Higher-Order Interpretations and Program Complexity*

Patrick Baillot[1] and Ugo Dal Lago[2]

1 **CNRS, ENS de Lyon, INRIA, UCBL, Université de Lyon, Laboratoire LIP**
patrick.baillot@ens-lyon.fr
2 **Università di Bologna & INRIA**
dallago@cs.unibo.it

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 [8]. 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 the notion of simply-typed term rewriting systems [30], we define higher-order polynomial interpretations for them and 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.

1998 ACM Subject Classification F.3.2 Semantics of Programming Languages

Keywords and phrases implicit complexity, higher-order rewriting, quasi-interpretations

Digital Object Identifier 10.4230/LIPIcs.CSL.2012.62

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: they consist in characterizations of complexity classes in terms of paradigmatic programming languages (recursion schemes [25, 6], λ-calculus [26], term rewriting systems [8], 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) [8, 9], whose complexity can be kept under control by way of variations of the powerful techniques developed to check termination of TRSs, namely path orderings [16], dependency pairs [28] and the interpretation method [24]. Many

* This work was partially supported by the ARC INRIA project "ETERNAL" and by the ANR-08-BLANC-0211-01 project "COMPLICE".

Computer Science Logic 2012 (CSL'12).
Editors: Patrick Cégielski, Arnaud Durand; pp. 62–76

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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 very powerful, in particular when the interpretation method is relaxed and coupled with recursive path orderings, like in quasi-interpretations [9].

The cited results indeed represent the state-of-the art in resource 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 if 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 [15]. However, it seems difficult to ensure in that case that the target first-order programs satisfy termination criteria such as those used in [9]. The article [10] 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 [21, 23, 30]. 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 [22, 29], 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 [30] (STTRSs in the following), we define a system of higher-order polynomial interpretations [29] for them and prove that, following [8], this allows to exactly characterize, among others, 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 [9]. This is the subject of sections 3 and 4 below.

An extended version of this paper with all proofs is available [3].

2 Simply-Typed Term Rewriting Systems

2.1 Definitions and Notations

We recall here the definition of a STTRS, following [30, 2]. We will actually consider a subclass of STTRSs, basically the one of those STTRSs whose rules' left hand side consists in a function symbol applied to a sequence of *patterns*. For first-order rewrite systems this corresponds to the notion of *constructor rewrite system*.

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

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

A *functional type* is a type which contains an occurrence of $\to$. Some examples of base types are the type W_n of n-ary words 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 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 of variables occurring in t. t is closed iff $FV(t) = \emptyset$.

To simplify the writing of terms we will often elide their type. We will also write $(t\ \overline{s})$ for $(t\ s_1 \ldots s_n)$. Therefore any term t is of the form $(\ldots((\alpha\ \overline{s_1})\ \overline{s_2})\ldots\overline{s_k})$ where $k \geq 0$ and $\alpha \in \mathcal{X} \cup \mathcal{C} \cup \mathcal{F}$. We will also use the following convention: any term t of the form $(\ldots((s\ \overline{s_1})\ \overline{s_2})\ldots\overline{s_k})$ will be written $(\!(s\ \overline{s_1}\ \ldots\overline{s_k})\!)$ or $(\!(s\ s_{11}\ \ldots s_{1n_1}\ldots s_{k1}\ldots s_{kn_k})\!)$. Observe however that, e.g., if t has type $A_1 \times A_2 \to (B_1 \times B_2 \to B)$, t_i has type A_i for $i = 1, 2$, s_i has type B_i for $i = 1, 2$, then both $(t\ t_1\ t_2)$ and $((t\ t_1\ t_2)\ s_1\ s_2)$ are well-typed (with type $B_1 \times B_2 \to B$ and B, respectively), but $(t\ t_1)$ and $(t\ t_1\ t_2\ s_1)$ are not well-typed. We define the size $|t|$ of a term t as the number of symbols (elements of $\mathcal{F} \cup \mathcal{C} \cup \mathcal{X}$) it contains. We denote $t\{x/s\}$ the substitution of term s for x in t.

A *pattern* is a term generated by the following grammar:

$$p, p_i := x^A \mid (\mathbf{c}^{D_1 \times \ldots \times D_n \to D}\ p_1^{D_1} \ldots p_n^{D_n}).$$

$\mathcal{P}$ is the set of all patterns. Observe that patterns of functional type are necessarily variables. We consider *rewriting rules* in the form $t \to s$ such that:

1. t and s are terms of the same type A, $FV(s) \subseteq FV(t)$, and any variable appears at most once in t;
2. t must have the form $(\!(\mathtt{f}\ \overline{p_1} \ldots \overline{p_k})\!)$ where each $\overline{p_i}$ for $i \in 1, \ldots, k$ consists of patterns only. The rule is said to be a rule *defining* $\mathtt{f}$, while the total number of patterns in $\overline{p_1}, \ldots, \overline{p_k}$ is the *arity* of the rule.

Now, a *simply-typed term rewriting system* (STTRS in the following) is a set R of non-overlapping rewriting rules such that for every function symbol $\mathtt{f}$, every rule for $\mathtt{f}$ has the same arity, which is said to be the arity *of* $\mathtt{f}$. A *program* $\mathtt{P} = (\mathtt{f}, R)$ is given by a STTRS R and a chosen function symbol $\mathtt{f} \in \mathcal{F}$.

In the next section, a notion of reduction will be given which crucially relies on the concept of a value. More specifically, only values will be passed as arguments to functions. Formally, we say that a term is a *value* if either:

1. it has a type D and is in the form $(\mathbf{c}\ v_1 \ldots v_n)$, where $v_1, \ldots, v_n$ are themselves values;
2. or it has functional type A and is of the form $(\!(\mathtt{f}\ v_1\ \ldots\ v_n)\!)$, where the terms in $v_1, \ldots\ v_n$ are themselves values and n is *strictly* smaller than the arity of $\mathtt{f}$.

Condition 2 is reminiscent of the λ-calculus, where an abstraction is a value. We denote values as v, u and the set of all values as $\mathcal{V}$.

2.2 STTRSs: Dynamics

The evaluation of terms will be formalized by a rewriting relation. Before that we need to introduce notions of substitution and unification.

A substitution σ is a mapping from variables to values with a finite domain, and such that $\sigma(x^A)$ has type A. A substitution σ is extended in the natural way to a function from $\mathcal{T}$ to itself, that we shall also write σ. The image of a term t under the substitution σ is denoted $t\sigma$. *Contexts* are defined as terms but with the proviso that they contain exactly one occurrence of a special constant $\bullet^A$ (*hole*) having type A. They are denoted as $\mathcal{C}$, $\mathcal{D}$ ... If $\mathcal{C}$ is a context with hole $\bullet^A$, and t is a term of type A, then $\mathcal{C}\{t\}$ is the term obtained from $\mathcal{C}$ by replacing the occurrence of $\bullet^A$ by t. Consider a STTRS R. We say that s reduces to t in call-by-value, denoted as $s \to_R t$, if there exists a rule $l \to r$ of R, a context $\mathcal{C}$ and a substitution σ such that $l\sigma$ is a closed term, $s = \mathcal{C}\{l\sigma\}$ and $t = \mathcal{C}\{r\sigma\}$. When there is no ambiguity on R, we simply write $\to$ instead of $\to_R$.

2.3 Typed λ-calculi as STTRSs

Please notice that one of the advantages of STTRSs over similar formalisms (like [23]) is precisely the simplicity of the underlying unification mechanism, which does not involve any notion of binding and is thus computationally simpler than higher-order matching. There is a price to pay in terms of expressivity, obviously. The choice of the STTRS framework as higher-order calculus is *not* too restrictive, however: one can show that typed λ-calculi equipped with *weak call-by-value reduction* can be seen as STTRSs. This is achieved using ideas developed for encodings of the λ-calculus into first-order term rewrite systems [15]. In particular, abstractions become function symbols, in the spirit of λ-lifting. More about these embeddings can be found in [3], where encodings of PCF and Gödel's T are described in detail.

3 Higher-Order Polynomial Interpretations

We want to demonstrate how first-order rewriting-based techniques for ICC can be adapted to the higher-order setting. Our goal is to devise criteria ensuring complexity bounds on programs of *first-order type* possibly containing subprograms of *higher-order types*. A typical application will be to find out under which conditions a higher-order functional program such as *e.g.* `map`, `iteration` or `foldr`, fed with a (first-order) polynomial time program produces a polynomial time program.

As a first illustrative step we consider the approach based on polynomial interpretations from [8], which offers the advantage of simplicity. We thus build a theory of *higher-order polynomial interpretations* for STTRSs. It starts from a particular concrete instantiation of the methodology proposed in [30] for proving termination by interpretation, on which we prove additional properties in order to obtain polynomial time complexity bounds.

Higher-order polynomials (HOPs) take the form of terms in a typed λ-calculus whose only base type is that of natural numbers. To each of those terms can be assigned a strictly monotonic function in a category $\mathbb{FSPOS}$ with products and functions. So, the whole process can be summarized by the following diagram:

$$\text{STTRSs} \xrightarrow{[\cdot]} \text{HOPs} \xrightarrow{[\![\cdot]\!]} \mathbb{FSPOS}$$

3.1 Higher-Order Polynomials

Let us consider types built from a base type $\mathbf{N}$:

$$A, B ::= \mathbf{N} \mid A \to A.$$

The expression $A^n \to B$ stands for the type $\underbrace{A \to \ldots \to A}_{n \text{ times}} \to B$. Let C_P be the following set of constants: $C_P = \{+ : \mathbf{N}^2 \to \mathbf{N}, \times : \mathbf{N}^2 \to \mathbf{N}\} \cup \{\overline{n} : \mathbf{N} \mid \ n \in \mathbb{N}^\star\}$. Observe that in C_P we have constants of type $\mathbf{N}$ only for *strictly* positive integers. We consider the following grammar of Church-typed terms:

$$M := x^A \ \mid \ \mathbf{c}^A \mid (M^{A\to B} N^A)^B \mid (\lambda x^A.M^B)^{A\to B},$$

where $\mathbf{c}^A \in C_P$ and *in* $(\lambda x^A.M^B)$ *we require that* x *occurs free in* M. A *higher-order polynomial* (HOP) is a term of this grammar which is in β-normal form. We use an infix notation for $+$ and $\times$. We assume given the usual set-theoretic interpretation of types and terms, denoted as $[\![A]\!]$ and $[\![M]\!]$: if M has type A and $FV(M) = \{x_1^{A_1}, \ldots, x_n^{A_n}\}$, then $[\![M]\!]$ is a map from $[\![A_1]\!] \times \ldots \times [\![A_n]\!]$ to $[\![A]\!]$. We denote by $\equiv$ the equivalence relation which identifies terms which denote the same function, e.g. we have: $\lambda x.(\overline{2} \times ((\overline{3} + x) + y)) \equiv \lambda x.(\overline{6} + (\overline{2} \times x + \overline{2} \times y))$. Noticeably, even if HOPs can be built using higher-order functions, the first order fragment only contains polynomials:

▶ **Lemma 3.1.** *If* M *is a HOP of type* $\mathbf{N}^n \to \mathbf{N}$ *and such that* $FV(M) = \{y_1 : \mathbf{N}, \ldots, y_k : \mathbf{N}\}$*, then the function* $[\![M]\!]$ *is bounded by a polynomial function.*

3.2 Semantic Interpretation

Now, we consider a subcategory $\mathbb{FSPOS}$ of the category $\mathbb{SPOS}$ of strict partial orders as objects and *strictly monotonic* total functions as morphisms. Objects of $\mathbb{FSPOS}$ are freely generated as follows:

- $\mathcal{N}$ is the domain of *strictly positive* integers, equipped with the natural strict order $\prec_{\mathcal{N}}$,
- if σ, τ are objects, then $\sigma \times \tau$ is obtained by the product ordering,
- $\sigma \to \tau$ is the set of strictly monotonic total functions from σ to τ, equipped with the following strict order: $f \prec_{\sigma\to\tau} g$ if for any a of σ we have $f(a) \prec_\tau g(a)$.

Actually we will also need to compare the semantics of terms which do not have the same free variables. For that we define: if $f \in \sigma_1 \times \ldots \times \sigma_n \to \tau$, $g \in \sigma_1 \times \ldots \times \sigma_m \to \tau$ and $n \leq m$, then: $f \prec g$ if $\forall a_1 \in \sigma_1, \ldots \forall a_m \in \sigma_m,\ f(a_1, \ldots, a_n) \prec_\tau g(a_1, \ldots, a_m)$.

$\mathbb{FSPOS}$ is a subcategory of $\mathbb{SET}$ with all the necessary structure to interpret types and terms. $[\![A]\!]_\prec$ denotes the semantics of A as an object of $\mathbb{FSPOS}$: we choose to set $[\![\mathbf{N}]\!]_\prec = \mathcal{N}$, while $[\![A_1 \times \ldots \times A_n \to A]\!]_\prec$ is $[\![A_1]\!]_\prec \times \ldots \times [\![A_n]\!]_\prec \to [\![A]\!]_\prec$. Let M be a HOP of type A with free variables $x_1^{A_1}, \ldots, x_n^{A_n}$. Then for every $e \in [\![A_1 \times \ldots \times A_n]\!]_\prec$, there is a naturally defined $f \in [\![A]\!]_\prec$. Moreover, this correspondence is strictly monotone and thus defines an element of $[\![A_1 \times \ldots \times A_n \to A]\!]_\prec$ which we denote as $[\![M]\!]_\prec$.

3.3 Assignments and Polynomial Interpretations

We consider $\mathcal{X}$, $\mathcal{C}$ and $\mathcal{F}$ as in Sect. 2. To each variable x^A we associate a variable $\underline{x}^{\underline{A}}$ where $\underline{A}$ is obtained from A by replacing each occurrence of base type by the base type $\mathbf{N}$ and by curryfication. We will sometimes write x (resp. A) instead of $\underline{x}$ (resp. $\underline{A}$) when it is clear from the context.

An *assignment* $[\,\cdot\,]$ is a map from $\mathcal{C} \cup \mathcal{F}$ to HOPs such that if $f \in \mathcal{C} \cup \mathcal{F}$ has type A, $[f]$ is a closed HOP of type $\underline{A}$. Now, for $t \in \mathcal{T}$ of type A, we define an HOP $[t]$ of type $\underline{A}$ by induction on t:

- if $t = x \in \mathcal{X}$, then $[t]$ is $\underline{x}$;
- if $t \in \mathcal{C} \cup \mathcal{F}$, $[t]$ is already defined;

- otherwise, if $t = (t_0\ t_1 \dots t_n)$ then $[t] \equiv (\dots([t_0][t_1])\dots[t_n])$.

Observe that in practice, computing $[t]$ will in general require to do some β-reduction steps.

Now, we say that an assignment $[\,\cdot\,]$ is a *higher polynomial interpretation* or simply a *polynomial interpretation* for a STTRS R iff for every $l \to r \in R$, we have that $[\![r]\!]_\prec \prec [\![l]\!]_\prec$. Note that in the particular case where the program only contains first-order functions, this notion of polynomial interpretation coincides with the classical one for first-order TRSs. In the following, we assume that $[\,\cdot\,]$ is a polynomial interpretation for R. A key property is the following, which tells us that the interpretation of terms strictly decreases along any reduction step:

▶ **Lemma 3.2.** *If $s \to t$, then $[\![t]\!]_\prec \prec [\![s]\!]_\prec$.*

As a consequence, the interpretation of terms (of base type) is itself a bound on the length of reduction sequences:

▶ **Proposition 3.3.** *Let t be a closed term of base type D. Then $[t]$ has type $\mathbf{N}$ and any reduction sequence of t has length bounded by $[\![t]\!]_\prec$.*

3.4 A Complexity Criterion

Proving a STTRS to have a polynomial interpretation is not enough to guarantee its time complexity to be polynomially bounded. To ensure that, we need to impose some constraints on the way constructors are interpreted.

We say that the assignment $[\,\cdot\,]$ is *additive* if any constructor $\mathbf{c}$ of type $D_1 \times \dots \times D_n \to D$, where $n \geq 0$, is interpreted by a HOP $M_{\mathbf{c}}$ whose semantic interpretation $[\![M_{\mathbf{c}}]\!]_\prec$ is a polynomial function of the form: $p(y_1, \dots, y_n) = \sum_{i=1}^{n} y_i + \gamma_{\mathbf{c}}$, with $\gamma_{\mathbf{c}} \geq 1$. Additivity ensures that the interpretation of first-order values is proportional to their size:

▶ **Lemma 3.4.** *Let $[\,\cdot\,]$ be an additive assignment. Then there exists $\gamma \geq 1$ such that for any value v of type D, we have $[\![v]\!]_\prec \leq \gamma \cdot |v|$.*

A function $f : (\{0,1\}^*)^m \to \{0,1\}$ is said to be *representable* by a STTRS R if there is a function symbol $\mathtt{f}$ of type $(W_2)^n \to W_2$ in R which computes f in the obvious way. We can now state the main result about polynomial interpretations:

▶ **Theorem 3.5** (Polynomial Bound)**.** *Let R be a STTRS with an additive polynomial interpretation $[\,\cdot\,]$. Consider a function symbol $\mathtt{g}$ of type $(W_2)^n \to W_2$. Then, there exists a polynomial $p : \mathbb{N}^n \to \mathbb{N}$ such that, for any $w_1, \dots, w_n \in \{0,1\}^\star$, any reduction of $(\mathtt{g}\ \underline{w_1} \dots \underline{w_n})$ has length bounded by $p(|w_1|, \dots, |w_n|)$. This holds more generally for $\mathtt{g}$ of type $D_1 \times \dots \times D_n \to D$.*

A key property we use in the proof of Theorem 3.5 is that function symbols of first-order type are interpreted by functions which are bounded by a polynomial. This is a consequence of Lemma 3.1 and is the main reason why we have chosen to define the interpretation of terms via HOPs.

▶ **Corollary 3.6.** *The functions on binary words representable by STTRSs admitting an additive polynomial interpretation are exactly the polytime functions.*

The fact that all polynomial time functions can be represented follows from [8], essentially because our setting subsumes that of this paper.

The results we have just described are quite robust: one is allowed to extend C_P with new combinators, provided their set-theoretic semantics are strictly monotone functions for

which Lemma 3.1 continues to hold. However, the class of polynomial time STTRSs which can be proved such by way of higher-order polynomial interpretations is quite restricted, as we are going to argue.

3.5 Examples

Consider the STTRS defined by the following rules:

$$\begin{aligned} ((\texttt{map}\ f)\ \mathbf{nil}^D) &\to \mathbf{nil}^E; \\ ((\texttt{map}\ f)\ (\mathbf{cons}^D\ x\ xs)) &\to (\mathbf{cons}^E\ (f\ x)\ ((\texttt{map}\ f)\ xs)); \end{aligned}$$

with the following types:

$$\begin{aligned} f &: D \to E; & \texttt{map} &: (D \to E) \to L(D) \to L(E); \\ \mathbf{nil}^D &: L(D); & \mathbf{cons}^D &: D \times L(D) \to L(D); \\ \mathbf{nil}^E &: L(E); & \mathbf{cons}^E &: E \times L(E) \to L(E). \end{aligned}$$

Here $D, E, L(D), L(E)$ are base types. For simplicity we use just one **cons** and one **nil** notation for both types D and E. The interpretation below was given in [30] for proving termination, but here we show that it also gives a polynomial time bound. Now, we choose the following assignment of HOPs:

$$\begin{aligned} [\mathbf{nil}] &= 2 : \mathbf{N}; \\ [\mathbf{cons}] &= \lambda n.\lambda m.(n + m + 1) : \mathbf{N} \to \mathbf{N} \to \mathbf{N}; \\ [\texttt{map}] &= \lambda \phi.\lambda n.n \times (\phi\ n) : (\mathbf{N} \to \mathbf{N}) \to \mathbf{N} \to \mathbf{N}. \end{aligned}$$

One can check that the condition $[\![r]\!]_\prec \prec [\![l]\!]_\prec$ holds for both rules above. We thus have an additive polynomial interpretation for `map`, therefore Corollary 3.6 applies and we can conclude that for any `f` also satisfying the criterion, (`map f`) computes a polynomial time function.

Now, one might want to apply the same method to an iterator `iter`, of type $(D \to D) \times D \to NAT \to D$, which when fed with arguments f, d, n iterates f exactly n times starting from d. However there is no additive polynomial interpretation for this program. Actually, this holds for very good reasons: `iter` can produce an exponential-size function when fed with a fast-growing polynomial time function, *e.g.* `double` : $NAT \to NAT$.

One way to overcome this issue could be to show that `iter` *does* admit a valid polynomial interpretation, *provided* its domain is restricted to some particular functions, admitting a *small* polynomial interpretation, of the form $\lambda n.(n + c)$, for some constant c. This could be enforced by considering a refined type systems for HOPs. But the trouble is that there are very few programs which admit a polynomial interpretation of this form! Intuitively the problem is that polynomial interpretations need to bound simultaneously the execution time *and* the size of the intermediate values. In the sequel we will see how to overcome this issue.

4 Beyond Interpretations: Quasi-Interpretations

The previous section has illustrated our approach. However we have seen that the intensional expressivity of higher-order polynomial interpretations is too limited. In the first-order setting this problem has been overcome by decomposing into *two distinct conditions* the role

$$\frac{\mathtt{f}^{A} \in \mathcal{NF}}{\Gamma \mid \Delta \vdash \mathtt{f} : A} \qquad \frac{\mathbf{c}^{A} \in \mathcal{C}}{\Gamma \mid \Delta \vdash \mathbf{c} : A} \qquad \frac{}{\Gamma \mid x : A, \Delta \vdash x : A} \qquad \frac{}{x : D, \Gamma \mid \Delta \vdash x : D}$$

$$\frac{\mathtt{f}^{A_1,\ldots,A_n \to B} \in \mathcal{RF}, \text{with arity } n \qquad \Gamma \mid \emptyset \vdash s_i : A_i}{\Gamma \mid \Delta \vdash ((\mathtt{f}\ s_1 \ldots s_n)) : B} \qquad \frac{\Gamma \mid \Delta \vdash t : A_1 \times \ldots \times A_n \to B \qquad \Gamma \mid \Delta_i \vdash s_i : A_i}{\Gamma \mid \Delta, \Delta_1, \ldots, \Delta_n \vdash (t\ s_1 \ldots s_n) : B}$$

Figure 1 A Linear Type System for STTRS terms.

played by polynomial interpretations [27, 9]: (i) a termination condition, (ii) a condition enforcing a bound on the size of values occurring during the computation. In [9], this has been implemented by using: for (i) some specific recursive path orderings, and for (ii) a notion of *quasi-interpretation*. We will examine how this methodology can be extended to the higher-order setting.

The first step will take the form of a termination criterion defined by a linear type system for STTRSs together with a path-like order, to be described in Section 4.1 below. The second step consists in shifting from a semantic world of strictly monotonic functions to one of monotonic functions. This corresponds to a picture like the following, and is the subject of sections 4.2 and 4.3.

$$\text{STTRSs} \xrightarrow{[\cdot]} \text{HOMPs} \xrightarrow{[\![\cdot]\!]} \mathbb{FPOS}$$

4.1 The Termination Criterion

The termination criterion has two ingredients: a typing ingredient and a syntactic ingredient, expressed using an order $\sqsubset$ on the function symbols. Is it restrictive for expressivity? The syntactic ingredient is fairly expressive, since it allows to validate all programs coming from System T (see [3] for more details). As to the full termination criterion, including the typing ingredient, it is general enough to embed Hofmann's SLR [19] and LFPL [20], which are distinct restrictions of System T capturing polytime functions.

Formally, introducing the typing ingredient requires splitting the class $\mathcal{F}$ into two disjoint classes $\mathcal{RF}$ and $\mathcal{NF}$. The intended meaning is that functions in $\mathcal{NF}$ will not be defined in a recursive way, while functions in $\mathcal{RF}$ can. We further assume given a strict order $\sqsubset$ on $\mathcal{F}$ which is well-founded. If t is a term, $t \sqsubset \mathtt{f}$ means that for any $\mathtt{g}$ occurring in t we have $\mathtt{g} \sqsubset \mathtt{f}$. The rules of a linear type system for STTRS terms are in Figure 1. In a judgement $\Gamma \mid \Delta \vdash t : A$, the sub-context Δ is meant to contain linear variables while Γ is meant to contain non-linear variables.

A STTRS *satisfies the termination criterion* if every rule $((\mathtt{f}\ \overline{p_1} \ldots \overline{p_k})) \to s$ satisfies:

1. either $\mathtt{f} \in \mathcal{RF}$, there are a term r and sequences of patterns $\overline{q_1}, \ldots, \overline{q_k}$ such that $s = r\{x/((\mathtt{f}\ \overline{q_1} \ldots \overline{q_k}))\}$, we have $\Gamma \mid x : B, \Delta \vdash r : B$, $r \sqsubset \mathtt{f}$, for any i, j, $q_{i,j}$ is subterm of $p_{i,j}$ and there exist i_0, j_0 s.t. $q_{i_0,j_0} \neq p_{i_0,j_0}$;
2. or we have $\Gamma \mid \Delta \vdash s : B$ and $s \sqsubset \mathtt{f}$.

Observe that because of the typability condition in 1., this termination criterion implies that there is at most one recursive call in the right-hand-side s of a rule.

Given a term t, its *definitional depth* is the maximum, over any function symbol $\mathtt{f}$ ap-

$$
\begin{aligned}
\mathcal{TS}_v(X) =& 1, \quad \text{if } v \text{ is a first order value,}\\
\mathcal{TS}_{(\!(\mathtt{f}\ t_1 \ldots t_n)\!)}(X) =& 1 + \left(\sum_{\substack{t_j \in \mathcal{FO}\\ j \leq \mathit{arity}(\mathtt{f})}} \mathcal{TS}_{t_j}(X)\right) + \left(\sum_{\substack{t_j \in \mathcal{HO}\\ j \leq \mathit{arity}(\mathtt{f})}} n \cdot X \cdot \mathcal{TS}_{t_j}(X)\right)\\
&+ \left(\sum_{j \geq \mathit{arity}(\mathtt{f})+1} \mathcal{TS}_{t_j}(X)\right) + \left(\sum_{s \in \mathcal{R}(\mathtt{f})} n \cdot X \cdot \mathcal{TS}_s(X)\right), \quad \text{if } \mathtt{f} \in \mathcal{RF};\\
\mathcal{TS}_{(\!(\mathtt{f}\ t_1 \ldots t_n)\!)}(X) =& 1 + \left(\sum_{1 \leq j \leq n} \mathcal{TS}_{t_j}(X)\right) + \left(\sum_{s \in \mathcal{R}(\mathtt{f})} \mathcal{TS}_s(X)\right), \quad \text{if } \mathtt{f} \in \mathcal{NF};\\
\mathcal{TS}_{(\!(\mathtt{c}\ t_1 \ldots t_n)\!)}(X) =& 1 + \left(\sum_{1 \leq j \leq n} \mathcal{TS}_{t_j}(X)\right);\\
\mathcal{TS}_{(\!(x\ t_1 \ldots t_n)\!)}(X) =& 1 + \left(\sum_{1 \leq j \leq n} \mathcal{TS}_{t_j}(X)\right).
\end{aligned}
$$

Figure 2 The Definition of $\mathcal{TS}_{(\cdot)}(X)$.

pearing in t, of the length of the longest descending $\sqsubset$-chain starting from $\mathtt{f}$. The definitional depth of t is denoted as $\partial(t)$. By a standard reducibility argument, one can prove that every term of a STTRS satisfying the termination criterion is strongly normalizing (see again [3] for the details).

In the rest of this section, we show that all that matters for the time complexity of STTRSs satisfying the termination criterion is the size of first-order values that can possibly appear along the reduction of terms. In other words, we are going to prove that if the latter is bounded, then the complexity of the starting term is known, modulo a fixed polynomial. Showing this lemma, which will be crucial in the following, requires introducing many auxiliary definitions and results.

Given a term t and a natural number $n \in \mathbb{N}$, n is said to be *a bound of first order values for* t if for every reduct s of t, if s contains a first-order value v, then $|v| \leq n$. Suppose a function symbol $\mathtt{f}$ takes n base arguments. Then $\mathtt{f}$ is said to have *base values bounded by a function* $q : \mathbb{N}^n \to \mathbb{N}$ if $(\mathtt{f}\ t_1 \ldots t_n)$ has $q(|t_1|, \ldots, |t_n|)$ as a bound of its first-order values whenever $t_1, \ldots, t_n$ are first-order values. Given a function symbol $\mathtt{f}$, $\mathcal{R}(\mathtt{f})$ denotes the set of terms appearing in the right-hand side of rules for $\mathtt{f}$, not taking into account recursive calls. For every term t, define its *space-time weight* as a polynomial $\mathcal{TS}_t(X)$ on the indeterminate X, by induction on $(\partial(t), |t|)$, following the lexicographic order, as in Figure 2. We denote here by $\mathcal{FO}$ (resp. $\mathcal{HO}$) the arguments t_j of $\mathtt{f}$ of base type (resp. functional type). The *collapsed size* $||t||$ of a term t is its size, where however all first-order values count for 1. We define a rewrite relation $\Rightarrow$ which is like $\to$, except that whenever a recursive function symbol is unfolded, it is unfolded *completely* in just one rewrite step.

We are now ready to explain why the main result of this section holds. First of all, $\mathcal{TS}_t(X)$ is an upper bound on the collapsed size of t, a result which can be proved by induction on t:

▶ **Lemma 4.1.** *For every $n \geq 1$ and for every t, $\mathcal{TS}_t(n) \geq ||t||$.*

Moreover, $\mathcal{TS}_t(X)$ decreases along any $\Rightarrow$ step if X is big enough:

▶ **Lemma 4.2.** *If n is a bound of first-order values for t, and $t \Rightarrow s$, then $\mathcal{TS}_t(n) > \mathcal{TS}_s(n)$.*

It is now easy to reach our goal:

▶ **Proposition 4.3.** *Suppose that R satisfies the termination criterion. Moreover, suppose that $\mathtt{f}$ has base values bounded by a function $q : \mathbb{N}^n \to \mathbb{N}$. Then, there is a polynomial $p : \mathbb{N} \to \mathbb{N}$ such that if $t_1, \ldots, t_n$ are first-order values and $(\mathtt{f}\ t_1 \ldots t_n) \to^m s$, then $m, |s| \leq p(q(|t_1|, \ldots, |t_n|))$.*

To convince yourself that linearity is needed to get a result like Proposition 4.3, consider the following STTRS, whose terms cannot be typed in our linear type system:

$$\begin{array}{lcl}
((\mathtt{comp}\ x\ y)\ z) & \to & (x\ (y\ z)) \\
(\mathtt{autocomp}\ x) & \to & (\mathtt{comp}\ x\ x) \\
(\mathtt{id}\ x) & \to & x \\
(\mathtt{expid}\ \mathbf{0}) & \to & \mathtt{id} \\
(\mathtt{expid}\ (\mathtt{s}\ x)) & \to & (\mathtt{autocomp}\ (\mathtt{expid}\ x))
\end{array}$$

Both $\mathtt{id}$ and $(\mathtt{expid}\ t)$ (for every value t of type NAT) can be given type $NAT \to NAT$. Actually, they all are the same function, extensionally. But try to see what happens if $\mathtt{expid}$ is applied to natural numbers of growing sizes: there is an exponential blowup going on which does not find any counterpart in first-order values.

4.2 Higher-Order Max-Polynomials

We want to refine the type system for higher-order polynomials, in order to be able to use types to restrict the domain of functionals. The grammar of types is now the following one:

$$S ::= \mathbf{N} \mid S \multimap S; \qquad A ::= S \mid A \to A.$$

Types of the first (resp. second) grammar are called linear types (resp. types) and denoted as $R, S \ldots$ (resp. $A, B, C \ldots$). The linear function type $\multimap$ is a subtype of $\to$, i.e., one can define a relation $\sqsubseteq$ between types by stipulating that $S \multimap R \sqsubseteq S \to R$ and by closing the rule above in the usual way, namely by imposing that $A \to B \sqsubseteq C \to E$ whenever $C \sqsubseteq A$ and $B \sqsubseteq E$.

We now consider the following new set of constructors:

$$D_P = \{+ : \mathbf{N} \multimap \mathbf{N} \multimap \mathbf{N}, max : \mathbf{N} \multimap \mathbf{N} \multimap \mathbf{N}, \times : \mathbf{N} \to \mathbf{N} \to \mathbf{N}\} \cup \{\overline{n} : \mathbf{N} \mid\ n \in \mathbb{N}^\star\},$$

and we define the following grammar of Church-typed terms

$$M := x^A \mid \mathbf{c}^A \mid (M^{A \to B} N^A)^B \mid (\lambda x^A.M^B)^{A \to B} \mid (M^{S \multimap R} N^S)^R \mid (\lambda x^S.M^R)^{S \multimap R}$$

where $\mathbf{c}^A \in D_P$. We also require that:

- in $(\lambda x^A.M^B)^{A \to B}$, the variable x^A occurs *at least once* in M^B;
- in $(\lambda x^S.M^R)^{S \multimap R}$, the variable x^S occurs *exactly once* in M^R and in linear position (i.e., it cannot occur on the right-hand side of an application $N^{A \to B} L^A$).

One can check that this class of Church-typed terms is preserved by β-reduction. A *higher-order max-polynomial* (HOMP) is a term as defined above and which is in β-normal form. We define the following objects and constructions on objects:

- $\mathcal{N}$ is the domain of *strictly positive* integers, equipped with the natural partial order, denoted here $\leq_{\mathcal{N}}$,

- if σ, τ are objects, then $\sigma \times \tau$ is obtained by the product ordering,
- $\sigma \Rightarrow \tau$ is the set of monotonic total functions from σ to τ, equipped with the extensional order: $f \leq_{\sigma\Rightarrow\tau} g$ if for any a of σ we have $f(a) \leq_\tau g(a)$.

This way, one obtains a subcategory $\mathbb{FPOS}$ of the category $\mathbb{POS}$ with partial orders as objects and *monotonic* total functions as morphisms. As before with $\prec$ we define $\leq$ so as to compare the semantics of terms which do not have the same free variables.

In order to interpret the $\multimap$ construction in this category we introduce a notion of *size*. A size is a (finite) multiset of elements of $\mathbb{N}$. The empty multiset will be denoted $\emptyset$. Given a multiset $\mathcal{S}$, we denote by $\max \mathcal{S}$ its maximal element and by $\sum \mathcal{S}$ the sum of its elements. By convention $\max \emptyset = \sum \emptyset = 0$. Now, given an object σ of the category $\mathbb{FPOS}$, we say that an element $e \in \sigma$ *admits a size* in the following cases:

- If σ is $\mathcal{N}$, then e is an integer n, and $\mathcal{S}$ is a size of e iff we have: $\max \mathcal{S} \leq n \leq \sum \mathcal{S}$.
- If $\sigma = \sigma_1 \times \cdots \times \sigma_n$, then $\mathcal{S}$ is a size of $e = (e_1, \ldots, e_n)$ iff there exists for any $i \in \{1, \ldots, n\}$ a multiset $\mathcal{S}_i$ which is a size of e_i, and such that $\mathcal{S} = \cup_{i=1}^n \mathcal{S}_i$.
- If $\sigma = \tau \Rightarrow \rho$, then $\mathcal{S}$ is a size of e iff for any f of τ which has a size $\mathcal{T}$, $\mathcal{S} \cup \mathcal{T}$ is a size of $e(f)$. $\tau \twoheadrightarrow \rho$ is the subset of all those functions in σ which admit a size.

We denote by $[\![A]\!]_\leq$ the semantics of A as an object of $\mathbb{FPOS}$, where $\mathbf{N}$ is mapped to $\mathcal{N}$, $\to$ is mapped to $\Rightarrow$ and $\multimap$ to $\twoheadrightarrow$. As for HOPs, any HOMP M can be naturally interpreted as a monotonic function between the appropriate partial orders, which we denote by $[\![M]\!]_\leq$. We will speak of the *size* of an HOMP M, by which we mean a size of its interpretation $[\![M]\!]_\leq$. Note that not all terms admit a size. For instance $\times : \mathbf{N} \to \mathbf{N} \to \mathbf{N}$ does not admit a size. If M reduces to N, then they have the same sizes, if any. Let us examine some examples:

- The term $\overline{n}$ of type $\mathbf{N}$ admits the following sizes: $[n]$, $\underbrace{[1, \ldots, 1]}_{k \text{ times}}$ with $k \geq n$, and more generally $[n_1, \ldots, n_k]$ such that $\forall i \in \{1, k\}$, $n_i \leq n$ and $\sum_{i=1}^k n_i \geq n$.
- The terms max and $+$ of type $\mathbf{N} \multimap \mathbf{N} \multimap \mathbf{N}$ admit as size $\emptyset$ or $[0]$.
- The terms $\lambda x.(x + 3)$, $\lambda x. \max(x, 3)$ of type $\mathbf{N} \multimap \mathbf{N}$ both have size 3.
- The term $\lambda f.(f\ 2\ 3)$ of type $(\mathbf{N} \multimap \mathbf{N}) \multimap \mathbf{N}$ has size $[2, 3]$.

Actually, if we consider first-order terms with types of the form $\mathbf{N} \multimap \ldots \multimap \mathbf{N}$, it is sufficient to consider singletons as sizes. If we only wanted to deal with these terms we could thus use integers for sizes, instead of multisets. Non-singleton multisets only become necessary when we move to higher-order types, as in the last example above:

▶ **Proposition 4.4.** *If M is a HOMP of type $\mathbf{N}^k \multimap \mathbf{N}$ with free variables $x_1 : \mathbf{N}, \ldots, x_n : \mathbf{N}$ which are linear in M, then it admits a size of the form $[m]$ where $m \in \mathbb{N}$.*

The following will be useful to obtain the Subterm Property:

▶ **Lemma 4.5.** *For every type A there is a closed HOMP of type A.*

4.3 Higher-Order Quasi-Interpretations

Now, a HOMP assignment $[\,\cdot\,]$ is defined by: for any $f^A \in \mathcal{X}$ (resp. $f^A \in \mathcal{C} \cup \mathcal{F}$), $[f]$ is a variable $\underline{f}$ (resp. a closed HOMP M) with a type B, where B is obtained from (the curryfication of) A by:

- replacing each occurrence of a base type D by $\mathbf{N}$,
- replacing each occurrence of $\to$ in A by either $\to$ or $\multimap$.

For instance if $A = (D_1 \to D_2) \to D_3$ we can take for B any of the types: $(\mathbf{N} \multimap \mathbf{N}) \to \mathbf{N}$, $(\mathbf{N} \to \mathbf{N}) \to \mathbf{N}$, etc. In the sequel we will write $\underline{A}$ for any of these types B. Now,

if $t = (t_0\ t_1 \dots t_n)$ then $[t]$ is defined if for any $0 \leq i \leq n$, $[t_i]$ is defined and if $[t] \equiv (\dots([t_0][t_1])\dots[t_n])$ is well-typed. Additive HOMP assignments are defined just as additive HOP assignments. Now, we say that an assignment $[\,\cdot\,]$ is a *quasi-interpretation for* R if for any rule $l \to r$ of R, $[l]$ and $[r]$ are defined and have the same type, and it holds that $[\![l]\!]_{\leq} \geq [\![r]\!]_{\leq}$. Observe that contrarily to the case of polynomial interpretations, these inequalities are not strict, and moreover they are stated with respect to the new domains, taking into account the distinction between the two connectives $\to$ and $\multimap$.

The interpretation of a term does not, like in the strict case, necessarily decrease along a reduction step. However, it cannot increase: if $t \to^* s$, then $[\![s]\!]_{\leq} \leq [\![t]\!]_{\leq}$. This, together with the possibility of forming HOMPs of arbitrary type (Lemma 4.5) implies the following, crucial, property:

▶ **Proposition 4.6** (Subterm Property)**.** *Suppose that an STTRS R has an additive quasi-interpretation $[\,\cdot\,]$. Then, for every function symbol $\mathtt{f}$ of arity n with base arguments, there is a polynomial $p : \mathbb{N}^n \to \mathbb{N}$ such that if $(\mathtt{f}\ t_1 \dots t_n) \to^* s$ and if s contains an occurrence of a base term r, then $|r| \leq p(|t_1|, \dots, |t_n|)$.*

And here is the main result of this Section:

▶ **Theorem 4.7** (Polytime Soundness)**.** *If an STTRS R has an additive quasi-interpretation, R satisfies the termination criterion and $\mathtt{f}$ has arity n with base type arguments, then there is a polynomial $p : \mathbb{N}^n \to \mathbb{N}$ such that whenever $(\mathtt{f}\ t_1 \dots t_n) \to^m s$, it holds that $m, |s| \leq p(|t_1|, \dots, |t_n|)$. So if $\mathtt{f}$ has a type $D_1 \times \dots \times D_n \to D$ then instances of $\mathtt{f}$ can be computed in polynomial time.*

Proof. A consequence of Proposition 4.6 and of Proposition 4.3. ◀

Notice how Theorem 4.7 is proved by first observing that terms of STTRSs having a quasi-interpretation are bounded by natural numbers which are not too big with respect to the input, thus relying on the termination criterion to translate these bounds to *complexity* bounds.

Higher-order quasi interpretations, like their strict siblings, can be extended by enlarging D_P so as to include more combinators, provided they are bounded by polynomials. One of these extensions is discussed in [3] and allows to (re)prove LFPL programs to represent polytime functions.

4.4 Examples

Consider the program `foldr` given by:

$$((\mathtt{foldr}\ f\ b)\ \mathbf{nil}) \to b; \tag{1}$$

$$((\mathtt{foldr}\ f\ b)\ (\mathbf{cons}\ x\ xs)) \to (f\ x\ ((\mathtt{foldr}\ f\ b)\ xs)); \tag{2}$$

where functions, variables and constructors have the following types:

$$\mathtt{foldr} : (D \times E \to E) \times E \to L(D) \to E; \qquad f : D \times E \to E;$$

and **nil**, **cons** typed as in Sect. 3.5. Now, we choose as assignment:

$$[\mathbf{nil}] = 1 : \mathbf{N}; \qquad [\mathbf{cons}] = \lambda n.\lambda m.n + m + 1 : \mathbf{N} \to \mathbf{N} \to \mathbf{N};$$

$$[\mathtt{foldr}] = \lambda\phi.\lambda p.\lambda n.p + n \times (\phi\ 1\ 1) : (\mathbf{N} \multimap \mathbf{N} \multimap \mathbf{N}) \to \mathbf{N} \to \mathbf{N} \to \mathbf{N}.$$

Observe the $\multimap$ in the type of the first argument of [`foldr`] which is the way to restrict the domain of arguments. We then obtain the following interpretations of terms:

$$\begin{aligned}
[((\texttt{foldr}\ f\ b)\ \mathbf{nil})] &= \underline{b} + 1 \times (\underline{f}\ 1\ 1);\\
[((\texttt{foldr}\ f\ b)\ (\mathbf{cons}\ x, xs))] &= \underline{b} + (\underline{x} + \underline{xs} + 1) \times (\underline{f}\ 1\ 1);\\
[(f\ x\ ((\texttt{foldr}\ f\ b)\ xs))] &= \underline{f}\ \underline{x}\ (\underline{b} + \underline{xs} \times (\underline{f}\ 1\ 1)).
\end{aligned}$$

It is easy to see that condition $[\![r]\!]_{\leq} \leq [\![l]\!]_{\leq}$ holds for rule (1). As to rule (2) consider $\phi \in \mathcal{N} \multimap \mathcal{N} \multimap \mathcal{N}$. We know that ϕ has a size $c \geq 0$, and thus for every $x, y \in \mathcal{N}$,

$$c \leq \phi\ x\ y \leq x + y + c. \tag{3}$$

Then we have:

$$\begin{aligned}
\underline{f}\ \underline{x}\ (\underline{b} + \underline{xs} \times \underline{f}(1,1)) &\leq \underline{x} + \underline{b} + \underline{xs} \times (\underline{f}\ 1\ 1) + c \leq \underline{x} \times (\underline{f}\ 1\ 1) + \underline{b} + \underline{xs} \times (\underline{f}\ 1\ 1) + c\\
&\leq \underline{b} + (\underline{x} + \underline{xs} + 1) \times (\underline{f}\ 1\ 1),
\end{aligned}$$

where for the two last steps we used $(\underline{f}\ 1\ 1) \geq 1$ and $(\underline{f}\ 1\ 1) \geq c$ (because of (3)). So $[\![r]\!]_{\leq} \leq [\![l]\!]_{\leq}$ also holds for (2) and we have an additive quasi-interpretation. As to the termination criterion, it is satisfied because in rule (2), xs is a strict sub-pattern of $(\mathbf{cons}\ x\ xs)$ and the term $(f\ x\ y)$ can be typed in the linear type system as required. Summing up, we can apply Theorem 4.7 and conclude that if the termination criterion is satisfied by all functions, if $t^{D \times E \to E}$, b^E are terms and $[t]$ is a HOMP with type $\mathbf{N} \multimap \mathbf{N} \multimap \mathbf{N}$, then $(\texttt{foldr}\ t\ b)$ is a polynomial time program of type $L(D) \to E$.

Note that the idea of ensuring complexity bounds when the programs are fed with functional arguments admitting additional conditions had already been suggested in [10], on particular examples. The present setting using types, however, brings a more systematic account of this property.

5 Discussion and Relation with Other ICC Systems

The authors believe that the interest of the present work does not lie much in bringing yet another ready-to-use ICC system but rather in offering a new *framework* in which to design ICC systems and prove their complexity properties. Indeed, considered as an ICC system our setting presents two limitations:

1. given a program one needs to *find* an assignment and to *check* that it is a valid quasi-interpretation, which in general will be difficult to automatize;
2. the termination criterion currently does not allow to reuse higher-order arguments in full generality.

To overcome 2. we think it will be possible to design more liberal termination criteria, while attacking 1. could possibly consist in defining type systems such that if a program is well-typed, then it admits a quasi-interpretation, and for which one could devise type-inference algorithms. On the other hand, recently introduced techniques for inferring higher-order polynomial interpretations [17] could shed some light on this issue, which is however outside the scope of this paper.

Related and Further Work. Let us first compare our approach to other frameworks for proving complexity soundness results. At first-order, we have already emphasized the fact that our setting is an extension of the quasi-interpretation approach of [9] (see also [1] for the relation with non-size-increasing, at first-order). At higher-order, various approaches based on realizability have been used [14, 11]. While these approaches were developed for

logics or System T-like languages, our setting is adapted to a language with recursion and pattern-matching. We think it might also be easier to use in practice.

Let us now discuss the relations with known ICC systems. Several variants of System T based on restriction of recursion and linearity conditions [19, 7, 12] have been proposed which characterize polynomial time. Another system [20] based on a linear type system for non-size-increasing computation, called LFPL, offers more intensional expressivity. Terms of the latter calculus can indeed be proved to be reducible in polynomial time by showing they admit quasi-interpretations and satisfy the termination criterion (details are omitted here, due to space constraints, but can be found in [3]). With respect to [20], the advantages we bring are a slightly more general handling of higher-order arguments, but also the possibility to capture size-increasing polytime algorithms. As an example, we are able to assign a quasi-interpretation to (STTRSs computing) functions in Bellantoni and Cook's algebra BC [6] (see again [3]).

Some other works are based on type systems built out of variants of linear logic [5, 18, 4]. They are less expressive for first-order functions but offer more liberal disciplines for handling higher-order arguments. In future work we will examine if they could suggest a more flexible termination condition for our setting, maybe itself based on quasi-interpretations, following [13].

6 Conclusions

We have advocated the usefulness of simply typed term rewriting systems to smoothly extend notions from first-order rewrite systems to the higher-order setting. Our main contribution is a new framework for studying (and distilling) ICC systems for higher-order languages. While up to now quite distinct techniques had been successful for providing expressive criteria for polynomial time complexity at first-order and at higher-order respectively, our approach brings together these techniques: interpretation methods on the one hand, and semantic domains and type systems on the other. We have illustrated the strength of this framework by designing an ICC system for polynomial time based on a termination criterion and on quasi-interpretations, which allows to give some sufficient conditions for programs built with higher-order functionals (like `foldr`) to work in bounded time. We think this setting should allow in future work to devise new, more expressive, systems for ensuring complexity bounds for higher-order languages.

References

1 Roberto Amadio. Synthesis of max-plus quasi-interpretations. *Fundam. Inform.*, 65:29–60, 2005.

2 Takahito Aoto and Toshiyuki Yamada. Termination of simply typed term rewriting by translation and labelling. In *RTA 2003*, volume 2706 of *LNCS*. Springer, 2003.

3 Patrick Baillot and Ugo Dal Lago. Higher-order interpretations and program complexity (long version). Available at `http://hal.archives-ouvertes.fr/hal-00667816`, 2012.

4 Patrick Baillot, Marco Gaboardi, and Virgile Mogbil. A polytime functional language from light linear logic. In *ESOP 2010*, volume 6012 of *LNCS*, pages 104–124, 2010.

5 Patrick Baillot and Kazushige Terui. Light types for polynomial time computation in lambda calculus. *Inf. Comput.*, 207(1):41–62, 2009.

6 Stephen J. Bellantoni and Stephen A. Cook. A new recursion-theoretic characterization of the poly-time functions. *Computational Complexity*, 2:97–110, 1992.

7 Stephen J. Bellantoni, Karl-Heinz Niggl, and Helmut Schwichtenberg. Higher type recursion, ramification and polynomial time. *Ann. Pure Appl. Logic*, 104(1-3):17–30, 2000.

8 Guillaume Bonfante, Adam Cichon, Jean-Yves Marion, and Hélène Touzet. Algorithms with polynomial interpretation termination proof. *J. Funct. Program.*, 11(1):33–53, 2001.

9 Guillaume Bonfante, J.-Y. Marion, and Jean-Yves Moyen. Quasi-interpretations a way to control resources. *Theor. Comput. Sci.*, 412(25):2776–2796, 2011.

10 Guillaume Bonfante, Jean-Yves Marion, and Romain Péchoux. Quasi-interpretation synthesis by decomposition. In *ICTAC 2007*, volume 4711 of *LNCS*, pages 410–424. Springer, 2007.

11 Aloïs Brunel and Kazushige Terui. Church => Scott = Ptime: an application of resource sensitive realizability. In *DICE 2010*, volume 23 of *EPTCS*, pages 31–46, 2010.

12 Ugo Dal Lago. The geometry of linear higher-order recursion. In *LICS 2005*, pages 366–375, 2005.

13 Ugo Dal Lago and Marco Gaboardi. Linear dependent types and relative completeness. In *LICS 2011*, pages 133–142, 2011.

14 Ugo Dal Lago and Martin Hofmann. Realizability models and implicit complexity. *Theor. Comput. Sci.*, 412(20):2029–2047, 2011.

15 Ugo Dal Lago and Simone Martini. On constructor rewrite systems and the lambda-calculus. In *ICALP 2009*, volume 5556 of *LNCS*, pages 163–174. Springer, 2009.

16 Nachum Dershowitz. Orderings for term-rewriting systems. *Theor. Comput. Sci.*, 17(3):279–301, 1982.

17 Carsten Fuhs and Cynthia Kop. Polynomial interpretations for higher-order rewriting. In *RTA 2012*, volume 15 of *LIPIcs*, pages 176–192. Schloss Dagstuhl, 2012.

18 Marco Gaboardi and Simona Ronchi Della Rocca. A soft type assignment system for lambda -calculus. In *CSL 2007*, volume 4646 of *LNCS*, pages 253–267. Springer, 2007.

19 Martin Hofmann. A mixed modal/linear lambda calculus with applications to Bellantoni-Cook safe recursion. In *CSL 1997*, volume 1414 of *LNCS*, pages 275–294, 1997.

20 Martin Hofmann. Linear types and non-size-increasing polynomial time computation. *Inf. Comput.*, 183(1):57–85, 2003.

21 Jean-Pierre Jouannaud and Mitsuhiro Okada. A computation model for executable higher-order algebraic specification languages. In *LICS 1991*, pages 350–361, 1991.

22 Jean-Pierre Jouannaud and Albert Rubio. The higher-order recursive path ordering. In *LICS 1999*, pages 402–411, 1999.

23 Jan Willem Klop, Vincent van Oostrom, and Femke van Raamsdonk. Combinatory reduction systems: Introduction and survey. *Theor. Comput. Sci.*, 121(1&2):279–308, 1993.

24 D. Lankford. On proving term rewriting systems are noetherian. Technical Report MTP-3, Louisiana Tech. University, 1979.

25 D. Leivant. Predicative recurrence and computational complexity I: word recurrence and poly-time. In *Feasible Mathematics II*, pages 320–343. Birkhauser, 1994.

26 Daniel Leivant and Jean-Yves Marion. Lambda calculus characterizations of poly-time. *Fundam. Inform.*, 19(1/2), 1993.

27 Jean-Yves Marion and Jean-Yves Moyen. Efficient First Order Functional Program Interpreter with Time Bound Certifications. In *LPAR 2000*, volume 1955 of *LNAI*, pages 25–42. Springer, 2000.

28 Georg Moser and Andreas Schnabl. The derivational complexity induced by the dependency pair method. *Logical Methods in Computer Science*, 7(3), 2011.

29 Jaco van de Pol. *Termination of Higher-order Rewrite Systems.* PhD thesis, Utrecht University, 1996.

30 Toshiyuki Yamada. Confluence and termination of simply typed term rewriting systems. In *RTA 2001*, volume 2051 of *LNCS*, pages 338–352. Springer, 2001.