Information and Computation 248 (2016) 56–81

Contents lists available at ScienceDirect

Information and Computation

www.elsevier.com/locate/yinco

Higher-order interpretations and program complexity ☆

Patrick Baillot [a], Ugo Dal Lago [b,c,*]

[a] Laboratoire d'Informatique du Parallélisme, Université de Lyon, CNRS, Ecole Normale Supérieure de Lyon, INRIA, Université Claude Bernard Lyon 1, France
[b] Università di Bologna, Italy
[c] INRIA Sophia Antipolis, France

ARTICLE INFO

Article history:
Received 9 December 2013
Available online 4 January 2016

Keywords:
Implicit computational complexity
Term rewriting systems
Type systems
Lambda-calculus

ABSTRACT

Polynomial interpretations and their generalizations like quasi-interpretations have been used in the setting of first-order functional languages to design criteria ensuring statically some complexity bounds on programs [10]. This fits in the area of implicit computational complexity, which aims at giving machine-free characterizations of complexity classes. In this paper, we extend this approach to the higher-order setting. For that we consider simply-typed term rewriting systems [35], we define higher-order polynomial interpretations for them, and we give a criterion ensuring that a program can be executed in polynomial time. In order to obtain a criterion flexible enough to validate interesting programs using higher-order primitives, we introduce a notion of polynomial quasi-interpretations, coupled with a simple termination criterion based on linear types and path-like orders.

1. Introduction

The problem of statically analyzing the performance of programs can be attacked in many different ways. One of them consists in verifying *complexity* properties early in the development cycle, when programs are still expressed in high-level languages, like functional or object-oriented idioms. And in this scenario, results from an area known as *implicit* computational complexity (ICC in the following) can be useful: often, they consist in characterizations of complexity classes in terms of paradigmatic programming languages (recursion schemes [30,8], λ-calculus [31], term rewriting systems [10], etc.) or logical systems (proof-nets, natural deduction, etc.), from which static analysis methodologies can be distilled. Examples are type systems, path-orderings and variations on the interpretation method. The challenge here is defining ICC systems which are not only simple, but also *intensionally* powerful: many natural programs among those with bounded complexity should be recognized as such by the ICC system, i.e., should actually be programs *of* the system.

One of the most fertile direction in ICC is indeed the one in which programs are term rewriting systems (TRS in the following) [10,11], whose complexity can be kept under control by way of variations of the powerful techniques developed to check termination of TRSs, namely path orderings [20], dependency pairs [33] and the interpretation method [29]. Many different complexity classes have been characterized this way, from polynomial time to polynomial space, to exponential time to logarithmic space. And remarkably, many of the introduced characterizations are intensionally powerful, in particular

☆ This work has been partially supported by ANR Project ELICA ANR-14-CE25-0005 and by the LABEX MILYON (ANR-10-LABX-0070) of Université de Lyon, within the program "Investissements d'Avenir" (ANR-11-IDEX-0007) operated by the French National Research Agency (ANR).

* Corresponding author.
E-mail addresses: patrick.baillot@ens-lyon.fr (P. Baillot), dallago@cs.unibo.it (U. Dal Lago).

http://dx.doi.org/10.1016/j.ic.2015.12.008
0890-5401/

when the interpretation method is relaxed and coupled with recursive path orderings, like in quasi-interpretations [11]. These techniques can be fruitfully combined into concrete tools (see, e.g., [4]).

The cited results indeed represent the state-of-the art in complexity analysis for *first-order* functional programs, i.e. when functions are *not* first-class citizens. If the class of programs of interest includes higher-order functional programs, the techniques above can only be applied when programs are either defunctionalized or somehow put in first-order form, for example by applying a translation scheme due to the second author and Simone Martini [19]. However, it seems difficult to ensure in that case that the target first-order programs satisfy termination criteria such as those used in [11], although some promising results in this direction have been recently obtained [3]. The article [13] proposed to get around this problem by considering a notion of *hierarchical union* of TRSs, and showed that this technique allows to handle some examples of higher-order programs. This approach is interesting but it is not easy to assess its generality, besides particular examples. In the present work we want to switch to a higher-order interpretations setting, in order to provide a more abstract account of such situations.

We thus propose to generalize TRS techniques to systems of higher-order rewriting, which come in many different flavours [26,28,35]. The majority of the introduced higher-order generalizations of rewriting are quite powerful but also complex from a computational point of view, being conceived to model not only programs but also proofs involving quantifiers. As an example, even computing the reduct of a term according to a reduction rule can in some cases be undecidable. Higher-order generalizations of TRS techniques [27,34], in turn, reflect the complexity of the languages on top of which they are defined. Summing up, devising ICC systems this way seems quite hard.

In this paper, we consider one of the simplest higher-order generalizations of TRSs, namely Yamada's simply-typed term rewriting systems [35] (STTRSs in the following), we define a system of higher-order polynomial interpretations [34] for them and prove that, following [10], this allows to exactly characterize the class of polynomial time computable functions. We show, however, that this way the class of (higher-order) programs which can be given a polynomial interpretation does not include interesting and natural examples, like `foldr`, and that this problem can be overcome by switching to another technique, designed along the lines of quasi-interpretations [11]. This is the subject of sections 4 and 5, which also show how non-trivial examples can be proved polytime this way.

Another problem we address in this paper is related to the expressive power of simply-typed term rewriting systems. Despite their simplicity, simply-typed term rewriting systems subsume the simply-typed λ-calculus and extensions of it with full recursion, like PCF. This can be proved following [19] and is the subject of Section 3.

A preliminary version of this work appeared as a conference paper in [6]. Compared to this short version, the main contributions of the present paper are the following ones:

- detailed proofs of the results (several of them had to be omitted or only sketched in [6] because of space constraints);
- description of translations of typed λ-calculus and PCF into STTRS (Section 3);
- more examples of programs to which one can apply the complexity criterion of higher-order quasi-interpretations (Section 5.6);
- discussion of embeddings of other ICC systems into our setting (Section 6).

2. Simply-typed term rewriting systems

2.1. Basic definitions and notation

We recall here the definition of a simply-typed term rewriting systems (STTRS), following [35,2]. We will actually consider a subclass of STTRSs, basically the one of those STTRSs whose rules' left hand side consists of a function symbol applied to a sequence of *patterns*. For first-order rewriting systems this corresponds to the notion of a *constructor rewriting system*.

We consider a denumerable set of base types, which we call *data-types*, that we denote with metavariables like D and E. *Types* are defined by the following grammar:

$$A, A_i ::= D \mid A_1 \times \cdots \times A_n \to A.$$

A *functional type* is a type which contains an occurrence of $\to$. Some examples of base types are the type W_n of strings over an alphabet of n symbols, and the type NAT of tally integers.

We denote by $\mathcal{F}$ the set of *function symbols* (or just *functions*), $\mathcal{C}$ the set of *constructors* and $\mathcal{X}$ the set of *variables*. Constructors $\mathbf{c} \in \mathcal{C}$ have a type of the form $D_1 \times \cdots \times D_n \to D$, for $n \geq 0$. For instance W_n has constructors **empty** of type W_n and $\mathbf{c}_1, \ldots, \mathbf{c}_n$ of type $W_n \to W_n$. Functions $\mathtt{f} \in \mathcal{F}$, on the other hand, can themselves have any functional type. Variables $x \in \mathcal{X}$ can have any type. *Terms* are typed and defined by the following grammar:

$$t, t_i := x^A \mid \mathbf{c}^A \mid \mathtt{f}^A \mid (t^{A_1 \times \cdots \times A_n \to A}\, t_1^{A_1} \ldots t_n^{A_n})^A,$$

where $x^A \in \mathcal{X}$, $\mathbf{c}^A \in \mathcal{C}$, $\mathtt{f}^A \in \mathcal{F}$. We denote by $\mathcal{T}$ the set of all terms. Observe how application is primitive and is in general treated differently from other function symbols. This is what makes STTRSs different from ordinary TRSs. $FV(t)$ is the set