

Parallel computable higher type functionals

(Extended Abstract)

Peter Clote* A. Ignjatović† B. Kapron‡

1 Introduction to higher type functionals

The primary aim of this paper is to introduce higher type analogues of some familiar parallel complexity classes, and to show that these higher type classes can be characterized in significantly different ways. Recursion-theoretic, proof-theoretic and machine-theoretic characterizations are given for various classes, providing evidence of their naturalness.

In this section, we motivate the approach of our work. In proof theory, primitive recursive functionals of higher type were introduced in Gödel’s *Dialectica* [13] paper, where they were used to “witness” the truth of arithmetic formulas. For instance, a function f witnesses the formula $\forall x \exists y \Phi(x, y)$, where Φ is quantifier-free, provided that

$$\forall x \Phi(x, f(x)),$$

while a type 2 functional F witnesses the formula $\forall x \exists y \forall u \exists v \Phi(x, y, u, v)$, provided that

$$\forall x \forall u \Phi(x, f(x), u, F(x, f(x), u)).$$

Gödel’s formal system \mathbf{T} is a variant of the finite typed λ -calculus, with the adjunction of a recursor R at all finite levels. It is known that the type 1 functionals of system \mathbf{T} are exactly the functions provably recursive in first order Peano arithmetic.

In the 1930’s and 40’s, some work in mathematical logic centered around what is known as Church’s thesis — Turing machine computable, μ -recursive, λ -calculus definable, Herbrand computable, etc. were all shown to be equivalent formulations of the notion

of “partial recursive”. In [18], S.C. Kleene investigated various notions of recursive functionals of higher type, comparing a Turing machine based notion with a schematic definitional notion, etc.

Recently, a series of papers on polynomial time computable higher type functionals has appeared (see [4, 10, 9]). From the standpoint of computational complexity, this work is of interest, because (1) functionals provide a natural generalization of relative computation (Turing machines with oracles), (2) λ -calculus is the prototype functional programming language; thus particular systems of λ -calculi corresponding to resource bounded computations may shed light on issues of efficiency for functional languages.

From a mathematical standpoint, such λ -calculi provide functionals for “feasibly realizing” certain bounded formulas of arithmetic, thus characterizing the “provably total” functions of a weak theory of arithmetic.

In [4], S. Buss gave the first definition of polynomial time computable functionals of all finite types by defining types with run times. His intent was to think of a type 2 functional, for instance, as a feasible function having arguments e, x , where e is a Gödel number of a type 1 functional and x an integer.

In [10], S. Cook and A. Urquhart gave a different definition of polynomial time computable functionals without specifying run times within the type descriptions. The Cook-Urquhart system PV^ω is quite elegant, having been fashioned from Cook’s free variable equational calculus PV and Gödel’s system \mathbf{T} essentially by restricting the recursor R to allow only bounded recursion on notation for type 1 functionals.

In [9] and [17], S. Cook and B. Kapron considered the system PV^ω in the context of Kleene’s work on higher type computability. To that end, in [9] they give two characterizations of the PV^ω definable functionals of finite type using higher type programming languages. On characterization uses “typed bounded for loop” programs, which compute exactly the higher type functionals which are closed PV^ω terms. They show as well that “typed while loop” programs, with

*Research supported in part by NSF CCR-9102896.

Mailing address: clote@bcuhs1.bc.edu, Department of Computer Science, Boston College, Chestnut Hill MA 02167.

†Mailing address: ignjat@1cl.cmu.edu, Department of Philosophy, Carnegie-Mellon University, Pittsburgh PA

‡Mailing address: bmkapron@csr.uvic.ca, Department of Computer Science, University of Victoria, Victoria, BC, CANADA, V8W 3P6

runtimes bounded by a basic feasible functional, compute basic feasible functionals. In [17] it is shown that for type 2 functionals, the polynomial time computable functionals can be characterized using an oracle Turing machine model, where running times are bounded in a natural way using *second-order polynomials*.

In this paper, we show that characterizations similar to those of [9] and [17] can be given for higher type classes corresponding to the parallel complexity classes AC^0 and NC . Our general approach is as follows. We begin with recursion-theoretic, machine independent characterizations of these classes, which were originally formulated by P. Clote in [7]. These characterizations admit a fairly straightforward generalization to higher types. Namely, for type 2 functionals we only need to introduce the application functional $Ap(f, x) = f(x)$ as an initial functional, and parameterize the closure schemes to allow function inputs. For higher types, we use typed λ -calculi. One drawback of recursion-theoretic characterizations is that they are lacking “computational” intuition. Thus, we also show that machine-theoretic characterizations are possible. For type 2 functionals, this involves the introduction of the oracle concurrent access random access machine (OCRAM). This is an extension of Immerman’s [15] CRAM model. We use second-order polynomials to bound the processors and running times in this model, and obtain type 2 analogues of the equivalence of machine-theoretic and recursion-theoretic characterizations for AC^0 and NC . For higher types, we show that the functionals definable in the typed λ -calculi that we define can be characterized as those functionals which are computable in certain higher type loop programming languages. Note that for higher types, the problem of giving a characterization in a general computational model with resource bounds, remains open.

Since *all* of the above cited papers dealing with polynomial time depend on A. Cobham’s machine independent characterization of the class \mathcal{FP} of polynomial time computable functions, we give some the background results of Cobham, Mehlhorn and Kapron, and then cite characterizations of parallel complexity classes due to P. Clote in [7]. These machine independent characterizations of parallel complexity classes play an important role, analogous to that of Cobham’s work for sequential computation, in the our development of parallel computable higher type functionals. We compare the type 1 and type 2 functionals thus obtained with those defined from familiar machine models provided with function oracles.

2 Function algebras

In this section, we survey some results concerning function complexity classes. First, some notation. For any complexity class C , let \mathcal{FC} denote the collection of functions f of polynomial growth,¹ whose *bit graph*

$$A_f = \{(x, i) : i\text{-th bit of } f(x) \text{ is } 1\}$$

belongs to C .

Beginning with the introduction of primitive recursive functions, many attempts have been made to characterize subclasses of recursive functions in a natural algebra, where functions are built up from certain *initial* functions by effective operations (composition and usually some sort of iteration, recursion, minimization operator, etc.). In 1963, A. Cobham [8], the first to isolate polynomial time as a complexity class,² characterized the polynomial time computable functions as the smallest class of functions closed under a variant of bounded primitive recursion, called “bounded recursion on notation”.

Definition 1 *The function f is defined by bounded recursion on notation (BRN) from g, h_0, h_1, b if*

$$\begin{aligned} f(0, \vec{x}) &= g(\vec{x}), \\ f(2n, \vec{x}) &= h_0(n, \vec{x}, f(n, \vec{x})) \text{ if } n \neq 0 \\ f(2n+1, \vec{x}) &= h_1(n, \vec{x}, f(n, \vec{x})), \end{aligned}$$

provided that $f(n, \vec{x}) < b(n, \vec{x})$ holds for all n, \vec{x} .

To state Cobham’s characterization, we need to introduce the following additional initial functions: $o(x) = 0$, $s_0(x) = 2x$, $s_1(x) = 2x + 1$, $|x| = \lceil \log_2(x + 1) \rceil$, $x \# y = 2^{|x| \cdot |y|}$.

Theorem 2 (A. Cobham [8])

The collection \mathcal{FP} of polynomial time computable functions is the closure of the initial functions $o, s_0, s_1, i_k^n, \#$ under composition and BRN.

The class NC is the collection of functions computable in polylogarithmic time with a polynomial number of processors on a parallel random access machine or PRAM (see Stockmeyer-Vishkin [21]). For $k \geq 0$, we define the subclasses AC^k of NC using a modification of the PRAM first introduced by Immerman [15]: AC^k is the class of functions computable by

¹For some polynomial p and for all x , it is the case that $|f(x)| \leq p(|x|)$.

²J. Edmonds [11] independently drew attention to polynomial time by giving an efficient maximum matching algorithm for graphs. In [19], K. Mehlhorn credits theorem 2 as independently due to Weihrauch.

a *concurrent random access machine* CRAM in parallel time $O(\log^k(n))$ with processor bound $n^{O(1)}$. The justification for our definition of the classes AC^k in terms of the CRAM rather than the PRAM or uniform boolean circuit model lies in the robustness of this approach for the low-level classes AC^0 and AC^1 . Indeed, the class AC^0 , as defined above, has been shown to be equal to a number of quite differently defined classes, such as first-order definable functions [15], the logtime hierarchy LH [2], and a function algebra A_0 independently developed [7], and has arguably the best claim to be the most natural definition of constant parallel time.

To characterize parallel classes, we need the additional initial function $bit(x, i)$ giving the coefficient of 2^i in the binary representation of x .

We now introduce some bounded recursion schemes. Let $x \frown y$ denote the number obtained by concatenating the binary notation of x with the binary notation of y .

Definition 3 *The function f is defined by concatenation recursion on notation (CRN) from g, h_0, h_1 if*

$$\begin{aligned} f(0, \vec{y}) &= g(\vec{y}) \\ f(2n, \vec{y}) &= f(x, \vec{y}) \frown bit(h_0(n, \vec{y}), 0), \text{ if } n \neq 0 \\ f(2n+1, \vec{y}) &= f(x, \vec{y}) \frown bit(h_1(n, \vec{y}), 0). \end{aligned}$$

Definition 4 *The function f is defined by weak bounded recursion on notation WBRN from g, h, b if*

$$\begin{aligned} F(0, \vec{x}) &= g(\vec{x}) \\ F(2n, \vec{x}) &= h_0(n, \vec{x}, F(n, \vec{x})), \text{ if } n \neq 0 \\ F(2n+1, \vec{x}) &= h_1(n, \vec{x}, F(n, \vec{x})) \\ f(n, \vec{x}) &= F(|n|, \vec{x}) \end{aligned}$$

provided that $F(n, \vec{x}) < b(n, \vec{x})$ holds for all n, \vec{x} .

Let

$$\text{INITIAL} = \{o, s_0, s_1, i_k^n, bit, \#, |x|\}.$$

Definition 5 *The algebra A_0 is the smallest class of functions containing INITIAL and closed under composition and CRN. The algebra A is the closure of INITIAL under composition, CRN and WBRN, while A_k is the collection of those functions in A allowing at most k nested applications of WBRN.*

We state the principal characterization of parallel complexity classes from [7].

Theorem 6 ([7]) $AC^0 = A_0$, $AC^k = A_k$, $NC = A$.

3 Type 2 functional algebras and oracle Turing machines

In [19] K. Mehlhorn extends Cobham's definition to type 2 functionals. Here, we follow [17, 23], which differs only cosmetically from Mehlhorn's original definition. Recall that a type 2 functional is a function which takes a list of functions and numbers as input, and returns a number as output.

Definition 7 (Townsend [23]) *F is defined from H, G_1, \dots, G_m by functional composition if for all \vec{f}, \vec{x} ,*

$$F(\vec{f}, \vec{x}) = H(\vec{f}, G_1(\vec{f}, \vec{x}), \dots, G_m(\vec{f}, \vec{x}), \vec{x}).$$

F is defined from G by expansion if for all $\vec{f}, \vec{g}, \vec{x}, \vec{y}$,

$$F(\vec{f}, \vec{g}, \vec{x}, \vec{y}) = G(\vec{f}, \vec{x}).$$

F is defined from G, G_1, \dots, G_m by functional substitution if for all \vec{f}, \vec{x} ,

$$\begin{aligned} F(\vec{f}, \vec{x}) &= \\ &H(\vec{f}, \lambda y. G_1(\vec{f}, \vec{x}, y), \dots, \lambda y. G_m(\vec{f}, \vec{x}, y), \vec{x}). \end{aligned}$$

F is defined from G, H, K by limited recursion on notation (LRN) if for all \vec{f}, \vec{x}, y ,

$$\begin{aligned} F(\vec{f}, \vec{x}, 0) &= G(\vec{f}, \vec{x}) \\ F(\vec{f}, \vec{x}, y) &= \\ &H(\vec{f}, \vec{x}, y, F(\vec{f}, \vec{x}, \lfloor \frac{y}{2} \rfloor)), \text{ if } y \neq 0. \end{aligned}$$

provided that $F(\vec{f}, \vec{x}, y) < K(\vec{f}, \vec{x}, y)$ holds for all \vec{f}, \vec{x}, y .

Following [23], we define the collection of type 2 basic feasible functionals BFF.³

Definition 8 ([23, 17]) *The class of basic feasible functionals, denoted BFF is the smallest class of functionals containing all type 1 polytime functions and the application functional Ap , defined by $Ap(f, x) = f(x)$, and which is closed under functional composition, expansion, functional substitution, and LRN.*

In [19] Mehlhorn introduces the Turing machine model with function oracle, charging unit-cost for a function oracle call, independent of the length of the function value returned. Mehlhorn's model has an oracle input tape and an oracle output tape, thus avoiding the situation where m successive iterates of a function $f(f(\dots f(x) \dots))$ might take m steps.

³Townsend [23] calls this class POLY. We have followed [17, 9] in designating this class as BFF.

In [17], B. Kapron and S.A. Cook consider runtimes of type 2 functionals which are polynomial in the lengths of the function and integer inputs. This requires the notion of *second-order polynomial*.

Definition 9 (Kapron, Cook [17]) *First-order variables are elements of the set $\{n_1, n_2, \dots\}$. Second-order variables are elements of the set $\{L_1, L_2, \dots\}$. Second-order polynomials are defined inductively by*

1. Any $c \in \mathbf{N}$ is a second-order polynomial.
2. First-order variables are second-order polynomials.
3. If P, Q are second-order polynomials and L is a second-order variable, then $P + Q$, $P \cdot Q$, and $L(P)$ are second-order polynomials.

The size of integer x is $|x|$, the length of the binary representation of x . We define the size of a type 1 functional as follows.

Definition 10 *Let f be a type 1 m -ary function. Then*

$$|f|(n_1, \dots, n_m) = \max_{|x_i| \leq n_i} |f(x_1, \dots, x_m)|$$

For an integer x , $\|x\|$ denotes $\lfloor \log(|x|) \rfloor$.

$$\|f\|(n_1, \dots, n_m) = \max_{|x_i| \leq n_i} \|f(x_1, \dots, x_m)\|$$

In [17], Kapron and Cook define the class **BPT** of type 2 *basic polytime functionals* as those which are computable on an oracle Turing machine in time which is bounded by a second-order polynomial $P(|f_1|, \dots, |f_m|, |x_1|, \dots, |x_k|)$ in the size of the function inputs f_1, \dots, f_m and integer inputs x_1, \dots, x_k . They also prove the following result.

Theorem 11 (Kapron, Cook [17]) *The basic feasible functionals **BFF** are exactly the basic polytime functionals **BPT**.*

The type 2 analogue of concatenation recursion on notation is given by the following.

Definition 12 *F is defined from G, H, K by concatenation recursion on notation (CRN) if for all \vec{f}, \vec{x}, y ,*

$$\begin{aligned} F(\vec{f}, \vec{x}, 0) &= G(\vec{f}, \vec{x}) \\ F(\vec{f}, \vec{x}, s_0(y)) &= F(\vec{f}, \vec{x}, y) \hat{\sim} \text{bit}(H(\vec{f}, \vec{x}, y), 0), \\ &\quad \text{if } y \neq 0 \\ F(\vec{f}, \vec{x}, s_1(y)) &= F(\vec{f}, \vec{x}, y) \hat{\sim} \text{bit}(K(\vec{f}, \vec{x}, y), 0). \end{aligned}$$

Definition 13 *The type 2 functional H is defined by weak bounded recursion on notation **WBRN** from G, H_0, H_1, K if*

$$\begin{aligned} F(\vec{f}, \vec{x}, 0) &= G(\vec{f}, \vec{x}) \\ F(\vec{f}, \vec{x}, s_0(y)) &= H_0(\vec{f}, \vec{x}, y, F(\vec{f}, \vec{x}, y)), \\ &\quad \text{if } y \neq 0 \\ F(\vec{f}, \vec{x}, s_1(y)) &= H_1(\vec{f}, \vec{x}, y, F(\vec{f}, \vec{x}, y)) \\ H(\vec{f}, \vec{x}, y) &= F(\vec{f}, \vec{x}, |y|) \end{aligned}$$

provided that $F(\vec{f}, \vec{x}, y) < K(\vec{f}, \vec{x}, y)$ holds for all \vec{f}, \vec{x}, y .

Definition 14 *The algebra \mathcal{A}_0 is the smallest class of functionals (of type 1 and 2) containing $o, s_0, s_1, i_k^n, \text{bit}, |x|, \#$, Ap and closed under functional composition, expansion, functional substitution and CRN. The algebra \mathcal{A} is the closure of $o, s_0, s_1, i_k^n, \text{bit}, |x|, \#$, Ap under functional composition, expansion, functional substitution CRN and **WBRN**, while \mathcal{A}_k is the collection of those functionals in \mathcal{A} allowing at most k nested applications of **WBRN**.*

4 Parallel machine model

The parallel random access machine **PRAM**, originating in the work of Goldschlager [14], Fortune-Wyllie [12], and Shiloach-Vishkin [20], is a very widely accepted model of parallel computation. In [21], Stockmeyer and Vishkin related the concurrent read, concurrent write **CRCW-PRAM** to unbounded fan-in boolean circuit families. In [15], Immerman introduced the concurrent random access machine **CRAM**, a slight modification of the **CRCW-PRAM** which admits a unit-cost **SHIFT** instruction. There and in [2], Immerman extended the Stockmeyer-Vishkin result to prove that languages computable in uniform **CRAM** time $O(\log^k(n))$ with processor bound $n^{O(1)}$ are exactly those in *first order* uniform unbounded fan-in boolean circuits of depth $O(\log^k(n))$ and size $n^{O(1)}$, for all $k \geq 0$.

In this paper, we study parallel complexity classes of higher type functionals. To define type 2 parallel computable functionals, we introduce the oracle concurrent random access machine **OCRAM**, which allows oracle calls to type 1 functions. A fundamental question in this model is how to simulate the application functional $Ap(f, x) = f(x)$, where the integer input x is (by convention) given in binary with each bit in a

different global register, and what to charge for a function oracle call on x which returns the function value $f(x)$. For 0,1 valued characteristic functions f , or more generally for polynomially bounded functions f , we could charge unit-cost for oracle calls. But what if f is the Ackermann function? The approach we adopt is to allow any processor in one step to retrieve the function value

$$f(x_i \cdots x_j) = f\left(\sum_{k=i}^j x_k \cdot 2^{j-k}\right)$$

where i, j are current values of local registers, and $i \leq j$, and the x_i 's are contained in special *oracle registers*.

An OGRAM has a sequence R_0, R_1, \dots of random access machines RAM's which operate in a synchronous fashion in parallel. Each R_i has its own local memory, an infinite collection of registers, each of which can hold an arbitrary non-negative integer. There is an infinite collection of global or common registers accessible to all RAM's, which are used for reading the input, processor intercommunication, and output. Global registers are denoted $M_0^g, M_1^g, M_2^g, \dots$, and local registers by M_0, M_1, M_2, \dots . At times, we may write $M_{i,j}$ to indicate the local register M_i of processor P_j . In the case that more than one RAM attempts to write in the same global memory location, the lowest numbered processor succeeds (priority resolution of write conflict). An input x is initially given in the global registers, the register M_i^g holding the i -th bit of x . All other registers are initially contain the the special symbol \$, which is different from 0, 1 and designates that the register is empty. Similarly at termination, the output y is specified in the global memory, the register M_i^g holding the i -th bit of y . At termination of a computation all other global registers contain the symbol \$.

For each k -ary function argument f , there are k infinite collections of *oracle registers*, the i -th collection labeled $M_0^{o,i}, M_1^{o,i}, M_2^{o,i}, \dots$, for $1 \leq i \leq k$. As with global memory, in the event of a write conflict the lowest numbered processor succeeds in writing to an oracle register. Let res (result), $op0$ (operand 0) and $op1$ (operand 1) be non-negative integers, as well as $op2, op3, \dots, op(2k)$. If any register occurring on the right side of an instruction contains '\$', then the register on the left side of the instruction will be assigned the value '\$' (undefined).

Instructions are as follows.

$$\begin{aligned} M_{res} &:= \text{constant} \\ M_{res} &:= \text{processor number} \\ M_{res} &:= M_{op1} \end{aligned}$$

$$\begin{aligned} M_{res} &:= M_{op1} + M_{op2} \\ M_{res} &:= M_{op1} \dot{-} M_{op2} \\ M_{res} &:= MSP(M_{op1}, M_{op2}) \\ M_{res} &:= LSP(M_{op1}, M_{op2}) \\ M_{res} &:= *M_{op1} \\ M_{res} &:= *M_{op1}^g \\ *M_{res} &:= M_{op1} \\ *M_{res}^g &:= M_{op1} \\ *M_{res}^o &:= 0 \\ *M_{res}^o &:= 1 \\ M_{res}^o &:= 0 \\ M_{res}^o &:= 1 \\ M_{res} &:= *M_{op1}^o \\ M_{res} &:= f([M_{op1} \cdots M_{op2}]_1, [M_{op3} \cdots M_{op4}]_2, \dots, \\ &\quad [M_{op(2k-1)} \cdots M_{op(2k)}]_k) \\ \text{GOTO label} & \\ \text{GOTO label IF } M_{op1} = M_{op2} & \\ \text{GOTO label IF } M_{op1} \leq M_{op2} & \\ \text{HALT} & \end{aligned}$$

Cutoff subtraction is defined by $x \dot{-} y = x - y$, provided that $x \geq y$, else 0. The shift operators MSP and LSP are defined as follows.

- $MSP(x, y) = \lfloor x/2^y \rfloor$, provided that $y < |x|$, otherwise '\$'.
- $LSP(x, y) = x - 2^y \cdot (\lfloor x/2^y \rfloor)$, provided that $y \leq |x|$, otherwise '\$'.

In [15] N. Immerman defined the CRAM to be essentially the PRAM of [21], but additionally allowing the SHIFT operator

$$M_{res} := \text{SHIFT}(M_{op1}, M_{op2})$$

where "SHIFT(x, y) causes the word x to be shifted y bits to the right". Here, one must assume the CRAM of [15] operates on positive and negative integers and that for $y \geq 0$, $\text{SHIFT}(x, -y) = x \cdot 2^y$, i.e. SHIFT by a negative value means *shift left*, provided that $y \leq |x|$.

Instructions with $*$ concern indirect addressing. For example, the instruction $M_{res} := *M_{op1}$ reads the current contents of local memory whose address is the value of M_{op1} into local memory M_{res} . The notation $[M_{op(2i-1)} \cdots M_{op(2i)}]_i$ denotes the integer whose binary notation is given in oracle registers $M_{M_{op(2i-1)}}^{o,i}$

through $M_{M_{op}(2i)}^{o,i}$. In other words,

$$[M_{op(2i-1)} \cdots M_{op(2i)}]_i = \sum_{m=M_{op}(2i-1)}^{M_{op}(2i)} M_m^{o,i} \cdot 2^{op(2i)-m}.$$

The instruction $*M_{res}^o := 0$ sets the contents of the oracle register whose address is given by the current contents of local memory M_{res} to 0. The instruction $M_{res}^o := *M_{op1}^o$ sets the contents of local memory M_{res} to be the current contents of the oracle register whose address is given by the current contents of local memory M_{op1} . With these instructions, it will be the case that oracle registers hold a 0 or 1 but no larger integer.

A program is a finite sequence of instructions. Each individual RAM of an oCRAM has the same program. Each instruction has unit-cost.

In characterizing AC^k in the non-oracle case, Stockmeyer and Vishkin [21] require a polynomial bound $p(n)$ on the number of active processors on inputs of length n . With the above definition of oCRAM one might hope to characterize the class of type 2 functionals computable in constant parallel time with a second-order polynomial number of processors as exactly the type 2 functionals in the algebra \mathcal{A}_0 . Unfortunately, this is not true (details omitted).

To rectify this situation, we first define $\mathcal{Q} = \mathcal{Q}(M, f, x, t)$ to be the values *queried* by the oCRAM M at time t or before, given function input f and integer input x . Define $f_{\mathcal{Q}}$ by

$$f_{\mathcal{Q}}(u) = \begin{cases} f(u) & \text{if } u \in \mathcal{Q} \\ 0 & \text{otherwise.} \end{cases}$$

We say that the oCRAM M has polynomially many processors, if there is a second order polynomial P , such that for all f, x, t at time t , given input f, x , at most processors 0 through $P(|f_{\mathcal{Q}}|, |x|)$ are active, where $\mathcal{Q} = \mathcal{Q}(M, f, x, t)$. The oCRAM M is said to run in time t , if at time t all processors 0 through $P(|f_{\mathcal{Q}}|, |x|)$ are halted.

The input to an oCRAM is stipulated by $X_0 X_1 X_2 \cdots X_n \$\$ \cdots$ where $X_i = x_{n-i}$ for $i \leq n$. This corresponds to the usual convention that the leftmost bit of the binary representation of $x = \sum_{i \leq n} x_i 2^i$ is the most significant bit. Multiple inputs are separated by a single symbol '\$' as in $x_n \cdots x_0 \$ y_m \cdots y_0 \$\$$.

If $F(f, \vec{x})$ abbreviates $F(f_1, \dots, f_m, x_1, \dots, x_n)$ and P is a second order polynomial, then $P(|\vec{f}|, |\vec{x}|)$ abbreviates $P(|f_1|, \dots, |f_m|, |x_1|, \dots, |x_n|)$.

Theorem 15 *If $F(\vec{f}, \vec{x})$ is a functional belonging to \mathcal{A}_0 then there exists a second order polynomial*

$P(|\vec{f}|, |\vec{x}|)$ such that $F(\vec{f}, \vec{x})$ is computable by an oCRAM in constant time using $P(|\vec{f}|, |\vec{x}|)$ many processors.

Proof. We must show that all initial functionals are computable by constant-time oCRAM's with a second order polynomial bound on the number of processors, and that the closure schemes preserve this computability. We will just consider the Ap functional. Complete details may be found in the full paper.

• oCRAM program for $Ap(f, x) = f(x)$.

```

1  M0 = 0
2  M1 = processor number
3  M2 = *M1g
4  if (M2 = $) then M0g = M1
5  M3 = M0g ÷ 1 % M3 = |x| ÷ 1
6  *M1g = $ % erase global memory
7  *M1o = M2 % in Pi, Mio = Xi
8  M4 = f([M0...M3])
9  M5 = BIT(M4, M1)
10 if (M5 = $) then M0g = M1
11 M5 = M0g % M5 = |f(x)|
12 M6 = M5 ÷ (M1 + 1)
13 M7 = BIT(M4, M6)
14 *M1o = $
15 *M1g = $
16 *M1g = M7
17 if (M1 ≥ M5) then M1g = $
    % erase trailing 0's
18 HALT % Now Xi = RBIT(f(x), i)

```

To obtain the converse, we must arithmetize the computation of an oCRAM within \mathcal{A}_0 . Details of the following result are given in the full paper.

Theorem 16 *If a functional $F(\vec{f}, \vec{x})$ is computable by an oCRAM M in constant time with $P(|\vec{f}|, |\vec{x}|)$ many processors, for a second order polynomial P , then $F \in \mathcal{A}_0$.*

Summarizing, we now have the desired result.

Theorem 17 *A functional $F(\vec{f}, \vec{x}) \in \mathcal{A}_0$ if and only if it is computable on an oCRAM in constant time with at most $P(|\vec{f}|, |\vec{x}|)$ many processors, for some second-order polynomial P .*

We can similarly characterize the class \mathcal{A} . To do so, we introduce some machinery from proof theory.

In [6] the first order theory TNC was defined and it was shown that the functions Σ_1^b definable (i.e. provably total with nondeterministic polytime recognizable graph) in TNC are exactly the functions of NC . Independently, and at the same time, B. Allen [1] introduced the first order theory D_2^1 and showed that the functions Σ_1^b definable in D_2^1 are also exactly the functions of NC . In [22] G. Takeuti defined the elegant first order theory R_2^1 and showed it to be equivalent (i.e. bi-interpretable) to both TNC and D_2^1 .

We define a second order theory \mathbf{R}_2^1 based on Takeuti's elegant formulation of R_2^1 . The theory \mathbf{R}_2^1 has a two-sorted language with

- numerical variables x_0, x_1, x_2, \dots ranging over non-negative integers,
- function variables $f_0^n, f_1^n, f_2^n, \dots$ of arity n , for each $n \geq 0$, ranging over functions from \mathbf{N}^n into \mathbf{N} ,
- constant 0,
- unary function symbols $s, \lfloor x/2 \rfloor, |x|, pr$
- binary function symbols $+, \cdot, \#, \div, MSP, Ap_n$ for each $n \geq 1$,
- binary relation symbol \leq .

Generally we use x, y, z, \dots for numerical variables, and f, g, h, \dots for function variables, without indicating the arity, which should be clear from the context. As well, we may often drop the arity symbol in Ap_n , writing for instance $Ap(f, x, y)$ instead of $Ap_2(f^2, x, y)$. For readers familiar with S. Buss' system S_2^1 [3], the language of \mathbf{R}_2^1 is that of S_2^1 together with pr, \div, MSP, Ap_n , for $n \geq 1$. Terms are built up inductively:

- $0, x_0, x_1, \dots$ are terms,
- $Ap_n(f_i^n, t_1, \dots, t_n)$ is a term for each variable f_i^n of arity n , provided t_1, \dots, t_n are terms,
- $Ap_1(s, t), Ap_1(\lfloor x/2 \rfloor, t), Ap_1(|x|, t), Ap_1(pr, t), Ap_2(+, t_1, t_2), Ap_2(\cdot, t_1, t_2), Ap_2(\#, t_1, t_2), Ap_2(\div, t_1, t_2), Ap_2(MSP, t_1, t_2)$ are terms, provided that t, t_1, t_2 are terms,
- $s(t), \lfloor t/2 \rfloor, |t|, pr(t), t_1 + t_2, t_1 \cdot t_2, t_1 \# t_2, t_1 \div t_2, MSP(t_1, t_2)$ are terms, provided that t, t_1, t_2 are terms.

Note that $f(x)$ is *not* a term, for function *variable* f , though we can express the equivalent by $Ap(f, x)$. Sharply bounded formulas, also called Δ_0^b , are formed from atomic formulas by closure under the boolean operations and *sharply bounded quantification*:

$$\exists x < |t(\vec{f}, \vec{y})| \phi(\vec{f}, \vec{y}, x)$$

and

$$\forall x < |t(\vec{f}, \vec{y})| \phi(\vec{f}, \vec{y}, x).$$

The class of Σ_1^b formulas is obtained from the class of Δ_0^b formulas by closure under \wedge, \vee (but not \neg), sharply bounded quantification and *bounded existential quantification*:

$$\exists x < t(\vec{f}, \vec{y}) \phi(\vec{f}, \vec{y}, x).$$

The axioms of \mathbf{R}_2^1 are given by a finite collection BASIC of open formulas consisting of the defining equations for the functions and simple properties relating the functions symbols $0, s, \lfloor x/2 \rfloor, |x|, pr, +, \cdot, \div, MSP, Ap_n$ and the ordering relation \leq amongst themselves, together with the axiom of bit extensionality and the scheme $\Sigma_1^b - LBIND$.

The BASIC axioms of \mathbf{R}_2^1 include those axioms of the BASIC collection of S. Buss [3], together with finitely many additional open axioms relating the new function symbols $pr, \div, MSP, Ap_1, Ap_2$.

The axiom of bit extensionality is

$$|a| = |b|, \forall i < |a| (bit(a, i) = bit(b, i)) \rightarrow a = b$$

where bit is defined by

$$\begin{aligned} Mod2(a) &= a \div 2 \cdot (\lfloor a/2 \rfloor) \\ bit(a, i) &= Mod2(MSP(a, i)) \end{aligned}$$

The scheme of $\Sigma_1^b - LBIND$ is

$$\frac{?, \phi(\lfloor a/2 \rfloor) \rightarrow \Delta, \phi(a)}{?, \phi(0) \rightarrow \Delta, \phi(|t|)}$$

where ϕ is Σ_1^b and a satisfies the eigenvariable condition, i.e. a does not appear in the lower sequent.

Definition 18 *The functional $F(\vec{f}, \vec{x})$ is Σ_1^b -definable in \mathbf{R}_2^1 if there are sequences of terms $\vec{t}_1, \dots, \vec{t}_n, \vec{t}_{n+1}$ and a Σ_1^b -formula ϕ_F such that*

$$\begin{aligned} 1. \mathbf{R}_2^1 \vdash \quad & \forall \vec{f} \forall \vec{x} \exists \vec{z}_1 < \vec{t}_1(\vec{f}, \vec{x}) \dots \exists \vec{z}_n < \\ & \vec{t}_n(\vec{f}, \vec{x}, \vec{z}_1, \dots, \vec{z}_{n-1}) \\ & \exists y < \vec{t}_{n+1}(\vec{f}, \vec{x}, \vec{z}_1, \dots, \vec{z}_n) \phi_F(\vec{f}, \vec{x}, y) \end{aligned}$$

2. The following is true in the standard model
- $$\begin{array}{l} \forall \vec{f}, \forall \vec{x} \exists z_1 < t_1(\vec{f}, \vec{x}) \dots \exists z_n < \\ t_n(\vec{f}, \vec{x}, z_1, \dots, z_{n-1}) < \\ [F(\vec{f}, \vec{x}) < \\ t_{n+1}(\vec{f}, \vec{x}, z_1, \dots, z_n) \wedge \phi_F(\vec{f}, \vec{x}, F(\vec{f}, \vec{x}))]. \end{array}$$

Theorem 19 *The functional $F(\vec{f}, \vec{x})$ belongs to \mathcal{A} if and only if $F(\vec{f}, \vec{x})$ is Σ_1^b -definable in \mathbf{R}_2^1 .*

Proof. (Sketch) The proof from left to right is by induction on complexity build-up of functionals in \mathcal{A} . Definition by WBRN is handled by Σ_1^b -LBIND. The direction from right to left proceeds as in the type 1 case [6], by induction on the number of inferences, after applying Gentzen's cut elimination theorem. Details are omitted. ■

It is straightforward to give a Σ_1^b -definition of an OCRAM computation of the functional $F(\vec{f}, \vec{x})$ running in time $\|G(\vec{f}, \vec{x})\|^{O(1)}$ with $O(\|G(\vec{f}, \vec{x})\|)$ processors, for $G \in \mathcal{A}$. Using Σ_1^b -LBIND, it follows that F is Σ_1^b -definable in \mathbf{R}_2^1 . By applying the previous theorem we conclude the following result.

Corollary 20 *The functional $F(\vec{f}, \vec{x})$ belongs to \mathcal{A} if and only if $F(\vec{f}, \vec{x})$ is computed on an OCRAM in time $\|G(\vec{f}, \vec{x})\|^{O(1)}$ with $O(\|G(\vec{f}, \vec{x})\|)$ processors, for some $G \in \mathcal{A}$.*

5 λ -calculi for parallel computable higher type functionals

We now consider parallel computable functionals of all finite types. In our notation and definitions, we try to follow Chapter 5 of [10]. The class of all *finite types* is given inductively as follows: 0 is a finite type, if σ and τ are finite types then $(\sigma \rightarrow \tau)$ is a finite type. Natural numbers are of type 0, number theoretic functions are of type 1, etc. An inductive argument shows that every finite type σ can be put in the unique form

$$(\sigma_1 \rightarrow (\sigma_2 \rightarrow (\dots \rightarrow (\sigma_n \rightarrow 0) \dots))).$$

In the following, we will omit parentheses when possible, with association understood to the right.

For a set of function symbols \mathcal{F} of arbitrary type, we can form the set $\Lambda(\mathcal{F})$ of λ -terms over \mathcal{F} in the standard way: we start with \mathcal{F} and an infinite set of variables of each type, and close under the operations of lambda abstraction and application. Recall that if S is a term of type τ and X is a variable of type σ ,

then the *lambda abstraction* $(\lambda X.S)$ is a term of type $(\sigma \rightarrow \tau)$, and if S is a term of type $(\sigma \rightarrow \tau)$, and T is a term of type σ , then the *application* (ST) is a term of type τ .

The terms of AV^ω are those of $\Lambda(\mathcal{F})$, where \mathcal{F} is $\{0, s_0, s_1, s, |x|, tr, parity, trunc, pad, \#, cond, R\}$.

The intended interpretation of the function symbols is given by: $s_0, s_1, tr, parity, trunc, pad, \#$ are as $s_0(x) = 2 \cdot x, s_1(x) = 2 \cdot x + 1, s(x) = x + 1, |x| = \lceil \log_2(x + 1) \rceil, tr(x) = \lfloor x/2 \rfloor, parity(x) = x \bmod 2, trunc(x, y) = \lfloor x/2^y \rfloor, pad(x, y) = 2^{|y|} \cdot x, \text{ and } x\#y = 2^{|x| \cdot |y|}$. The *conditional* function satisfies

$$cond(x, y, z) = \begin{cases} y & \text{if } x = 0 \\ z & \text{else} \end{cases}$$

and the recursor R satisfies

$$R(x, h, y) = \begin{cases} y & \text{if } x = 0 \\ s_{parity(h(x))}(R(\lfloor x/2 \rfloor, h, y)) & \text{else.} \end{cases}$$

To characterize *NC* computable functionals, we add a new term *sqrt* of type

$$0 \rightarrow 0$$

and a new term T of type

$$0 \rightarrow (0 \rightarrow 0 \rightarrow 0) \rightarrow (0 \rightarrow 0) \rightarrow 0 \rightarrow 0$$

to the above definition of AV^ω . This function is defined by $sqrt(x) = MSP(x, tr(\lfloor s_0(x) \rfloor))$.

Fact 21 $|sqrt(x)| = \lfloor \frac{|x|}{2} \rfloor$.

Fact 21 shows that $sqrt(x)$ has the same size as the integer part of the square root of x , hence the name. It follows that the smallest value k for which $sqrt$, iterated k times on x yields 0, must be $\lceil \|x\| \rceil$.

The interpretation of the recursor T is

$$\begin{aligned} T(0, Z, W, y) &= y \\ T(x, Z, W, y) &= \min\{W(x), Z(x, T(sqrt(x), Z, W, y))\} \\ &\quad \text{if } x > 0. \end{aligned}$$

This scheme clearly corresponds to weak bounded recursion

Using standard techniques of λ -calculus (strong normalization) together with the machine characterizations we had of type 2 functionals, the following immediately result.

Theorem 22 *For type 2 functional F ,*

- F belongs to AV^ω iff F is computable on an OCRAM in constant time with a polynomial number of processors.*

2. F belongs to NCV^ω iff F is computable on an OGRAM in time $\|G(\vec{f}, \vec{x})\|^{O(1)}$ with $O(\|G(\vec{f}, \vec{x})\|)$ processors, for some $G \in \mathcal{A}$.

Corollary 23

$$AV^\omega \subset NCV^\omega \subset PV^\omega$$

where all inclusions are proper.

Proof. Oracle separations are well known for the corresponding machine and circuit classes. ■

6 Higher type loop languages

In [9] it is shown that the basic feasible functionals of finite type can be characterized as those functionals computable by a certain class of loop programs known as *bounded typed loop programs* (BTLP's). A similar characterization is possible for the functions definable in AV^ω and NCV^ω . We must modify the approach of [9] in two ways. First of all, because the looping constructs which we allow are very weak, we require a richer repertoire of basic assignment instructions. Secondly, a more elaborate syntax is required for loop statements, in order to allow a finer calibration of computational power.

Before formulating our programming model, we will introduce the domain of computation over which we will work. The functionals of finite type were introduced by Kleene in [18]. We consider a class similar to Kleene's, which we call the *hereditarily total functionals of finite type*. These are defined as follows. $\mathbf{Fn}(\tau)$ denote the functionals of type τ . Then $\mathbf{Fn}(0) = \mathbf{N}$ and $\mathbf{Fn}(\tau_1 \rightarrow \dots \rightarrow \tau_k \rightarrow 0)$ consists of all total \mathbf{N} -valued functions with domain $\mathbf{Fn}(\tau_1) \times \dots \times \mathbf{Fn}(\tau_k)$. \mathbf{Fn} , the class of hereditarily total functionals of finite type, is just \bigcup_τ is a type $\mathbf{Fn}(\tau)$.

All of the loop instructions which we will consider will have the form

$$\mathbf{loop } y_i, y_j, W \mathbf{ begin } I \mathbf{ end} \tag{1}$$

where I is some list of instructions. The intuition behind such a loop instruction is that we iterate the body I of the loop y_j times. At the k th iteration $1 \leq k \leq x$, variable y_i takes on the value k . Any assignment taking place during the k th iteration of I must have a result bounded by $W(k)$ (note that W may be a constant function). In order to obtain characterizations of the classes in question, we will place restrictions on y_j, W and I .

We now give a general description of programs without looping constructs. We will call such programs *typed straight line programs* (TSLP's). A typed straight line program (or program, for short) consists of a declaration section and an instruction section. For each type τ , there are variables $X_0^\tau, X_1^\tau, \dots$.

The declaration section is required for the definition of subprograms, and consists of three declarations, which have the following form:

- Input declaration

$$\mathbf{input } Y_1^{\tau_1}, \dots, Y_k^{\tau_k} \tag{2}$$

- Output declaration

$$\mathbf{output } x \tag{3}$$

- Subprogram declarations

$$D_1, \dots, D_l$$

where $l \geq 0$.

Each subprogram declaration D_i has the form

$$\mathbf{subprogram } V^\sigma : Q \tag{4}$$

where Q is a program and V^σ is a variable of type σ called the *program name*.

The instruction section consists of a sequence of instructions. *Basic instructions* are of the form $x \leftarrow \mathbf{op}(y_1, \dots, y_k)$ where x and the y_i 's are type 0 variables, and \mathbf{op} is a type 1 function symbol of AV^ω . *Function calls* have the form $x \leftarrow X(Y_1, \dots, Y_k)$ where x has type 0, X has type $\tau_1 \rightarrow \dots \rightarrow \tau_k \rightarrow 0$ and X_i has type τ_i . Note that X and the Y_i 's may be either input variables or subprogram names.

More details on the structure and semantics of programs will be given on the full paper. We note that the approach here is very similar to that of [9], and refer the reader to that paper for details.

We now turn to loop programs. We define restrictions of the general loops given in equation 1. The first are called *concatenation loops* and have the form

$$\begin{aligned} &\mathbf{loop } y_i, |y_j|, |y_j| + |z| \mathbf{ begin} \\ &\quad z \leftarrow \mathbf{cond}(\mathbf{bit}(y_i, 1), \\ &\quad \quad \mathbf{cond}(\mathbf{bit}(h(0, y_i), s_0(z), s_1(z))), \\ &\quad \quad \mathbf{cond}(\mathbf{bit}(h(1, y_i), s_0(z), s_1(z)))) \\ &\mathbf{end}, \end{aligned}$$

and weak bounded loops have the form

$$\mathbf{loop } y_i, \|y_j\|, W \mathbf{ begin } I \mathbf{ end},$$

where W is the name of a subprogram of type $(0 \rightarrow 0)$. We modify TSLP's by adding concatenation loops to form *typed concatenation loop programs* (TCLP's); and by adding weakly bounded loops and a basic assignment $x \leftarrow \text{sqr}t(y)$ to form *weakly bounded typed loop programs* (WBTLP's).

We obtain the following result.

Theorem 24 *Over the hereditarily total functionals of finite type, those functionals definable in AV^ω and NC^ω correspond, respectively, to the functionals computable by TCLP's and WBTLP's.*

The proof of this theorem appears in the full paper. In proving the forward direction, we need to show that the looping constructs in our language are powerful to simulate the recursors in the corresponding calculi. The converse requires ability to define a next-step function for computations. This is done compositionally, using recursors in the case involving loops.

References

- [1] B. Allen. Arithmetizing uniform NC . *Annals of Pure and Applied Logic*, 53(1):1–50, 1991.
- [2] D. Mix Barrington, N. Immerman, and H. Straubing. On uniformity in NC^1 . *J. Comp. Syst. Sci.*, 41(3):274–306, 1990.
- [3] S. Buss. *Bounded Arithmetic*, volume 3 of *Studies in Proof Theory*. Bibliopolis, 1986. 221 pages.
- [4] S. Buss. The polynomial hierarchy and intuitionistic bounded arithmetic. In A.L. Selman, editor, *Structure in Complexity Theory*, volume 223, pages 77–103. 1986. Springer Lecture Notes in Computer Science.
- [5] P. Clote. $ALOGTIME$ and a conjecture of S.A. Cook. *Annals of Mathematics and Artificial Intelligence*, 6:57–106, 1992.
- [6] P. Clote and G. Takeuti. Bounded arithmetics for NC , $ALOGTIME$, L and NL . *Annals of Pure and Applied Logic*, 56:73–117, 1992.
- [7] P.G. Clote. Sequential, machine-independent characterizations of the parallel complexity classes $ALOGTIME$, AC^k , NC^k and NC . In P.J. Scott S.R. Buss, editor, *Feasible Mathematics*, pages 49–70. Birkhäuser, 1990.
- [8] A. Cobham. The intrinsic computational difficulty of functions. In Y. Bar-Hillel, editor, *Logic, Methodology and Philosophy of Science II*, pages 24–30. North-Holland, 1965.
- [9] S.A. Cook and B.M. Kapron. Characterizations of the feasible functionals of finite type. In P.J. Scott S.R. Buss, editor, *Feasible Mathematics*, pages 71–98. Birkhäuser, 1990.
- [10] S.A. Cook and A. Urquhart. Functional interpretations of feasibly constructive arithmetic. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing, Seattle*, 1989. pp. 107-112.
- [11] J. Edmonds. Paths, trees, flowers. *Canad. J. Math.*, 17:449–467, 1965.
- [12] S. Fortune and J. Wyllie. Parallelism in random access machines. In *10th Annual ACM Symposium on Theory of Computing*, 1978. pp. 114-118.
- [13] K. Gödel. Über eine bisher noch nicht benutzte Erweiterung des finiten Standpunktes. *Dialectica*, 12:280–287, 1958.
- [14] L. Goldschlager. A unified approach to models of synchronous parallel machines. *Journal of the Association of Computing Machinery*, 29(4):pp. 1073–1086, October 1982.
- [15] N. Immerman. Expressibility and parallel complexity. *SIAM J. Comput.*, 18(3):625–638, 1989.
- [16] B. Kapron. Feasible computation in higher types. Technical Report 249/91, University of Toronto, March 1991. 104 pages.
- [17] B. Kapron and S.A. Cook. A new characterization of Mehlhorn's poly time functionals. In *Proceedings of the 32nd Symposium on Foundations of Computer Science*, 1991. pp. 342-347.
- [18] S.C. Kleene. Recursive functionals and quantifiers of finite types i. *Transactions of the American Mathematical Society*, 91:1–52, 1959.
- [19] K. Mehlhorn. Polynomial and abstract subrecursive classes. *JCSS*, 12:147–178, 1976.
- [20] Y. Shiloach and U. Vishkin. Finding the maximum, merging and sorting in a parallel computation model. *Journal of Algorithms*, 3:57–67, 1982.
- [21] L. Stockmeyer and U. Vishkin. Simulation of parallel random access machines by circuits. *SIAM Journal on Computing*, 13:409–422, 1984.
- [22] G. Takeuti. $RSUV$ isomorphism. In P. Clote and J. Krajíček, editors, *Arithmetic, Proof Theory and Computational Complexity*, pages 364–386. Oxford University Press, 1993.
- [23] M. Townsend. Complexity for type-2 relations. *Notre Dame Journal of Formal Logic*, 31:241–262, 1990.