A Characterization of Alternating Log Time by First Order Functional Programs

Guillaume Bonfante, Jean-Yves Marion, and Romain Péchoux

Loria-INPL, École Nationale Supérieure des Mines de Nancy, B.P. 239, 54506 Vandoeuvre-lès-Nancy Cedex, France
Guillaume.Bonfante@loria.fr, Jean-Yves.Marion@loria.fr, Romain.Pechoux@loria.fr

Abstract. We a give an intrinsic characterization of the class of functions which are computable in NC^1 that is by a uniform, logarithmic depth and polynomial size family circuit. Recall that the class of functions in *ALogTime*, that is in logarithmic time on an Alternating Turing Machine, is NC^1. Our characterization is in terms of first order functional programming languages. We define measure-tools called Sup-interpretations, which allow to give space and time bounds and allow also to capture a lot of program schemas. This study is part of a research on static analysis in order to predict program resources. It is related to the notion of Quasi-interpretations and belongs to the implicit computational complexity line of research.

1 Introduction

This study concerns interpretation methods for proving complexity bounds of first order functional programs. Such methods provide machine independent characterization of functional complexity classes, that Cobham [15] initiated. They also provide static analysis of the computational resources, which are necessary to run a program. Such an analysis should guarantee the amount of memory, time or processors which are necessary to execute a program on all inputs.

Implicit computational complexity (ICC) proposes syntactic characterizations of complexity classes, which lean on a data ramification principle like safe recursion [7], lambda-calculus [26] or data tiering [24]. We mention this line of works because they are inherently fundamentals, in the sense that one has to introduce such characterizations before one can proceed with the development of further studies and applications. Here, the term ICC is use as a name for characterizations of complexity classes which are syntactic and do not explicitly refer to computational resources.

It bears stressing to discuss on the two main difficulties that we have to face in order to provide a compelling resource static analysis. The first is that the method should capture a broad class of programs in order to be useful. From a theoretical perspective, this means that we are trying to characterize a large class of programs, which represents functions in some complexity classes. Traditional results focus on capturing all functions of a complexity class and we should

M. Hermann and A. Voronkov (Eds.): LPAR 2006, LNAI 4246, pp. 90–104, 2006.

call this approach extensional whereas our approach is rather intentional. This change of point of view is difficult because we have to keep in mind that the set of polynomial time programs is Σ_2-complete. The second difficulty is related to the complexity of the static analysis suggested. The resource analysis procedure should be decidable and easily checkable. But inversely, a too "easy" resource analysis procedure won't, certainly, delineate a meaningful class of programs.

There are at least four directions inspired by ICC approaches which are related with our topic and that we briefly review. The first direction deals with linear type disciplines in order to restrict computational time and began with the seminal work of Girard [20] which defined Light Linear Logic. The second direction is due to Hofmann [21], which introduced a resource atomic type, the diamond type, into the linear type system for higher order functional programming. Unlike the two former approaches and the next one, the third one considers imperative programming language and is developed by Kristiansen-Jones [22], Niggl-Wunderlich [31], and Marion-Moyen [30].

Lastly, the fourth approach is the one on which we focus in this paper. It concerns term rewriting systems and interpretation methods for proving complexity bounds. This method consists in giving an interpretation to computed functions, which provides an upper bound on function output sizes. The method analyses the program data flow in order to measure the program complexity. We have developed two kinds of interpretation methods for proving complexity. The first method concerns Quasi-interpretations, which is surveyed in [10]. The second method, which concerns this paper, is the *sup-interpretation method*, that we introduced in [29]. The main features of interpretation methods for proving complexity bounds are the following.

1. The analysis include broad classes of algorithms, like greedy algorithms, dynamic programming [28] and deal with non-terminating programs [29].
2. Resource verification of bytecode programs is obtained by compiling first order functional and reactive programs. See for example [3,2,18].
3. There are heuristics to determine program complexity. See [1,11]

1.1 Backgrounds on *ALogTime* and NC^1

We write $\log(n)$ to mean $\lceil \log_2(n+1) \rceil$. Recall that the floor function $\lfloor x \rfloor$ is the greatest integer $\leq x$, and the ceiling function $\lceil x \rceil$ is least integer $\geq x$.

We refer to *Random Access Alternating Turing Machine* of [13], called ATM. An ATM has random access read only input tapes as well as work tapes. The states of the ATM are classified as either conjunctive, disjunctive or reading. The computation of an ATM proceeds in two stages. The first stage consists in spawning two successor configurations from a root configuration. The second stage consists in evaluating backward the configuration tree generated in the first stage. An ATM outputs a single bit. A function $F : \{0,1\}^* \to \{0,1\}^*$ is bitwise computable in *ALogTime* if the function $F_{bit} : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}$ is computable by an ATM in time $O(\log(n))$. The function F_{bit} is defined by $F_{bit}(x,u)$ is equal to the i'th bit of $F(x)$, where i is the integer that u represents

in binary. Following Cook [17], we say that a function $F : \{0,1\}^* \to \{0,1\}^*$ is computed in *ALogTime* if ϕ is bitwise computable in *ALogTime* and ϕ is polynomially bounded.

A circuit C_n is a directed acyclic graph built up from Boolean gates *And*, *Or* and *Not*. Each gate has an in-degree less or equal to two. A circuit has n input nodes and $g(n)$ output nodes, where $g(n) = O(n^c)$ for some constant $c \geq 1$. Thus, a circuit C_n computes a function $f_n : \{0,1\}^n \to \{0,1\}^{g(n)}$. A circuit family is a sequence of Boolean circuits $C = (C_n)_n$, which computes a family of finite functions (f_n) over $\{0,1\}^*$. Inversely a function f is computed by a circuit family $(C_n)_n$ if the restriction of f to inputs of size n is computed by C_n. The complexity of a circuit depends on its height (that is the longest path from an input to an output gate) and its size (that is the number of gates).

The class NC^1 functions is the set of functions which are computed by U_{E^*}-uniform circuit families of polynomial size (i.e. bounded $O(n^d)$ for some degree d) and of depth $O(\log(n))$ where n is the circuit input length.

NC^1 contains functions associated with binary addition, subtraction, and more generally prefix sum of associative operators. Buss [12] showed that the evaluation of Boolean formulae is a complete problem for NC^1. The class NC^1 contains functions which are computed by very fast parallel algorithms.

Uniformity condition ensures that there is a procedure which, given n, produces a description of the circuit C_n. All along, we shall consider U_{E^*}-uniform family of circuits, which is sufficient one to establish the equivalent Theorem 1. Barrington, Immerman and Straubing [6] studied other equivalent uniform conditions. The U_{E^*}-uniformity condition is the following. The extended connection language L_{EC} of $C = (C_n)_n$ is a set of quadruplets (n, g, p, y) where the gate indicated by the path p from the gate numbered g is of type y in C_n. For NC^1, knowing whether an element is in the extended connection language L_{EC} for C is decidable in time $O(\log(n))$ by an ATM.

In [32], Ruzzo demonstrated the following equivalence.

Theorem 1. *A function $\phi : \{0,1\}^* \to \{0,1\}^*$ is in NC^1 if and only if ϕ is computed in ALogTime.*

The class NC^1 is included in the class Logspace, and so in the Ptime. Furst, Saxe and Spiser [19] and Atjai [5] established that AC^0 is strictly included in NC^1. Following [6] opinion, NC^1 is at the frontier where we begin to have some separation results, which is a motivation to study NC^1.

1.2 Results and Related Works

We consider a first order functional programming language over constructor term algebra. We define a class of programs that we call *explicitly additive arboreal* programs. We demonstrate that functions, which are computable by these programs, are exactly the functions computed in *ALogTime*. That is, they are computable in NC^1. To our knowledge, this is the first result, which connects a small class of parallel functions and term rewriting systems.

There are various characterizations of *ALogTime*, which are surveyed in [14] based on bounded recursion schema. Compton and Laflamme [16] give a characterization of *ALogTime* based on finite global functions. These results are clearly a guideline for us. However, there are only a few characterizations of *ALogTime* from which a resource static analysis is conceivable. Bloch [8] gives a characterization of *ALogTime* using a divide and conquer ramified recursion schema. Leivant and Marion [27] propose another characterization based on linear ramified recursion with substitutions. It is also worth mentioning [25,9] which capture *NC*. These purely syntactic characterizations capture a few algorithmic patterns. On the contrary, this work tries to delineate a broad class of algorithms. Parallel algorithms are difficult to design. Employing the sup-interpretation method leads to delineate efficient parallel programs amenable to circuit computing. Designing parallel implementations of first order functional programs with interpretation methods for proving complexity bounds, might be thus viable in the near future.

2 First Order Functional Programming

2.1 Syntax of Programs

We define a generic first order functional programming language. The vocabulary $\Sigma = \langle Cns, Op, Fct \rangle$ is composed of three disjoint domains of symbols. The arity of a symbol is the number n of arguments that it takes. The program grammar is the following.

$$
\begin{array}{lll}
\texttt{(Constructor terms)} & \mathcal{T}(Cns) \ni v & ::= \mathbf{c} \mid \mathbf{c}(v_1, \cdots, v_n) \\
\texttt{(terms/Expressions)} & \mathcal{T}(Cns, Fct, Var) \ni t & ::= \mathbf{c} \mid x \mid \mathbf{c}(t_1, \cdots, t_n) \\
 & & \mid \mathbf{op}(t_1, \cdots, t_n) \mid \mathtt{f}(t_1, \cdots, t_n) \\
\texttt{(patterns)} & Patterns \ni p & ::= \mathbf{c} \mid x \mid \mathbf{c}(p_1, \cdots, p_n) \\
\texttt{(rules)} & \mathcal{R} \ni r & ::= \mathtt{f}(p_1, \cdots, p_n) \rightarrow e^{\mathtt{f}}
\end{array}
$$

where $\mathbf{c} \in Cns$ is a constructor, $\mathbf{op} \in Op$ is an operator, $\mathtt{f} \in Fct$ is a function symbol. The set of variables Var is disjoint from Σ and $x \in Var$. In a rule, a variable of $e^{\mathtt{f}}$ occurs in the patterns $p_1, \cdots, p_n$ of the definition of $\mathtt{f}$. A program $\mathbf{p}$ is a list of rules. The program's main function symbol is the first function symbol in the program's list of rules. Throughout, we consider only orthogonal programs, that is, rule patterns are disjoint and linear. So each program is confluent.

Throughout, we write $\overline{e}$ to mean a sequence of expressions, that is $\overline{e} = e_1, \ldots, e_n$, for some n clearly determined by the context.

2.2 Semantics

The domain of computation of a program $\mathbf{p}$ is the constructor algebra $\mathtt{Values} = \mathcal{T}(Cns)$. Put $\mathtt{Values}^* = \mathtt{Values} \cup \{\mathbf{Err}\}$ where $\mathbf{Err}$ is the value associated when an error occurs. An operator $\mathbf{op}$ of arity n is interpreted by a function $[\![\mathbf{op}]\!]$ from $\mathtt{Values}^n$ to $\mathtt{Values}^*$. Operators are essentially basic partial functions like destructors or characteristic functions of predicates like $=$.

The language has a usual closure-based call-by-value semantics which is displayed in Figure 1. The computational domain is $\texttt{Values}^{\#} = \texttt{Values} \cup \{\mathbf{Err}, \bot\}$ where $\bot$ means that a program is non-terminating. A program $\mathbf{p}$ computes a partial function $[\![\mathbf{p}]\!] : \texttt{Values}^n \to \texttt{Values}^{\#}$ defined as follows. For all $v_i \in \texttt{Values}$, $[\![\mathbf{p}]\!](v_1, \cdots, v_n) = w$ iff $\mathbf{p}(v_1, \cdots, v_n) \downarrow w$. Otherwise $[\![\mathbf{p}]\!](v_1, \cdots, v_n) = \bot$. The meaning of $e \downarrow w$ is that e evaluates to the value w of $\texttt{Values}$. By definition, if no rule is applicable, then an error occurs and $e \downarrow \mathbf{Err}$.

A substitution σ is a finite function from variables to $\texttt{Values}$. The application of a substitution σ to a term e is noted $e\sigma$.

$$\frac{t_1 \downarrow w_1 \ldots t_n \downarrow w_n}{\mathbf{c}(t_1, \cdots, t_n) \downarrow \mathbf{c}(w_1, \cdots, w_n)}\ \mathbf{c} \in \mathit{Cns} \text{ and } \forall i, w_i \neq \mathbf{Err}$$

$$\frac{t_1 \downarrow w_1 \ldots t_n \downarrow w_n}{\mathbf{op}(t_1, \cdots, t_n) \downarrow [\![\mathbf{op}]\!](w_1, \cdots, w_n)}\ \mathbf{op} \in \mathit{Op}$$

$$\frac{t_1 \downarrow w_1 \ldots t_n \downarrow w_n \quad \mathtt{f}(p_1, \cdots, p_n) \to e \quad e\sigma \downarrow w}{\mathtt{f}(t_1, \cdots, t_n) \downarrow w}\ \text{where } \sigma(x_i) = w_i\ \forall i = 1, \ldots, n$$

Fig. 1. Call by value semantics of ground terms wrt a program $\mathbf{p}$

3 Sup-interpretations

Let us now turn our attention to the sup-interpretation method which is the main tool to analyze a program complexity. For this purpose, we define a special kind of program interpretation called *sup-interpretation*, which is associated to a *lightweight*, to provide a complexity measure.

3.1 Partial Assignments

A partial assignment $\mathcal{I}$ is a partial mapping from a vocabulary Σ such that for each symbol $\mathtt{f}$ of arity n, in the domain of $\mathcal{I}$, it yields a partial function $\mathcal{I}(f) : (\mathbb{R}_+)^n \longmapsto \mathbb{R}_+$, where $\mathbb{R}_+$ is the set of non-negative real numbers. The domain of a partial assignment $\mathcal{I}$ is noted $\mathrm{dom}(\mathcal{I})$. Because it is convenient, we shall always assume that partial assignments that we consider, are defined on constructors and operators. That is $\mathit{Cns} \cup \mathit{Op} \subseteq \mathrm{dom}(\mathcal{I})$.

An expression e is defined over $\mathrm{dom}(\mathcal{I})$ if each symbol belongs to $\mathrm{dom}(\mathcal{I})$ or is a variable of Var. Assume that an expression e is defined over $\mathrm{dom}(\mathcal{I})$ and has n variables. Take a denumerable sequence $X_1, \ldots, X_n, \ldots$. The partial assignment of e wrt $\mathcal{I}$ is the homomorphic extension that we write $\mathcal{I}^*(e)$. It denotes a function from $\mathbb{R}_+^n$ to $\mathbb{R}_+$ and is defined as follows:

1. If x_i is a variable of *Var*, let $\mathcal{I}^*(x_i) = X_i$
2. If b is a 0-ary symbol of Σ, then $\mathcal{I}^*(b) = \mathcal{I}(b)$.
3. If f is a symbol of arity $n > 0$ and $e_1, \cdots, e_n$ are expressions, then

$$\mathcal{I}^*(f(e_1, \cdots, e_n)) = \mathcal{I}(f)(\mathcal{I}^*(e_1), \ldots, \mathcal{I}^*(e_n))$$

3.2 Sup-interpretations and Lightweights

Definition 1 (Sup-interpretation). *A sup-interpretation is a partial assignment θ which verifies the three conditions below :*

1. *The assignment θ is weakly monotonic. That is, for each symbol $f \in dom(\theta)$, the function $\theta(f)$ satisfies for every $i = 1, \ldots, n$*

$$X_i \geq Y_i \Rightarrow \theta(f)(X_1, \cdots, X_n) \geq \theta(f)(Y_1, \cdots, Y_n)$$

2. *For each $v \in$ `Values`,*

$$\theta^*(v) \geq |v|$$

The size of an expression e is noted $|e|$ and is defined by $|\mathbf{c}| = 0$ where $\mathbf{c}$ is a 0-ary symbol and $|\mathbf{b}(e_1, \ldots, e_n)| = 1 + \sum_i |e_i|$ where $\mathbf{b}$ is a n-ary symbol.

3. *For each symbol $f \in dom(\theta)$ of arity n and for each value $v_1, \ldots, v_n$ of `Values`, if $[\![f]\!](v_1, \ldots, v_n)$ is defined, that is $[\![f]\!](v_1, \ldots, v_n) \in$ `Values`, then*

$$\theta^*(f(v_1, \ldots, v_n)) \geq \theta^*([\![f]\!](v_1, \ldots, v_n))$$

An expression e admits a sup-interpretation $\theta^*(e)$, wrt θ, if e is defined over $\mathrm{dom}(\theta)$. Intuitively, the sup-interpretation is a special program interpretation. Instead of yielding the program denotation, a sup-interpretation provides an approximation from above of the size of the outputs of the function denoted by the program.

Lemma 1. *Let e be an expression with no variable and which admits a sup-interpretation θ. Assume that $[\![e]\!]$ is defined, that is $[\![e]\!] \in$ `Values`. We then have (i) $\theta^*([\![e]\!]) \leq \theta^*(e)$ and (ii) $|[\![e]\!]| \leq \theta^*(e)$.*

Example 1. We illustrate the notion of sup-interpretation by a function, which divides by two a number. For this, we define the set of tally numbers thus,

$$\texttt{Uint} = \mathbf{0} \mid \mathbf{S}(\texttt{Uint})$$

We note $\overline{n} = \mathbf{S}^n(\mathbf{0})$. Next, we define the function $[\![\texttt{half}]\!]$ such that $[\![\texttt{half}]\!] = \lfloor \frac{\overline{n}}{2} \rfloor$ by the program below.

$$\texttt{half}(\mathbf{0}) \to \mathbf{0} \qquad \texttt{half}(\mathbf{S}(\mathbf{0})) \to \mathbf{0} \qquad \texttt{half}(\mathbf{S}(\mathbf{S}(y))) \to \mathbf{S}(\texttt{half}(y))$$

Now, a sup-interpretation of $\mathbf{0}$ is $\theta(\mathbf{0}) = 0$ and a sup-interpretation of $\mathbf{S}$ is $\theta(\mathbf{S})(X) = X+1$. Clearly, for any n, $\theta^*(\overline{n}) \geq |\overline{n}| = n$. Then, we set $\theta(\texttt{half})(X) = \lfloor \frac{X}{2} \rfloor$, which is a monotonic function. We check that condition (3) of Definition 1 is satisfied because $\theta^*(\texttt{half}(\overline{n})) = \lfloor \frac{\overline{n}}{2} \rfloor$. Notice that such a sup-interpretation is not a quasi-interpretation (a fortiori not an interpretation for proof termination) since it violates the subterm property.

We end by defining lightweights which are used to control the depth of recursive data-flows.

Definition 2 (Lightweight). *A lightweight ω is a partial assignment which ranges over Fct. To a given function symbol $\mathtt{f}$ of arity n it assigns a total function $\omega_{\mathtt{f}}$ from $\mathbb{R}_+^n$ to $\mathbb{R}_+$ which is weakly monotonic.*

3.3 Additive Assignments

Definition 3. *A partial assignment $\mathcal{I}$ is additive if*

1. *For each symbol f of arity n in $dom(\mathcal{I})$, $\mathcal{I}(f)$ is bounded by a polynomial of $\mathbb{R}_+[X_1, \cdots, X_n]$.*
2. *For each constructor $\mathbf{c} \in dom(\theta)$ of arity > 0,*

$$\theta(\mathbf{c})(X_1, \cdots, X_n) = \sum_{i=1}^{n} X_i + \alpha_{\mathbf{c}} \qquad \alpha_{\mathbf{c}} \geq 1$$

Lemma 2. *Assume that $\mathcal{I}$ is an additive assignment. There is a constant α such that for each value $\boldsymbol{u}$ of $\mathtt{Values}$, the following inequality is satisfied : $|\boldsymbol{u}| \leq \theta^*(\boldsymbol{u}) \leq \alpha \times |\boldsymbol{u}|$*

Throughout the following paper we consider sup-interpretations and lightweights, which are additive assignments.

4 Arboreal Programs

4.1 Fraternities

Given a program $\mathbf{p}$, we define *precedence* $\geq_{Fct}$ on function symbols. Set $\mathtt{f} \geq_{Fct} \mathtt{g}$ if there is a $\mathbf{p}$-rule $\mathtt{f}(p_1, \cdots, p_n) \to e$ and $\mathtt{g}$ is in e. Then, take the reflexive and transitive closure of $\geq_{Fct}$, also noted $\geq_{Fct}$. Next, we define $\mathtt{f} \approx_{Fct} \mathtt{g}$ and $\mathtt{f} >_{Fct} \mathtt{g}$ as usual. We define a rank function rk as a morphism from $(Fct, \geq_{Fct})$ into $(\mathbb{N}, \geq)$, so satisfying : $\mathrm{rk}(\mathtt{g}) < \mathrm{rk}(\mathtt{f})$, if $\mathtt{f} \geq_{Fct} \mathtt{g}$, and $\mathrm{rk}(\mathtt{f}) = \mathrm{rk}(\mathtt{g})$, if $\mathtt{f} \approx_{Fct} \mathtt{g}$.

A *context* is an expression $\mathsf{C}[\diamond_1, \cdots, \diamond_r]$ containing one occurrence of each $\diamond_i$. Here, we suppose that the $\diamond_i$'s are new symbols which are neither in Σ nor in *Var*. The substitution of each $\diamond_i$ by an expression d_i is noted $\mathsf{C}[d_1, \cdots, d_r]$.

Definition 4. *Given a program $\boldsymbol{p}$, a term $\mathsf{C}[\mathtt{g}_1(\overline{t_1}), \ldots, \mathtt{g}_r(\overline{t_r})]$ is a* fraternity *activated by $\mathtt{f}(p_1, \cdots, p_n)$ iff*

1. *There is a rule $\mathtt{f}(p_1, \cdots, p_n) \to \mathsf{C}[\mathtt{g}_1(\overline{t_1}), \ldots, \mathtt{g}_r(\overline{t_r})]$.*
2. *For each $i \in \{1, r\}$, $\mathtt{g}_i \approx_{Fct} \mathtt{f}$.*
3. *For every function symbol $\mathtt{h}$ in the context $\mathsf{C}[\diamond_1, \cdots, \diamond_r]$, $\mathtt{f} >_{Fct} \mathtt{h}$.*

4.2 Arboreal Programs

Definition 5 (Arboreal). *A program* $\boldsymbol{p}$ *admits an* arboreal *sup-interpretation iff there is a sup-interpretation* θ*, a lightweight* ω *and a constant* $K > 1$ *such that for every fraternity* $\mathsf{C}[\boldsymbol{g}_1(\overline{t_1}), \ldots, \boldsymbol{g}_r(\overline{t_r})]$ *activated by* $\boldsymbol{f}(p_1, \cdots, p_n)$*, and any substitutions* σ*, both conditions are satisfied:*

$$\omega_f(\theta^*(p_1\sigma), \ldots, \theta^*(p_n\sigma)) > 1 \tag{1}$$

$$\omega_f(\theta^*(p_1\sigma), \ldots, \theta^*(p_n\sigma)) \geq K \times \omega_{g_i}(\theta^*(t_{i,1}\sigma), \ldots, \theta^*(t_{i,m}\sigma)) \quad \forall 1 \leq i \leq r \tag{2}$$

The constant K *is called the arboreal coefficient of* $\boldsymbol{p}$*.*

Example 2. We show how to compute prefix sum, which is one of the canonical examples of an efficient parallel circuit computation. Suppose that $\bigodot$ is a binary associative operation over A. The prefix sum of a list $[x_1, \ldots, x_n]$ of elements of A, is $x_1 \bigodot \ldots \bigodot x_n$. Lists over A are defined as usual

$$\mathtt{List}(A) = [\,] \mid [A, \mathtt{List}(A)]$$

We take two operators **Left** and **Right**, which cut a list in two half.

$$\mathbf{Left}([\,]) = [\,] \qquad \mathbf{Left}([x_1, \ldots, x_n]) = [x_1, \ldots, x_{\lfloor \frac{n}{2} \rfloor}]$$

$$\mathbf{Right}([\,]) = [\,] \qquad \mathbf{Right}([x_1, \ldots, x_n]) = [x_{\lfloor \frac{n}{2} \rfloor + 1}, \ldots, x_n]$$

We write $[x_1, \ldots, x_n]$ instead of $[x_1, [x_2, \ldots, x_n]]$. Now, the prefix sum of a list is computed as follows.

$$\mathtt{sum}([x]) = x$$

$$\mathtt{sum}([x, y, L]) = \mathtt{sum}(\mathbf{Left}([x, y, L])) \bigodot \mathtt{sum}(\mathbf{Right}([x, y, L]))$$

Here, we consider $\bigodot$ as an infix operator using familiar conventions. Actually, the pattern $[x, y, L]$ captures a list of length at least 2.

The constructors and the operators admit the following sup-interpretations.

$$\theta([\,]) = 0 \qquad \theta([X, L]) = X + L + 1$$

$$\theta(\mathbf{Left})(N) = \lfloor \frac{N}{2} \rfloor \qquad \theta(\mathbf{Right})(N) = \lceil \frac{N}{2} \rceil$$

Indeed, since the size of a list is the number of its elements, we see that for any list L, we have $|L| = \theta(L)$. We might also check that $|[\![\mathbf{Left}(L)]\!]| \leq \theta(\mathbf{Left}(L))$ and $|[\![\mathbf{Right}(L)]\!]| \leq \theta(\mathbf{Right}(L))$. Next, `sum` satisfies the arboreal condition by taking $\omega_{\mathtt{sum}}(L) = L$ and $K = \frac{3}{2}$ (Hint : $L \geq 2$). Lastly, we shall see in a short while that `sum` is an example of an explicitly additive arboreal program.

We shall now show that a program admitting an arboreal sup-interpretation is terminating. Actually, the termination of an arboreal program may be established by the dependency pair method of Arts and Giesl [4], or by the size change principle for program termination of Lee, Jones and Ben-Amram [23]. However, it is worth to have a direct demonstration in order to establish an upper bound on derivation lengths.

4.3 Weighted Call-Trees

We now describe the notion of call-trees which is a representation of a program state transition sequences. Next, we show that, when we consider arboreal programs, we can assign weights to state transitions in such way that a state transition sequence is associated to a sequence of strictly decreasing weights. Lastly, weights provide a measure which gives us an upper bound on derivation lengths.

Call-Trees. Suppose that we have a program $\mathbf{p}$. A *state* $\langle \mathtt{f}, \mathbf{u}_1, \cdots, \mathbf{u}_n \rangle$ of $\mathbf{p}$ is a tuple where $\mathtt{f}$ is a function symbol of arity n and $\mathbf{u}_1, \cdots, \mathbf{u}_n$ are values of $\mathtt{Values}^*$.

A *state transition* of $\mathbf{p}$ is a triplet $\eta_1 \rightsquigarrow \eta_2$ between two states $\eta_1 = \langle \mathtt{f}, \mathbf{u}_1, \ldots, \mathbf{u}_n \rangle$ and $\eta_2 = \langle \mathtt{g}, \mathbf{v}_1, \cdots, \mathbf{v}_m \rangle$ where

1. $\mathtt{f}(p_1, \cdots, p_n) \to e$ is a rule of $\mathbf{p}$
2. there is a substitution σ such that $p_i\sigma \downarrow \mathbf{u}_i$ for any $1 \leq i \leq n$,
3. $e = \mathsf{C}[\mathtt{g}(d_1, \cdots, d_m)]$ and for any $1 \leq i \leq m$, $d_i\sigma \downarrow \mathbf{v}_i$

We write $\overset{*}{\rightsquigarrow}$ to mean the transitive closure of $\rightsquigarrow$. We define the $\langle \mathtt{f}, \mathbf{u}_1, \cdots, \mathbf{u}_n \rangle$ call-tree as a tree where (i) the set of nodes are labeled by states of $\{\eta \mid \langle \mathtt{f}, \mathbf{u}_1, \cdots, \mathbf{u}_n \rangle \overset{*}{\rightsquigarrow} \eta\}$, (ii) there is an edge between two nodes if there is a transition between both states, which labels the nodes. (iii) the root is a node labeled by the state $\langle \mathtt{f}, \mathbf{u}_1, \cdots, \mathbf{u}_n \rangle$.

A $\langle \mathtt{f}, \mathbf{u}_1, \cdots, \mathbf{u}_n \rangle$ call-tree may be an infinite tree. In this case, König's Lemma implies that there is a reduction strategy which leads to a infinite sequence of reductions.

Weighted Call-Trees. Throughout, it is convenient to use $\theta^*(\overline{v_j})$ to abbreviate $\theta^*(v_{j,1}), \ldots, \theta^*(v_{j,n})$. Given a sup-interpretation θ and a lightweight ω of a program, we assign to each state transition a weight, which is a pair (p, q) in $\mathbb{N} \cup \{\bot\} \times \mathbb{N} \cup \{\bot\}$ as follows. We have $\eta_1 = \langle \mathtt{f}, \mathbf{u}_1, \cdots, \mathbf{u}_n \rangle \overset{(p,\,q)}{\rightsquigarrow} \eta_2 = \langle \mathtt{g}, \mathbf{v}_1, \cdots, \mathbf{v}_m \rangle$ iff

- If $\mathtt{f} >_{Fct} \mathtt{g}$, then $(p, q) = (\mathrm{rk}(\mathtt{f}), 0)$.
- If $\mathtt{f} \approx_{Fct} \mathtt{g}$ and $\omega_{\mathtt{f}}(\theta^*(\overline{\mathbf{u}})) \geq 1$, then $(p, q) = (\mathrm{rk}(\mathtt{f}), \lceil \log_K(\omega_{\mathtt{f}}(\theta^*(\overline{\mathbf{u}})))\rceil)$
- Otherwise, $(p, q) = (\bot, \bot)$.

In the two first cases above, the weight is said to be defined,

Lemma 3. *Assume that $\mathbf{p}$ admits an arboreal sup-interpretation. The weight which is assigned to each state transition of $\mathbf{p}$ is defined.*

Proof. It suffices to prove that when $\mathtt{f} \approx_{Fct} \mathtt{g}$, we have $\omega_{\mathtt{f}}(\theta^*(\mathbf{u}_1), \ldots, \theta^*(\mathbf{u}_n)) \geq 1$. Since $\mathbf{p}$ admits an arboreal sup-interpretation, the situation is the following. $\mathtt{f}(\mathbf{u}_1, \cdots, \mathbf{u}_n)$ matches a unique rule $\mathtt{f}(p_1, \cdots, p_n) \to e$ because $\mathbf{p}$ is orthogonal. By definition of a state transition, $\mathtt{g}$ is in e. Since $\mathtt{f} \approx_{Fct} \mathtt{g}$, e is a fraternity activated by $\mathtt{f}(p_1, \cdots, p_n)$ such that $e = \mathsf{C}[\ldots, \mathtt{g}(\ldots), \ldots]$. Therefore, the condition (1) of Definition 5 holds, which completes the proof.

Intuitively, the weight associated to a transition indicates what is decreasing. In fact, there is two possibilities. In the first one, the function rank is strictly decreasing. In the second one, it is the lightweight which is strictly decreasing.

Theorem 2. *Assume that the program* $\boldsymbol{p}$ *admits an arboreal sup-interpretation. Then* $\boldsymbol{p}$ *is terminating. That is, for every function symbol* $\mathtt{f}$ *and for any values* $\boldsymbol{u}_1, \cdots, \boldsymbol{u}_n$ *in* `Values`, $[\![f]\!](\boldsymbol{u}_1, \cdots, \boldsymbol{u}_n)$ *is in* `Values`*.

Proof. Let $\langle \mathtt{f}, \mathbf{u}_1, \cdots, \mathbf{u}_n \rangle$ be a state of $\mathbf{p}$. Take a branch of the $\langle \mathtt{f}, \mathbf{u}_1, \cdots, \mathbf{u}_n \rangle$ call-tree $\eta_0 \overset{(p_0,q_0)}{\rightsquigarrow} \eta_1 \overset{(p_1,q_1)}{\rightsquigarrow} \eta_2 \overset{(p_2,q_2)}{\rightsquigarrow} \dots$ where $\eta_j = \langle \mathtt{f}_j, \overline{v_j} \rangle$.

We define an ordering on $\mathbb{N} \times \mathbb{N}$ by $(n,m) < (p,q)$ if $n < p$ or $n = p$ and $m < q$. We show that for any i such that $\eta_i \overset{(p_i,q_i)}{\rightsquigarrow} \eta_{i+1} \overset{(p_{i+1},q_{i+1})}{\rightsquigarrow} \eta_{i+2}$, we have $(p_i, q_i) > (p_{i+1}, q_{i+1})$. There are three cases to examine.

1. Suppose that $\mathtt{f}_i >_{Fct} \mathtt{f}_{i+1}$. Then, we have $p_i = \mathrm{rk}(\mathtt{f}_i) > p_{i+1} = \mathrm{rk}(\mathtt{f}_{i+1})$.
2. Suppose that $\mathtt{f}_i \approx_{Fct} \mathtt{f}_{i+1}$ and $\mathtt{f}_{i+1} >_{Fct} \mathtt{f}_{i+2}$. We have $p_i = \mathrm{rk}(\mathtt{f}_i) = p_{i+1} = \mathrm{rk}(\mathtt{f}_{i+1})$ and $q_i = \lceil \log_K(\omega_{\mathtt{f}_i}(\theta^*(\overline{v_i}))) \rceil > q_{i+1} = 0$, since $\omega_{\mathtt{f}_i}(\theta^*(\overline{v_i})) > 1$
3. Suppose that $\mathtt{f}_i \approx_{Fct} \mathtt{f}_{i+1}$ and $\mathtt{f}_{i+1} \approx_{Fct} \mathtt{f}_{i+2}$. As in the previous case, we have $p_i = p_{i+1}$. Now, we also have $q_i = \lceil \log_K(\omega_{\mathtt{f}_i}(\theta^*(\overline{v_i}))) \rceil > q_{i+1} = \lceil \log_K(\omega_{\mathtt{f}_{i+1}}(\theta^*(\overline{v_{i+1}}))) \rceil$. Indeed intuitively, each recursive state corresponds to the division of its lightweight by the arboreal constant $K > 1$. Formally, Condition (2) of Definition 5 claims that

$$\omega_{\mathtt{f}_i}(\theta^*(\overline{v_i})) \geq K \times \omega_{\mathtt{f}_{i+1}}(\theta^*(\overline{v_{i+1}}))$$
$$\lceil \log_K(\omega_{\mathtt{f}_i}(\theta^*(\overline{v_i}))) \rceil \geq \lceil \log_K(\omega_{\mathtt{f}_{i+1}}(\theta^*(\overline{v_{i+1}}))) \rceil + 1$$

In the three cases above, we have established that $(p_i, q_i) > (p_{i+1}, q_{i+1})$. Since the ordering $<$ is well-founded, the weight sequence is finite, which completes the proof.

5 Main Result

5.1 Explicitly Defined Functions

Given a program $\mathbf{p}$, a function symbol $\mathtt{f}$ is *explicitly defined* iff for each rule like $\mathtt{f}(p_1, \cdots, p_n) \to e$, the expression e is built from variables, constructors, operators and explicitly defined function symbols whose precedence is strictly less than $\mathtt{f}$. An expression e is explicit in $\mathbf{p}$ iff each function symbol occurring in e is explicitly defined in $\mathbf{p}$.

An explicit function is a function which is defined by a program in which any function symbols are explicitly defined.

Definition 6. *A program* $\boldsymbol{p}$ *is* explicitly fraternal *if and only if for each fraternity* $\mathsf{C}[\mathtt{g}_1(\overline{t_1}), \dots, \mathtt{g}_r(\overline{t_r})]$ *of* $\boldsymbol{p}$*, the context* $\mathsf{C}[\diamond_1, \cdots, \diamond_r]$ *and each* $\overline{t_i}$ *are explicitly defined in* $\boldsymbol{p}$*.*

5.2 Characterization of Alogtime

We encode the elements of $\texttt{Values}^*$ by binary words of $\{0,1\}^*$ using a mapping $code : \texttt{Values}^* \to \{0,1\}^*$ such that (i) *code* is computed in *ALogTime*, and (ii) each constructor of *Cns* is computed by an U_{E^*}-uniform, polynomial size, and constant depth circuit family wrt the encoding *code*.

A program $\mathbf{p}$ has flat operators if every operator of *Op* is computed by an U_{E^*}-uniform, polynomial size, and constant depth circuit family using the same encoding *code*.

A program $\mathbf{p}$ admits an *additive arboreal* sup-interpretation if it admits an arboreal sup-interpretation for which the sup-interpretation θ and the lightweight ω are additive assignments.

Definition 7. *A program $\boldsymbol{p}$ is* explicitly additive arboreal *if $\boldsymbol{p}$ admits an additive arboreal sup-interpretation, which is explicitly fraternal and all operators are flat.*

Given a function $\phi : \texttt{Values}^k \to \texttt{Values}$, we associate a function $\tilde{\phi} : \{0,1\}^* \to \{0,1\}$, which is defined by $\phi(\mathbf{u}) = \tilde{\phi}(code(\mathbf{u}))$, for any $\mathbf{u} \in \texttt{Values}$. A function ϕ over $\texttt{Values}$ is computed in *ALogTime* if the function $\tilde{\phi}$ is also computed in *ALogTime*.

Theorem 3. *A function ϕ over $\texttt{Values}$ is computed by a explicitly additive arboreal program if and only if ϕ is computed in ALogTime.*

Proof. It is a consequence of Lemma 7 and Lemma 9.

6 Circuit Evaluation of Exp. Add. Arboreal Programs

We now move toward an implementation of programs by uniform family of circuits. It will be appropriate to do this in several steps that we shall describe in more or less intuitive fashion. Indeed implementation details are not difficult but tedious, and will be written in the full forthcoming paper. Actually, the demonstration of Theorem 3 leans essentially on Lemmas 5 and 6.

6.1 Explicit Functions Are Constant Depth Computable

In the first step, we show that an explicit functions are computed in constant parallel time.

Lemma 4. *Assume that $\phi : \texttt{Values}^k \to \texttt{Values}^*$ is an explicit function from flat operators. Then, ϕ is computed by an U_{E^*}-uniform, polynomial size, and constant depth circuit family.*

Proof. An explicit function ϕ is defined by composition from constructors and operators. So, we complete the construction by a straightforward induction on the definition length, and by using a circuit implementation of constructors and destructors. The program which defines ϕ provides the U_{E^*}-uniformity.

6.2 Upper Bounds on Height and Size

In the second step, we establish a logarithmic upper bound on derivation lengths. Then, we show that computed values are polynomially bounded.

The height of a weighted call tree is the length of the longest branch.

Lemma 5. *Let $\boldsymbol{p}$ be an explicitly additive arboreal program. Let $\langle \boldsymbol{f}, \boldsymbol{u}_1, \cdots, \boldsymbol{u}_n \rangle$ be a state of $\boldsymbol{p}$. The height of the $\langle \boldsymbol{f}, \boldsymbol{u}_1, \cdots, \boldsymbol{u}_n \rangle$ call tree is bounded by $d \times \log(\max(|\boldsymbol{u}_1|, \ldots, |\boldsymbol{u}_n|))$ for some constant d.*

Proof. Put $n = \max_i(|\mathbf{u}_i|)$. We refine the demonstration of Theorem 2 by looking more carefully to a finite strictly decreasing sequence $(p_0, q_0) > \ldots > (p_\ell, q_\ell)$ of a branch of the $\langle \mathtt{f}, \mathbf{u}_1, \cdots, \mathbf{u}_n \rangle$ call-tree.

By definition of the partial ordering $<$ on $\mathbb{N} \times \mathbb{N}$, we have $\ell \leq (p_0+1) \times (q_0+1)$. Since, $p_0 \leq \max_{\mathtt{f}}(\mathrm{rk}(\mathtt{f}))$ and $q_0 \leq \lceil \log_K(\max_{\mathtt{f}}(\omega_{\mathtt{f}}(\theta^*(\overline{\mathbf{u}})))) \rceil$, we see that $\ell \leq \max_{\mathtt{f}}(\mathrm{rk}(\mathtt{f})) \times \lceil \log_K(\max_{\mathtt{f}}(\omega_{\mathtt{f}}(\theta^*(\overline{\mathbf{u}})))) \rceil$.

The fact that $\mathbf{p}$ admits an additive assignment implies that there is a polynomial P such that $\max_{\mathtt{f}}(\omega_{\mathtt{f}}(\theta^*(\overline{\mathbf{u}})) \leq P(\alpha \times n)$, where the constant α is given by Lemma 2. Putting altogether, there is a constant d such that $\ell \leq d \times \log_2(n)$.

Lemma 6. *Assume that $\boldsymbol{p}$ is an explicitly additive arboreal program. Then, there is a polynomial P such that for any values $\boldsymbol{u}_1, \cdots, \boldsymbol{u}_n$ and function symbol $\boldsymbol{f}$, we have*

$$|[\![\boldsymbol{f}]\!](\boldsymbol{u}_1, \cdots, \boldsymbol{u}_n)| \leq P(\max_i(|\boldsymbol{u}_i|))$$

Proof (Sketch of proof). Suppose that $\mathtt{f}$ is recursively defined, and so its computation implies fraternities. Each fraternity is explicitly defined, which means that the output size is linearly bounded by $a \times m + b$ where a and b are some constants, m is the input size, because of Lemma 4. The computation of $\langle \mathtt{f}, \mathbf{u}_1, \cdots, \mathbf{u}_n \rangle$ is made by iterating ℓ times the computation of explicit fraternity. So the output size is bounded by $a^\ell m + b \times \ell$. The length ℓ is bounded by the height of the $\langle \mathtt{f}, \mathbf{u}_1, \cdots, \mathbf{u}_n \rangle$ call-tree. By lemma 5, $\ell \leq d \times \log(\max(|\mathbf{u}_1|, \ldots, |\mathbf{u}_n|))$, for some constant d. Therefore, there is a polynomial P whose degree depends on the arboreal coefficient K and a such that $|[\![\mathtt{f}]\!](\mathbf{u}_1, \cdots, \mathbf{u}_n)| \leq P(\max_i(|\mathbf{u}_i|))$.

6.3 Programs Are in NC^1

In the third step, we construct an U_{E^*}-uniform, polynomial size, and constant depth circuit family which computes an explicitly additive arboreal program.

Lemma 7. *Suppose that a function $\phi : \mathtt{Values}^k \to \mathtt{Values}$ is defined by an explicitly additive arboreal program $\boldsymbol{p}$. Then, an U_{E^*}-uniform, polynomial size, and logarithmic depth circuit family computes $\tilde{\phi}$.*

Proof. Given an upper bound m on the input size, we construct a circuit C_m by induction on function symbol rank of $\mathbf{p}$. Actually, the depth of a circuit is bounded by $d \times \log(n)$ for some constant d because of Lemma 5. Lemma 6 states

that the size of the inputs and the outputs of each circuit layer is bounded by a polynomial. We see that circuits have a logarithmic depth and polynomial size. The U_{E^*}-uniformity condition is not too difficult to check, because the extended connection language is based on $\mathbf{p}$, which is given and on the upper-bounds obtained in the previous section.

7 Simulation of *ALogTime* Computable Functions

In this section, we prove that a function in *ALogTime* is computed by an explicitly additive arboreal program.

For this purpose, we consider the characterization [27] of *ALogTime* instead of dealing directly with ATM. There are at least two reasons to proceed in this way. The first is that it simplifies proofs which otherwise would require a lot of encodings. The second is that, as we say in the introduction, there is closed connection between ramified recursion used in implicit computational complexity and our approach.

In [27], the characterization is based on linear ramified recursion with substitution, called *LRRS*, using well-balanced trees as internal data structures.

LRRS functions compute over binary tree algebra $\mathtt{T}$. Initial functions consist of constructors, conditionals and destructors over $\mathtt{T}$. *LRRS* functions use one ramified recursion over 2 tiers, and is defined as follows.

$$\begin{aligned} \mathtt{f}(\mathbf{c}, \overline{u}; \overline{x}) &= \mathtt{g}_{\mathbf{c}}(\overline{u}; \overline{x}) && \mathbf{c} = \mathbf{0}, \mathbf{1}, \bot \\ \mathtt{f}(t\mathbf{*}t', \overline{u}; \overline{x}) &= \mathtt{g}(; \mathtt{f}(t, \overline{u}; \mathtt{h}_1(; \overline{x})), \ldots, \mathtt{f}(t', \overline{u}; \mathtt{h}_k(; \overline{x})), \overline{x}) \end{aligned}$$

where $\mathtt{g}, \mathtt{g}_c$ and the substitution functions $\mathtt{h}_1, \ldots, \mathtt{h}_k$ are previously defined functions. We separate tiers by a semicolon. A flat function is a function whose domain and range are at the same tier. A crucial point is that $\mathtt{g}$ and the substitution functions $\mathtt{h}_1, \ldots, \mathtt{h}_k$ are flat functions. Indeed, it was proved that flat functions are definable by composition of initial functions.

A function ϕ over the algebra of words $\mathtt{W} = \{\mathbf{0}, \mathbf{1}\}^*$ is said to be representable in *LRRS* if it is representable by some function f definable in *LRRS* and whose inputs represent the shortest (in the height) encoding of words by full binary trees.

Theorem 4 (Marion and Leivant). *A function $\mathtt{f}$ over $\{0,1\}^*$ is representable in LRRS if and only if it is bitwise in ALogTime and its growth is bounded by a polynomial in the size of the inputs.*

Now we are going to use this result in order to establish the completeness of our characterization:

Lemma 8. *A function ϕ which is representable in LRRS, is computed by an explicitly additive arboreal program $\boldsymbol{p}$.*

Proof (Sketch of proof). The simulation of *LRRS* functions is based on three points. The first point concerns the encoding of well balanced trees. In the simulation, we reduce *LRRS* trees into a list like in Example 2. The operator **Left** and **Right** allow to simulate well-balanced tree.

The second point is to see that a flat function of *LRRS* is explicitly defined.

The third point is to replace the linear ramified recursion scheme with parameter substitutions over binary trees by the following scheme:

$$\mathtt{f}([\mathbf{c}], \overline{u}, \overline{x}) \to \mathtt{g}_{\mathbf{c}}(\overline{u}, \overline{x}) \quad \mathbf{c} = \mathbf{0}, \mathbf{1}, \bot$$
$$\mathtt{f}([\mathbf{c}, \mathbf{b}, l], \overline{u}, \overline{x}) \to \mathtt{g}(\mathtt{f}(\mathbf{Left}([\mathbf{c}, \mathbf{b}, l]), \overline{u}, \mathtt{h}_1(\overline{x})), .., \mathtt{f}(\mathbf{Right}([\mathbf{c}, \mathbf{b}, l]), \overline{u}, \mathtt{h}_k(\overline{x})), \overline{x})$$

The program defined by the previous rules is explicitly fraternal, because $\mathtt{g}$, $\mathtt{g}_{\mathbf{c}}$, and $(\mathtt{h}_i)_i$ are flat functions.

The above Lemma entails the following one:

Lemma 9. *Every function ϕ in ALogTime is computable by an explicitly additive arboreal program $\boldsymbol{p}$.*

References

1. R. Amadio. Synthesis of max-plus quasi-interpretations. *Fundamenta Informaticae, 65(1–2)*, 2005.
2. R. Amadio, S. Coupet-Grimal, S. Dal-Zilio, and L. Jakubiec. A functional scenario for bytecode verification of resource bounds. In *CSL*, volume 3210 of *LNCS*, pages 265–279, 2004.
3. R. Amadio and S. Dal-Zilio. Resource control for synchronous cooperative threads. In *CONCUR*, volume 3170 of *LNCS*, pages 68–82, 2004.
4. T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *TCS*, 236:133–178, 2000.
5. M. Atjai. Σ^1_1-formulae on finite strutures. *Annals of Pure and Applied Logic*, 24:1–48, 1983.
6. D. Barrington, N. Immerman, and H. Straubing. On uniformity within nc. *J. of Computer System Science*, 41(3):274–306, 1990.
7. S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2:97–110, 1992.
8. S. Bloch. Function-algebraic characterizations of log and polylog parallel time. *Computational complexity*, 4(2):175–205, 1994.
9. G. Bonfante, R. Kahle, J.-Y. Marion, and I. Oitavem. Towards an implicit characterization of NCk. In *CSL'06*, LNCS, 2006.
10. G. Bonfante, J.-Y. Marion, and J.-Y. Moyen. Quasi-interpretation: a way to control ressources. *survey submitted, revision.* `http://www.loria/~marionjy`.
11. G. Bonfante, J.-Y. Marion, J.-Y. Moyen, and R. Péchoux. Synthesis of quasi-interpretations. *Workshop on Logic and Complexity in Computer Science, LCC2005, Chicago*, 2005. `http://www.loria/~pechoux`.
12. S. Buss. The boolean formula value problem is in ALOGTIME. In *STOC*, pages 123–131, 1987.
13. A. Chandra, D. Kozen, and L. Stockmeyer. Alternation. *Journal of the ACM*, 28:114–133, 1981.
14. P. Clote. Computational models and function algebras. In D. Leivant, editor, *LCC'94*, volume 960 of *LNCS*, pages 98–130, 1995.
15. A. Cobham. The intrinsic computational difficulty of functions. In *Conf. on Logic, Methodology, and Philosophy of Science*, pages 24–30. North-Holland, 1962.

16. K.J. Compton and C. Laflamme. An algebra and a logic for nc. *Inf. Comput.*, 87(1/2):240–262, 1990.
17. S.A. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64(1-3):2–21, 1985.
18. S. Dal-Zilio and R. Gascon. Resource bound certification for a tail-recursive virtual machine. In *APLAS 2005*, volume 3780 of *LNCS*, pages 247–263, 2005.
19. M. Furst, J. Saxe, and M. Spiser. Parity, circuits, and the polynomial time hierarchy. *Math. Systems Theory*, 17:13–27, 1984.
20. J.-Y. Girard. Light linear logic. In D. Leivant, editor, *LCC'94*, number 960 in LNCS, 1995.
21. M. Hofmann. Programming languages capturing complexity classes. *SIGACT News Logic Column 9*, 2000.
22. L. Kristiansen and N.D. Jones. The flow of data and the complexity of algorithms. In *New Computational Paradigms*, number 3526 in LNCS, pages 263–274, 2005.
23. C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *POPL*, volume 28, pages 81–92, 2001.
24. D. Leivant. Predicative recurrence and computational complexity I: Word recurrence and poly-time. In *Feasible Mathematics II*, pages 320–343. Birkhäuser, 1994.
25. D. Leivant. A characterization of NC by tree recurrence. In *39th Annual Symposium on Foundations of Computer Science, FOCS'98*, pages 716–724, 1998.
26. D. Leivant and J.-Y. Marion. Lambda Calculus Characterizations of Poly-Time. *Fundamenta Informaticae*, 19(1,2):167–184, September 1993.
27. D. Leivant and J.-Y. Marion. A characterization of alternating log time by ramified recurrence. *TCS*, 236(1-2):192–208, Apr 2000.
28. J.-Y. Marion and J.-Y. Moyen. Efficient first order functional program interpreter with time bound certifications. In *LPAR*, volume 1955 of *LNCS*, pages 25–42, 2000.
29. J.-Y. Marion and R. Péchoux. Resource analysis by sup-interpretation. In *FLOPS 2006*, volume 3945 of *LNCS*, pages 163–176, 2006.
30. J.-Y. Moyen. *Analyse de la complexité et transformation de programmes.* Thèse d'université, Nancy 2, Dec 2003.
31. K.-H. Niggl and H. Wunderlich. Certifying polynomial time and linear/polynomial space for imperative programs. *SIAM J. on Computing.* to appear.
32. W. Ruzzo. On uniform circuit complexity. *J. of Computer System Science*, 22(3):365–383, 1981.