The Strength of Non-size-increasing Computation (Introduction and Summary)

Martin Hofmann

TU Darmstadt, FB 4, Schloßgartenstr. 7, 64289 Darmstadt, Germany
mh@mathematik.tu-darmstadt.de

Abstract. We study the expressive power non-size increasing recursive definitions over lists. This notion of computation is such that the size of all intermediate results will automatically be bounded by the size of the input so that the interpretation in a finite model is sound with respect to the standard semantics. Many well-known algorithms with this property such as the usual sorting algorithms are definable in the system in the natural way. The main result is that a characteristic function is definable if and only if it is computable in time $O(2^{p(n)})$ for some polynomial p. The method used to establish the lower bound on the expressive power also shows that the complexity becomes polynomial time if we allow primitive recursion only. This settles an open question posed in [1,6]. The key tool for establishing upper bounds on the complexity of derivable functions is an interpretation in a finite relational model whose correctness with respect to the standard interpretation is shown using a semantic technique.

Keywords: computational complexity, higher-order functions, finite model, semantics
AMS Classification: 03D15, 03C13, 68Q15, 68Q55

The present document contains an introduction to and a summary of the results presented in the author's invited talk at MFCS'01. A full version of the paper is available as [8]. The author wishes to thank the organisers and the programme committee of MFCS'01 for the invitation to present his results there. He would also like to acknowledge financial support by the Mittag-Leffler institute which funded a research stay at that place during which the bulk of the research presented here was carried out.

Consider the following recursive definition of a function on lists:

$$\begin{aligned} \mathtt{twice}(\mathsf{nil}) &= \mathsf{nil} \\ \mathtt{twice}(\mathsf{cons}(x,l)) &= \mathsf{cons}(\mathsf{tt},\mathsf{cons}(\mathsf{tt},\mathtt{twice}(l))) \end{aligned} \tag{1}$$

Here nil denotes the empty list, $\mathsf{cons}(x,l)$ denotes the list with first element x and remaining elements l. tt,ff are the members of a type T of truth values. We have that $\mathtt{twice}(l)$ is a list of length $2\cdot|l|$ where $|l|$ is the length of l. Now consider

$$\begin{aligned} \mathtt{exp}(\mathsf{nil}) &= \mathsf{cons}(\mathsf{tt},\mathsf{nil}) \\ \mathtt{exp}(\mathsf{cons}(x,l)) &= \mathtt{twice}(\mathtt{exp}(l)) \end{aligned} \tag{2}$$

We have $|\mathtt{exp}(l)| = 2^{|l|}$ and further iteration leads to elementary growth rates.

This shows that innocuous looking recursive definitions can lead to enormous growth. In order to prevent this from happening it has been suggested in [?,10] to rule out definitions like (2) above, where a recursively defined function, here `twice`, is applied to the result of a recursive call. Indeed, it has been shown that such discipline restricts the definable functions to the polynomial-time computable ones and moreover every polynomial-time computable *function* admits a definition in this style.

Many naturally occurring *algorithms*, however, do not fit this scheme. Consider, for instance, the definition of insertion sort:

$$\begin{array}{l}\mathtt{insert}(x,\mathsf{nil}) = \mathsf{cons}(x,\mathsf{nil})\\ \mathtt{insert}(x,\mathsf{cons}(y,l)) = \mathsf{if}\ x \le y\ \mathsf{then}\ \mathsf{cons}(x,\mathsf{cons}(y,l))\ \mathsf{else}\ \mathsf{cons}(y,\mathtt{insert}(x,l))\\ \mathtt{sort}(\mathsf{nil}) = \mathsf{nil}\\ \mathtt{sort}(\mathsf{cons}(x,l)) = \mathtt{insert}(x,\mathtt{sort}(l))\end{array} \tag{3}$$

Here just as in (2) above we apply a recursively defined function (`insert`) to the result of a recursive call (`sort`), yet no exponential growth arises.

It has been argued in [3] and [6] that the culprit is definition (1) because it defines a function that increases the size of its argument and that non size-increasing functions can be arbitrarily iterated without leading to exponential growth.

In [3] a number of partly semantic criteria were offered which allow one to recognise when a function definition is non size-increasing. In [6] we have given syntactic criteria based on linearity (bound variables are used at most once) and a so-called resource type $\diamond$ which counts constructor symbols such as "cons" on the left hand side of an equation.

This means that cons becomes a ternary function taking one argument of type $\diamond$, one argument of some type A (the head) and a third argument of type $\mathsf{L}(A)$, the tail. There being no closed terms of type $\diamond$ the only way to apply cons is within a recursive definition; for instance, we can write

$$\begin{array}{l}\mathsf{append}(\mathsf{nil},l_2) = l_2\\ \mathsf{append}(\mathsf{cons}(d,a,l_1),l_2) = \mathsf{cons}(d,a,\mathsf{append}(l_1,l_2))\end{array} \tag{4}$$

Alternatively, we may write

$$\mathsf{append}(l_1,l_2) = \mathsf{match}\ l\ \mathsf{with}\ \mathsf{nil}{\Rightarrow}l_2 \mid \mathsf{cons}(d,a,l_1'){\Rightarrow}\mathsf{cons}(d,\mathsf{append}(l_1,l_2) \tag{5}$$

We notice that the following attempted definition of `twice` is illegal as it violates linearity (the bound variable d is used twice):

$$\begin{array}{l}\mathtt{twice}(\mathsf{nil}) = \mathsf{nil}\\ \mathtt{twice}(\mathsf{cons}(d,x,l)) = \mathsf{cons}(d,\mathsf{tt},\mathsf{cons}(d,\mathsf{tt},\mathtt{twice}(l)))\end{array} \tag{6}$$

The definition of `insert`, on the other hand, is in harmony with linearity provided that `insert` gets an extra argument of type $\diamond$ and, moreover, we assume that the inequality test returns its arguments for subsequent use.

The main result of [6] and [1] was that all functions thus definable by *structural recursion* are polynomial-time computable even when higher-order functions are allowed. In [7] it has been shown that general-recursive first-order definitions admit a translation into a fragment of the programming language C without dynamic memory allocation ("malloc") which on the one hand allows one to automatically construct imperative implementations of algorithms on lists which do not require extra space or garbage collection. More precisely, this translation maps the resource type $\diamond$ to the C-type `void *` of pointers. The `cons` function is translated into the C-function which extends a list by a given value using a provided piece of memory. It is proved that the pointers arising as denotation of terms of type $\diamond$ always point to free memory space which can thus be safely overwritten.

This translation also demonstrates that all definable functions are computable on a Turing machine with linearly bounded work tape and an unbounded stack (to accommodate general recursion) which by a result of Cook[1] [4] equals the complexity class $DTIME(2^{O(n)})$. It was also shown in [7] that any such function admits a representation.

In the presence of higher-order functions the translation into C breaks down as C does not have higher-order functions. Of course, higher-order functions can be simulated as closures, but this then requires arbitrary amounts of space as closures can grow proportionally to the runtime. In a system based on structural recursion such as [6] this is not a problem as the runtime is polynomially bounded there. The hitherto open question of complexity of general recursion with higher-order functions is settled in this work [8] and shown to require a polynomial amount of space only in spite of the unbounded runtime.

We thus demonstrate that a function is representable with general recursion and higher-order functions iff it is computable in polynomial space and an unbounded stack or equivalently (by Cook's result) in time $O(2^{p(n)})$ for some polynomial p. The lower bound of this result also demonstrates that indeed all characteristic functions of problems in P are definable in the structural recursive system. This settles a question left open in [1,6].

In view of the results presented in the talk (see also [8]), these systems of non size-increasing computation thus provide a very natural connection between complexity theory and functional programming. There is also a connection to finite model theory in that programs admit a sound interpretation in a finite model. This improves upon earlier combinations of finite model theory with functional programming [5] where interpretation in a finite model was achieved in a brute-force way by changing the meaning of constructor symbols, e.g. successor of the largest number N was defined to be N itself. In those systems it is the responsibility of the programmer to account for the possibility of cut-off when reasoning about the correctness of programs. In the systems studied here linearity and the presence of the resource types automatically ensure that cutoff

[1] This result asserts that if $L(n) > \log(n)$ then $DTIME(2^{O(L(n))})$ equals the class of functions computable by a Turing machine with an $L(n)$-bounded R/W-tape and an unbounded stack.

never takes place. Formally, it is shown that the standard semantics in an infinite model agrees with the interpretation in a certain finite model for all well-formed programs.

Another piece of related work is Jones' [9] where the expressive power of cons-free higher-order programs is studied. It is shown there that first-order cons-free programs define polynomial time , whereas second-order programs define EXPTIME. This shows that the presence of "cons", tamed by linearity and the resource type changes the complexity-theoretic strength. While loc. cit. also involves Cook's abovementioned result (indeed, this result was brought to the author's attention by Neil Jones) the other parts of the proof are quite different.

References

1. Klaus Aehlig and Helmut Schwichtenberg. A syntactical analysis of non-size-increasing polynomial time computation. In *Proceedings of the Fifteenth IEEE Symposium on Logic in Computer Science (LICS '00), Santa Barbara*, 2000. 58, 60
2. Stephen Bellantoni and Stephen Cook. New recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2:97–110, 1992.
3. Vuokko-Helena Caseiro. *Equations for Defining Poly-time Functions*. PhD thesis, University of Oslo, 1997. Available by ftp from `ftp.ifi.uio.no/pub/vuokko/0adm.ps`. 59
4. Stephen A. Cook. Linear-time simulation of deterministic two-way pushdown automata. *Information Processing*, 71:75–80, 1972. 60
5. Andreas Goerdt. Characterizing complexity classes by higher type primitive recursive definitions. *Theoretical Computer Science*, 100:45–66, 1992. 60
6. Martin Hofmann. Linear types and non size-increasing polynomial time computation. To appear in Information and Computation. See `www.dcs.ed.ac.uk/home/papers/icc.ps.gz` for a draft. An extended abstract has appeared under the same title in Proc. Symp. Logic in Comp. Sci. (LICS) 1999, Trento, 2000. 58, 59, 60
7. Martin Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, 7(4), 2000. An extended abstract has appeared in *Programming Languages and Systems*, G. Smolka, ed., Springer LNCS, 2000. 60
8. Martin Hofmann. The strength of non size-increasing computation. 2001. Presented at the workshop *Implicit Computational Complexity 2001*, Aarhus, 20 May 2001. Submitted for publication. See also `www.dcs.ed.ac.uk/home/mxh/icc01_hofmann.ps`. 58, 60
9. Neil Jones. The Expressive Power of Higher-Order Types or, Life without CONS. *Journal of Functional Programming*, 2001. to appear. 61
10. Daniel Leivant. Stratified Functional Programs and Computational Complexity. In *Proc. 20th IEEE Symp. on Principles of Programming Languages*, 1993. 59