return G(next_i,w,d_a(u_0),v_1,...,v_d);
  }
}
```

Observe that this program is a tail recursive program. As a consequence, to compute it, one needs only to store the recurrence arguments, that is a finite number of variables. Since the value of these latter variables can be stored in logarithmic space, the computation itself can be performed within the bound. Finally, the program returns the name `i` of the circuit that must be build, a pointer on each of the inputs of the circuit with their size. It is then routine to build the corresponding circuit at the corresponding position `w`. □

6 Conclusion

We established an implicit characterization of the parallel classes of complexity NC^k, for $k \geq 1$, which does not require any sorted framework. This result was achieved introducing a restricted form of simultaneous recursion with substitution (MIP) which does not obey to any kind of explicit bounds.

The potential of MIP to perform parallel operations with low complexity cost (see Lemma 17) makes this scheme not only the key idea of the given characterization of NC^k, but also a promising tool for further application. In particular, one may expect to reach AC^k using a MIP like schema. This is work in progress.

References

[1] D. Aspinall, L. Beringer, M. Hofmann, H.-W. Loidl, and A. Momigliano. A program logic for resources. *Theor. Comput. Sci.*, 389(3):411–445, 2007.

[2] J. L. Balcázar, J. Díaz, and J. Gabarró. *Structural complexity II*, volume 22 of *EATCS Monographs of Theoretical Computer Science*. Springer, 1990.

[3] S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the poly-time functions. *Computational Complexity*, 2:97–110, 1992.

[4] S. Bellantoni and I. Oitavem. Separating NC along the δ axis. *Theoretical Computer Science*, 318:57–78, 2004.

[5] S. Bloch. Function-algebraic characterizations of log and polylog parallel time. *Computational Complexity*, 4(2):175–205, 1994.

[6] G. Bonfante, R. Kahle, J.-Y. Marion, and I. Oitavem. Towards an implicit characterization of NC^k. In Z. Èsik, editor, *Computer Science Logic '06*, volume 4207 of *Lecture Notes in Computer Science*, pages 212–224. Springer, 2006.

[7] G. Bonfante, J.-Y. Marion, and R. Péchoux. A characterization of alternating log time by first order functional programs. In *LPAR*, pages 90–104, 2006.

[8] A. K. Chandra, D. J. Kožen, and L. J. Stockmeyer. Alternation. *Journal ACM*, 28:114–133, 1981.

[9] P. Clote. Sequential, machine independent characterizations of the parallel complexity classes *ALogTIME*, AC^k, NC^k and NC. In S. Buss and P. Scott, editors, *Feasible Mathematics*, pages 49–69. Birkhäuser, 1990.

[10] 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.

[11] J.-Y. Girard. Light linear logic. *Information and Computation*, 143(2):175–204, 1998.

[12] N. Immerman. *Descriptive Complexity*. Springer, 1998.

[13] L. Kristiansen and N. D. Jones. The flow of data and the complexity of algorithms. In S. B. Cooper, B. Löwe, and L. Torenvliet, editors, *CiE*, volume 3526 of *Lecture Notes in Computer Science*, pages 263–274. Springer, 2005.

[14] D. Leivant. A foundational delineation of computational feasiblity. In *Proceedings of the Sixth IEEE Symposium on Logic in Computer Science (LICS'91)*, 1991.

[15] D. Leivant. Predicative recurrence and computational complexity I: Word recurrence and poly-time. In P. Clote and J. Remmel, editors, *Feasible Mathematics II*, pages 320–343. Birkhäuser, 1994.

[16] D. Leivant. A characterization of NC by tree recursion. In *Foundations of Computer Science 1998*, pages 716–724. IEEE Computer Society, 1998.

[17] D. Leivant and J.-Y. Marion. A characterization of alternating log time by ramified recurrence. *Theoretical Computer Science*, 236(1–2):192–208, 2000.

[18] J.-Y. Marion. Predicative analysis of feasibility and diagonalization. In *TLCA*, volume 4583 of *Lecture Notes in Computer Science*, 2007.

[19] K.-H. Niggl and H. Wunderlich. Certifying polynomial time and linear/polynomial space for imperative programs. *SIAM J. Comput.*, 35(5):1122–1147, 2006.

[20] I. Oitavem. Characterizing NC with tier 0 pointers. *Mathematical Logic Quarterly*, 50:9–17, 2004.

[21] W. L. Ruzzo. On uniform circuit complexity. *Journal of Computer and System Sciences*, 22:365–383, 1981.

[22] H. Simmons. The realm of primitive recursion. *Archive for Mathematical Logic*, 27:177–188, 1988.