Theoretical Computer Science 585 (2015) 41–54

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Characterizing polynomial time complexity of stream programs using interpretations

Hugo Férée [a,c], Emmanuel Hainry [a,c], Mathieu Hoyrup [b,c], Romain Péchoux [a,c]

[a] Université de Lorraine, Nancy, France
[b] Inria Nancy – Grand Est, Villers-lès-Nancy, France
[c] Project-team CARTE, LORIA, UMR7503, France

ARTICLE INFO

Article history:
Available online 6 March 2015

Keywords:
Stream programs
Type-2 functionals
Interpretations
Polynomial time
Basic feasible functionals
Computable analysis
Rewriting

ABSTRACT

This paper provides a criterion based on interpretation methods on term rewrite systems in order to characterize the polynomial time complexity of second order functionals. For that purpose it introduces a first order functional stream language that allows the programmer to implement second order functionals. This characterization is extended through the use of exp-poly interpretations as an attempt to capture the class of Basic Feasible Functionals (BFF). Moreover, these results are adapted to provide a new characterization of polynomial time complexity in computable analysis. These characterizations give a new insight on the relations between the complexity of functional stream programs and the classes of functions computed by Oracle Turing Machine, where oracles are treated as inputs.

1. Introduction

Lazy functional languages like Haskell allow the programmer to deal with co-inductive datatypes in such a way that co-inductive objects can be evaluated by finitary means. Consequently, computations over streams, that is infinite lists, can be performed in such languages.

A natural question arising is the complexity of the programs computing on streams. Intuitively, the complexity of a stream program is the number of reduction steps needed to output the n first elements of a stream, for any n. However the main issue is to relate the complexity bound to the input structure. Since a stream can be easily identified with a function, a good way for solving such an issue is to consider computational and complexity models dealing with functions as inputs.

In this perspective, we want to take advantage of the complexity results obtained on type-2 functions (functions over functions), and in particular on BFF [1,2], to understand stream program complexity. For that purpose, we set Unary Oracle Turing Machine (UOTM), machines computing functions with oracles taking unary inputs, as our main computational model. This model is well-suited in our framework since it manipulates functions as objects with a well-defined notion of complexity. UOTM is a refinement of Oracle Turing Machine (OTM) on binary words which corresponds exactly to the BFF algebra in [3] under polynomial restrictions. UOTM is better suited than OTM to study stream complexity with a realistic complexity measure, since in the UOTM model accessing the nth element costs n transitions whereas it costs $\log(n)$ in the OTM model.

The Implicit Computational Complexity (ICC) community has proposed characterizations of OTM complexity classes using function algebra [3,4] and type systems [5,6] or recently as a logic [7].

E-mail address: hugo.feree@inria.fr (H. Férée).

http://dx.doi.org/10.1016/j.tcs.2015.03.008
0304-3975/

These latter characterizations are inspired by former characterizations of type-1 polynomial time complexity based on ramification [8,9]. This line of research has led to new developments of other ICC tools and in particular to the use of (polynomial) interpretations in order to characterize the classes of functions computable in polynomial time or space [10,11].

Polynomial interpretations [12,13] are a well-known tool used to show the termination of first order term rewrite systems. This tool has been adapted into variants, like quasi-interpretations and sup-interpretation [14], that allow the programmer to analyze program complexity. In general, interpretations are restricted to inductive data types and [15] was the first attempt to adapt such a tool to co-inductive data types including stream programs. In this paper, we introduce a second order variation of this interpretation methodology in order to constrain the complexity of stream program computation and we obtain a characterization of UOTM polynomial time computable functions. Using this characterization, we can analyze functions of this class in an easier way based on the premise that it is practically easier to write a first order functional program on streams than the corresponding Unary Oracle Turing Machine. The drawback is that the tool suffers from the same problems as polynomial interpretation: the difficulty to automatically synthesize the interpretation of a given program (see [16]). As a proof of versatility of this tool, we provide a partial characterization of the BFF class (the full characterization remaining open), just by changing the interpretation codomain: for that purpose, we use restricted exponentials instead of polynomials in the interpretation of a stream argument.

A direct and important application is that second order polynomial interpretations on stream programs can be adapted to characterize the complexity of functions computing over reals defined in Computable Analysis [17]. This approach is the first attempt to study the complexity of such functions through static analysis methods.

This paper is an extended version of [18] with complete proofs, additional examples and corrections.

1.1. Outline of the paper

In Section 2, we introduce (Unary) Oracle Turing Machines and their complexity. In Section 3, we introduce the studied first order stream language. In Section 4, we define the interpretation tools extended to second order and we provide a criterion on stream programs. We show our main characterization relying on the criterion in Section 5. Section 6 develops a new application, which was only mentioned in [18], to functions computing over reals.

2. Polynomial time oracle turing machines

In this section, we will define a machine model and a notion of complexity relevant for stream computations. This model (UOTM) is adapted from the Oracle Turing Machine model used by Kapron and Cook in their characterization of Basic Feasible functionals (BFF) [3]. In the following, $|x|$ will denote the size of the binary encoding of $x \in \mathbb{N}$, namely $\lceil \log_2(x) \rceil$.

Definition 1 *(Oracle Turing Machine).* An Oracle Turing Machine (denoted by OTM) $\mathcal{M}$ with k oracles and l input tapes is a Turing machine with, for each oracle, a state, one query tape and one answer tape.

Whenever $\mathcal{M}$ is used with input oracles $F_1, \dots F_k : \mathbb{N} \longrightarrow \mathbb{N}$ and arrives on the oracle state $i \in \{1, \dots, k\}$ and if the corresponding query tape contains the binary encoding of a number x, then the binary encoding of $F_i(x)$ is written on the corresponding answer tape. It behaves like a standard Turing machine on the other states.

We now introduce the unary variant of this model, which is more related to stream computations as accessing the n-th element takes at least n steps (whereas it takes $\log(n)$ steps in OTM. See Example 2 for details).

Definition 2 *(Unary Oracle Turing Machine).* A Unary Oracle Turing Machine (denoted UOTM) is an OTM where numbers are written using unary notation on the query tape, *i.e.* on the oracle state i, if w is the content of the corresponding query tape, then $F_i(|w|)$ is written on the corresponding answer tape.

Definition 3 *(Running time).* In both cases (OTM and UOTM), we define the cost of a transition as the size of the answer of the oracle, in the case of a query, and 1 otherwise.

In order to introduce a notion of complexity, we have to define the size of the inputs of our machines.

Definition 4 *(Size of a function).* The size $|F| : \mathbb{N} \longrightarrow \mathbb{N}$ of a function $F : \mathbb{N} \longrightarrow \mathbb{N}$ is defined by:

$$|F|(n) = \max_{k \leq n} |F(k)|$$

Remark 1. This definition is different from the one used in [3] (denoted here by $||.||$). Indeed, the size of a function was defined by $||F||(n) = \max_{|k| \leq n} |F(k)|$, in other words, $||f||(n) = |f|(2^n - 1)$. The reason for this variation is that in an UOTM, the oracle is closer to an infinite sequence (or stream as we will see in the following) than to a function, since it can access easily its first n elements but not its (2^n)th element which is the case in an OTM. In particular, this makes the size function computable in polynomial time (with respect to the following definitions).