Classes of sub-recursive function
Selected bibliography

Armando B. Matos[1]

November 7, 2012

[1]Email: armandobcm@yahoo.com

This is a personal bibliography on

classes of sub-recursive functions.

The references in the abstracts have been omitted.

Bibliography

[AAV10] Philippe Andary, Bruno Patrou Andary, and Pierre Valarcher. A representation theorem for primitive recursive algorithms. *Fundamenta Informaticae*, XX:1–18, 2010. ABSTRACT. We formalize the algorithms computing primitive recursive (PR) functions as the abstract state machines (ASMs) whose running length is computable by a PR function. Then we show that there exists a programming language (implementing only PR functions) by which it is possible to implement any one of the previously defined algorithms for the PR functions in such a way that their complexity is preserved.

[Axt59] Paul Axt. On a subrecursive hierarchy and primitive recursive degrees. *Transactions of the American Mathematical Society*, 92:85–105, 1959. (From the Introduction) We shall investigate some problems connected with these classifications [of general and primitive recursive degrees]. First a uniqueness property of classes C_y associated with notations for the same ordinal is described and shown to hold at ordinals less than ω^2 and to fail at ω^2. The k-recursive functions of Peter are located in the hierarchy below the ω^ω level. Although it is not yet settled whether all recursive functions are obtained, it is clear that $\cup_{y\in O}C_y$ is a large and interesting class. Finally primitive recursive degrees are studied, and certain similarities to and differences from the theory of general recursive degrees of [Kleene and Post reference] are obtained.

[Axt63] Paul Axt. Iteration of primitive recursion. *Zeitschrift fur Mathematis-*

che Logik und Grundlagen der Mathematik, 11, 1963.

[BD95] Stephen Brookes and Denis Dancanet. Sequential algorithms, deterministic parallelism, and intensional expressiveness. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on POPL*, pages 13–24, 1995. ABSTRACT. We call a language L_1 *intensionally more expressive* than a language L_2 if there are functions that can be computed faster in L_1 than in L_2. We study the intensional expressiveness of several languages: the Berry-Curien programming language of sequential algorithms, CDS0, a deterministic parallel extension to it, named CDSP, and various parallel extensions to the functional programming language PCF. The paper consists of two parts. In the first part we show that CDS0 can compute the minimum of two numbers n and p in unary representation in time $O(\min(n, p))$ [...] In the second part, we show that deterministic parallelism adds intensional expressiveness, setting a "folk" conjecture from the literature in the negative. [...] We identify a hierarchy of intensional expressiveness for deterministic problems.

[BHN98] Stephen J. Bellantoni and Karl Heinz Niggl. Ranking primitive recursions: The low Grzegorczyk classes revisited. *SIAM Journal of Computing*, 29:401–415, 1998. ABSTRACT. Traditional results in subrecursion theory are integrated with the recent work in "predicative recursion" by defining a simple ranking ρ of all primitive recursive functions. The hierarchy defined by this ranking coincides with the Grzegorczyk hierarchy at and above the linear-space level. Thus, the result is like an extension of the Schwichtenberg/Müller theorems. When primitive recursion is replaced by recursion on notation, the same series of classes is obtained except with the polynomial time computable functions at the first level.

[CF98] Löic Colson and Daniel Fredholm. System T, call-by-value and the minimum problem. *Theoretical Computer Science*, 206:301–315, 1998. ABSTRACT. It is shown that for Gödel s system T, evaluated call-by-value,

if an algorithm computes a non-trivial binary function (where trivial means constant or projection plus constant), then the time-complexity is at least linear in one of the inputs. This is in contrast to the call-by-name case. As a corollary, it follows that there is no algorithm in this setting which computes the minimum function in time-complexity $O(\min)$.

[Col91] Löic Colson. About primitive recursive algorithms. *Theoretical Computer Science*, 83:57–69, 1991. ABSTRACT. In the past few years, there has been a growing interest in the application of proof-theoretical methods to the design of functional programming languages. One approach relies on representation theorems, which show that a large class of general recursive functions can be encoded in a language where general recursion is replaced by primitive recursion with functions, functionals, as parameters. These results are however purely extensional in nature: they state that a large class of mathematical functions is representable in a given system, but they say nothing about the efficiency of such a representation. Although the intensional aspect is of primary concern for computer science, very little seems to be known about this question. This paper is a beginning in the study of this problem. We take as a case study the following computational model: a primitive recursive function is seen as defining a rewriting system which is evaluated in call-by-name. In this setting, we give a non-trivial necessary condition for an algorithm to be representable. As an application, we can show that the function $\inf$ (which computes the minimum of two integers in unary representation) cannot be programmed in complexity $O(\inf(n, p))$. Our proof method uses some basic notions of denotational semantics.

[Coq92] Thierry Coquand. Une preuve directe du theoreme d'ultime obstination. In *Compte Rendus de l'Academie des Sciences, Serie I*, number 314, 1992. ABSTRACT. (not yet).

[Dav01] René David. On the asymptotic behaviour of primitive recursive algorithms. *Theoretical Computer Science*, 266(1-2):159–193, 2001. AB-

STRACT. This paper develops a new semantics (the trace of a computation) that is used to study intensional properties of primitive recursive algorithms. It gives a new proof of the "ultimate obstination theorem" of L. Colson and extends it to the case when mutual recursion is permitted. The ultimate obstination theorem fails when other data types (e.g. lists) are used. I definene another property (the *backtracking property*) of the same nature but which is weaker than the ultimate obstination. This property is proved for every primitive recursive algorithm using any kind of data types.

[Fre96] Daniel Fredholm. Computing minimum with primitive recursion over lists. *Theoretical Computer Science*, 163(3):269–276, 1996. ABSTRACT. We show that there is no primitive recursive algorithm over the natural numbers and lists of natural numbers that computes the minimum of two numbers in time $O(\min)$, in call-by-value evaluation order. This is in contrast to the call-by-name case.

[Grz53] A. Grzegorczyk. Some classes of recursive functions. *Rozprawy Matematyczne*, 4:1–45, 1953. Introduction. In this paper an increasing sequence $\mathcal{E}^0$, $\mathcal{E}^1$,... of classes of recursive functions is examined. Each class $\mathcal{E}^n$ is closed under the operations of substitution and under the operation of limited recursion. The initial functions are primitive recursive ones. Therefore $\mathcal{E}^n \subset \mathcal{R}$, where $\mathcal{R}$ is the class of primitive recursive functions. Strictly speaking $\mathcal{R} = \cup_n \mathcal{E}^n$. Hence in the definition of the class $\mathcal{R}$ the operation of recursion cannot be eliminated or exchanged into the operation of limited recursion. The classes $\mathcal{E}^0$ and $\mathcal{E}^3$ will be examined in particular. For each function $f \in \mathcal{E}^0$ there exists a number k_0 such that $f(n) < n + k_0$. However, each recursive enumerable set is enumerable by some function of the class $\mathcal{E}^0$. We start with the investigation of the class $\mathcal{E}^3$. It is the class of elementary computable functions of Kalmar.

[Mey65] A. R. Meyer. Depth of nesting and the Grzegorczyk hierarchy. *Notices of the American Mathematical Society*, 12:342, 1965. ABSTRACT.

Loop programs have the property that an upper bound on the running time of a program is determined by its structure. Each program consists only of assignment and iteration (loop) statements, but all the arithmetic functions commonly encountered in digital computation can be computed by Loop programs. A simple procedure for bounding the running time is shown to be best possible; some programs actually achieve this bound, and it is effectively undecidable whether a program runs faster than the bound. The complexity of functions can be measured by the loop structure of the programs which compute them. The functions computable by Loop programs are precisely the primitive recursive functions.

[Mol73] Robert Moll. *Complexity classes of recursive functions*. PhD in Mathematics, Massachusetts Institute of technology, 1973. Contents of Chapter 1, "A survey of work on subrecursive hierarchies and subrecursive degrees": (i) ω-hierarchies of primitive recursive functions. (ii) ω-hierarchies of elementary functions. (iii) Transfinite hierarchies. (iv) Subrecursive degrees.

[Mos03] Yiannis N. Moschovakis. On primitive recursive algorithms and the greatest common divisor function. *Theoretical Computer Science*, 301(3):1–30, 2003. ABSTRACT. We establish linear lower bounds for the complexity of non-trivial, primitive recursive algorithms from piecewise linear given functions. The main corollary is that logtime algorithms for the greatest common divisor from such givens (such as *Stein's*) cannot be matched in efficiency by primitive recursive algorithms from the same given functions. The question is left open for the Euclidean algorithm, which assumes the remainder function.

[MR67a] A. R. Meyer and D. M. Ritchie. The complexity of loop programs. *Proceedings of 22nd National Conference of the ACM*, pages 465–469, 1967. (From the Intruduction) Although Loop [a language described in this paper]programs cannot compute all the computable functions, they can

compute all the primitive recursive functions. The functions computable by Loop programs are, in fact, precisely the primitive recursive functions. Several of our results can be regarded as an attempt to make precise the notion that the complexity of a primitive recursive function is apparent from its definition or program. This property is one of the reasons that the primitive recursive functions are used throughout the theory of computability, for [...] knowing that a function is computable is not very useful unless one can tell how difficult the function is to compute. A bound on the running time of a Loop program provides a rough estimate of the degree of difficulty of the computation defined by the program. Loop programs are so powerful that our bounds on running time cannot be of practical value-for functions computable by Loop programs are almost wholly beyond the computational capacity of any real device. Nevertheless they provide a good illustration of the theoretical issues involved in estimating the running time of programs, and we believe that readers with a practical orientation may find some of the results provocative.

[MR67b] A. R. Meyer and D. M. Ritchie. Computational complexity and program structure. *IBM Research Report RC 1817*, 1967.

[Nig01] Karl-Heinz Niggl. *Control structures in programs and computational complexity.* Habilitationsschrift zur Erlangung des akademischen Grades Dr. rer. nat. habil, Fakultät fur Informatik und Automatisierung, Technischen Universitä Ilmenau, 2001. This thesis is concerned with analysing the impact of nesting (restricted) control structures in programs, such as primitive recursion or loop statements, on the running time or computational complexity. The method obtained gives insight as to why some nesting of control structures may cause a blow up in computational complexity, while others do not. The method is demonstrated for three types of programming languages. Programs of the first type are given as lambda terms over ground-type variables enriched with constants for primitive recursion or recursion on notation. A second is concerned with ordinary loop

pro- grams and stack programs, that is, loop programs with stacks over an arbitrary but fixed alphabet, supporting a suitable loop concept over stacks. Programs of the third type are given as terms in the simply typed lambda calculus enriched with constants for recursion on notation in all finite types. As for the first kind of programs, each program t is uniformly assigned a measure $\mu(t)$, being a natural number computable from the syntax of t. For the case of primitive recursion, it is shown that programs of μ-measure $n+1$ compute exactly the functions in Grzegorczyk level $n+2$. In particular, programs of μ-measure 1 compute exactly the functions in FLINSPACE, the class of functions computable in binary on a Turing machine in linear space. The same hierarchy of classes is obtained when primitive recursion is replaced with recursion on notation, except that programs of μ-measure 1 compute precisely the functions in FPTIME, the class of the functions computable on a Turing machine in time polynomial in the size of the input. Another form of measure μ is obtained for the second kind of programs. It is shown that stack programs of μ-measure$n$ compute exactly the functions computable by a Turing machine in time bounded by a function in Grzegorczyk level $n+2$. In particular, stack programs of μ-measure 0 compute precisely the FPTIME functions. Furthermore, loop programs of μ-measure n compute exactly the functions in Grzegorczyk level $n+2$. In particular, loop programs of μ-measure 0 compute precisely the FLINSPACE functions. As for the third kind of programs, building on the insight gained so far, it is shown how to restrict recursion on notation in all finite types so as to characterise polynomial-time computability. The restrictions are obtained by using a ramified type structure, and by adding linear concepts to the lambda calculus. This gives rise to a functional programming language RA supporting recursion on notation in all finite types. It is shown that RA programs compute exactly the FPTIME functions.

[Par68] Charles Parsons. Hierarchies of primitive recursive functions. *Zeitschrift f. math. Logik und Grundlagen*, D, 1968. ABSTRACT. In this paper we

shall introduce a hierarchy of classes of primitive recursive functions and compare it to Grzegorczyk hierarchy of classes $\mathcal{E}^n$, and with two hierarchies of classes, which we call $\mathcal{D}^n$, based directly on nesting of primitive recursion [...] and on our classes $\mathcal{L}^n$, which are based on a more complex measure of the complexity of primitive recursive functions. The main outcome of our discussion is that except near the beginning, all three hierarchies coincide: if $p \geq 2$, $\mathcal{L}^p = \mathcal{E}^{p+1}$, and Schwichtenberg has shown that if $p \geq 3$, $\mathcal{D}^p = \mathcal{E}^{p+1}$. In particular, $\mathcal{L}^2$ is the class of elementary functions. Thus our work yields a characterization of the elementary functions in terms of nesting of recursion. We strengthen a result of Rödding by showing that every function elementary in a given function Ψ can be obtained by explicit definition from a constant set of elementary functions and a single function $\overline{\Psi}$ elementary in Ψ.

[RK66] B. Rotman and G. T. Kneebone. *The Theory of Sets and Transfinite Numbers*. Elsevier, 1966.

[Rob47] Raphael Robinson. Primitive recursive functions. *Bull. Amer. Math. Soc.*, 53(10):925–942, 1947.

[Rob65] Joel Robbin. *Subrecursive Hierarchies*. PhD in Mathematics, Princeton University, 1965. ABSTRACT. The classification problem for recursive functions is the problem of assigning ordinals to recursive functions as a measure of their complexity. In this paper we consider three approaches to this problem: the ordinal recursion hierarchy, the extended Grzegorczyk hierarchy, and the Kleene subrecursive hierarchy. We obtain characterizations of the nested n-fold recursive functions in terms of each of these hierarchies. In the last section of the paper we show some of the problems that arise when we try to generalize these hierarchies. A characterization of the nested n-fold recursive functions in terms of computational complexity on a Turing machine is also given in the paper.

[Rob68] Julia Robinson. Recursive functions of one variable. *Proceedings of the American Mathematical Society*, 9:815–820, 1968. .

[Tsi70] D. Tsichritzis. The equivalence problem of simple programs. *Journal of the ACM*, 17(4):729–738, 1970. ABSTRACT. Many problems, some of them quite meaningful, have been proved to be recursively unsolvable for programs in general. The paper is directed towards a class of programs where many decision problems are solvable. The equivalence problem has been proved to be unsolvable for the class L_2 of Loop programs defining the class of elementary functions. A solution is given for the class L_1 defining the class of simple functions. Further, a set of other decision problems not directly connected with the equivalence problem is investigated. These problems are found again to be unsolvable for the class L_2; but, as before, a solution is given for the class L_1. It is concluded, therefore, that there is a barrier of unsolvability between the classes L_1 and L_2.

[vdD03] Lou van den Dries. Generating the greatest common divisor, and limitations of primitive recursive algorithms. *Foundations of Computational Mathematics*, 3(3):297–324, 2003. ABSTRACT. The greatest common divisor of two integers cannot be generated in a uniformly bounded number of steps from these integers using arithmetic operations. The proof uses an elementary model-theoretic construction that enables us to focus on "integers with transcendental ratio". This unboundedness result is part of the solution of a problem posed by Y. Moschovakis on limitations of primitive recursive algorithms for computing the greatest common divisor function.

[Wai94] Stanley S. Wainer. The hierarchy of terminating recursive programs over $\mathbb{N}$. In *LCC*, pages 281–299, 1994. ABSTRACT. A terminating recursive program defines a total recursive functional, taking "given" functions to the function defined from them by the program. Termination means that the program has a well-founded computation tree with a recursive ordinal as it height, and Kleene noted that, in contrast with the well known "col-

lapsing phenomenon" for hierarchies of recursive functions, the resulting hierarchy expands right the way through ω_1^{CK}: i.e. for each recursive ordinal α there is a total recursive functional which can not be defined by any program of height less than α. In this paper we examine some of the ways in which the ordinal height of a program encodes its complexity. By a careful assignment of (proof theoretic) ordinal bounds to derivations in Kleene's equation calculus, the standard "fast", "medium", and "slow" growing hierarchies emerge as canonical complexity measures allowing different forms of recursion to be classified and compared. Known relationships between these hierarchies then yield measures of "transformational complexity" (e.g. recursive to tail recursive) in terms of their corresponding ordinal trade-offs. The underlying theme is that of Cut Elimination, but in an equational setting.