

Higher-order Interpretations and Program Complexity (Long Version)

Patrick Baillot* Ugo Dal Lago†

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 [1]. This fits in the area of implicit computational complexity, which aims at giving machine-free characterizations of complexity classes. Here we extend this approach to the higher-order setting. For that we consider the notion of simply typed term rewriting systems [2], we define higher-order polynomial interpretations (HOPI) for them and give a criterion based on HOPIs to ensure that a program can be executed in polynomial time. In order to obtain a criterion which is flexible enough to validate some 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 inferring *complexity* properties of programs early in development cycle, when the latter are still expressed in high-level programming 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 (λ -calculus, term rewriting systems, 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., are actually 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) [1, 3], whose complexity can be kept under control by way of variations of the powerful techniques developed to check termination of TRS, namely path orderings [4, 5], dependency pairs and the interpretation method [6]. 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 very powerful, in particular when the interpretation method is relaxed and coupled with recursive path orderings, like in quasi-interpretations [3].

The results cited above are very interesting and indeed represent the state-of-the art in resource analysis when programs are *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

*CNRS & ENS-Lyon, patrick.baillot@ens-lyon.fr

†Università di Bologna & INRIA, dallago@cs.unibo.it

Simone Martini [7]. There is strong evidence, however, that first-order programs obtained as results of one of this translations schemes are difficult to prove to have bounded complexity with traditional TRS techniques.

Another possibility consists in generalizing TRS techniques to systems of higher-order rewriting, which come in many different flavours [8, 9, 2]. The majority of the introduced higher-order generalization 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 modulo a reduction rule can in some cases be undecidable. Higher-order generalizations of TRS techniques [10], 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 (STTRSs in the following), we define a system of higher-order polynomial interpretations for them and prove that, following [1], 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 `foldl`, and that this problem can be overcome by switching to another technique, designed along the lines of quasi-interpretations [3]. This is the subject of sections 4 and 5 below.

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 lambda calculus and extensions of it with full recursion, like PCF. This can be proved following [7] and is the subject of Section 3.

2 Simply-Typed Term Rewriting Systems

2.1 Definitions and Notations

We recall here the definition of a STTRS, following [2, 11]. We will actually consider as *programs* a subclass of STTRSs, basically those where rules only deal with the particular case of 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* and we shall denote as D, E, \dots . *Types* are defined by the following grammar:

$$A, B ::= D \mid A_1 \times \dots \times A_n \rightarrow A.$$

A *functional type* is a type which contains an occurrence of \rightarrow . Some examples of base types are the type NAT of tally integers, and the type W_2 of binary words.

We denote by \mathcal{F} the set of *function symbols* (or just *functions*), \mathcal{C} that of *constructors* and \mathcal{X} that of *variables*. Constructors $\mathbf{c} \in \mathcal{C}$ have a type of the form $D_1 \times \dots \times D_n \rightarrow D$. Functions $\mathbf{f} \in \mathcal{F}$, 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:

$$t, t_i ::= x^A \mid \mathbf{c}^A \mid \mathbf{f}^A \mid (t^{A_1 \times \dots \times A_n \rightarrow A} t_1^{A_1} \dots t_n^{A_n})^A$$

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

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.

To simplify the writing of terms we will often elide their type. We will also write $(t \bar{s})$ for $(t s_1 \dots s_n)$. Therefore any term t is of the form $(\dots ((\alpha \bar{s}_1) \bar{s}_2) \dots \bar{s}_k)$ where $k \geq 0$, $\alpha \in \mathcal{X} \cup \mathcal{C} \cup \mathcal{F}$.

To simplify notation we will use the following convention: any term t is of the form $(\dots ((s \bar{s}_1) \bar{s}_2) \dots \bar{s}_k)$ will be written $((s \bar{s}_1 \dots \bar{s}_k))$. Observe however that, e.g., if t has type $A_1 \times A_2 \rightarrow (B_1 \times B_2 \rightarrow B)$,

t_i type A_i for $i = 1, 2$, s_i 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 $B_1 \times B_2 \rightarrow B$ and B , respectively), but $(t \ t_1)$ and $(t \ t_1 \ t_2 \ s_1)$ are not well-typed.

A crucial class of terms are patterns, which in particular are used in defining rewriting rules. Formally, a *pattern* is a term generated by the following grammar:

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

\mathcal{P} is the set of all patterns. Observe that if a pattern has a functional type then it must be a variable. We consider *rewriting rules* in the form $t \rightarrow s$ satisfying the following two constraints:

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 $((\mathbf{f} \ \overline{p_1} \dots \overline{p_k}))$ where each $\overline{p_i}$ for $i \in 1, \dots, k$ consists of patterns only. The rule is said to be a rule *defining* \mathbf{f} , while the total number of patterns in $\overline{p_1}, \dots, \overline{p_k}$ is the *arity* of the rule.

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

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 it has the form $(\mathbf{c} \ v_1 \dots v_n)$, where v_1, \dots, v_n are themselves values.
2. it has functional type A and is of the form $((\mathbf{f}, \overline{v_1} \dots \overline{v_n}))$, where the terms in $\overline{v_1}, \dots, \overline{v_n}$ are themselves values *and* the total number of terms in $\overline{v_1}, \dots, \overline{v_n}$ is strictly smaller than the arity of \mathbf{f} .

We denote values as v, u and their set by \mathcal{V} .

2.2 STTRSs: Dynamics

The dynamical process underlying a program is formalized through a rewriting relation. Preliminary to that are proper notions of unification and substitution, which are the subject of this section.

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{V} to \mathcal{T} , that we shall also write σ . The image of a term t will be denoted $t\sigma$. *Contexts* are defined as terms but with the proviso that they must contain exactly one occurrence of a special constant \bullet^A (*hole*), for a type A . They are denoted as $\mathcal{C}, \mathcal{D} \dots$. 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 , denoted as $s \rightarrow_R t$, if there exists a rule $l \rightarrow r$ of R , a context \mathcal{C} and a substitution σ such that $s = \mathcal{C}\{l\sigma\}$ and $t = \mathcal{C}\{r\sigma\}$. When there is no ambiguity on R we will simply write \rightarrow instead of \rightarrow_R .

Please notice that one of the advantages of STTRSs over similar formalisms (like [8]) 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. In the next section, however, we show how STTRSs are expressive enough to capture standard typed λ -calculi.

3 Typed λ -calculi as STTRSs

The goal of this Section is to illustrate the fact that the choice of the STTRS framework as higher-order calculus is *not* too restrictive: indeed we will show that we can simulate in it PCF equipped with *weak reduction* (i.e. where one does not reduce in the scope of abstractions). This is achieved using ideas developed for encodings of the λ -calculus into first-order rewrite systems [7].

3.1 A Few Words About PCF

We assume a total order \leq on \mathcal{X} . *PCF types* are defined as follows:

$$A, B ::= \text{NAT} \mid A \rightarrow A.$$

PCF terms, on the other hand, are defined as follows:

$$\begin{aligned} M, N ::= & x^A \mid (\lambda x^A. M^B)^{A \rightarrow B} \mid (M^{A \rightarrow B} N^A)^B \mid \mathbf{fix}^{((A \rightarrow B) \rightarrow A \rightarrow B) \rightarrow A \rightarrow B} \mid \mathbf{n}^{\text{NAT}} \mid \\ & \mathbf{succ}^{\text{NAT} \rightarrow \text{NAT}} \mid \mathbf{pred}^{\text{NAT} \rightarrow \text{NAT}} \mid (\mathbf{ifz} M^A N^A)^{\text{NAT} \rightarrow A} \end{aligned}$$

where n ranges over the natural numbers. We omit types in terms whenever this does not cause ambiguity. A *PCF value* is any term different from an application. PCF values are indicated with metavariables like V and W . A call-by-value operational semantics for PCF can be expressed by way of some standard reduction rules:

$$\begin{aligned} (\lambda x. M)V &\longrightarrow M\{V/x\} \\ \mathbf{fix} V &\longrightarrow V(\lambda x. (\mathbf{fix} V)x) \\ \mathbf{succ} \mathbf{n} &\longrightarrow \mathbf{n} + \mathbf{1} \\ \mathbf{pred} \mathbf{0} &\longrightarrow \mathbf{0} \\ \mathbf{pred} \mathbf{n} + \mathbf{1} &\longrightarrow \mathbf{n} \\ (\mathbf{ifz} M N) \mathbf{0} &\longrightarrow M \\ (\mathbf{ifz} M N) \mathbf{n} + \mathbf{1} &\longrightarrow N \end{aligned}$$

The reduction rules above can be propagated to any applicative context by the rules below

$$\frac{M \longrightarrow N}{ML \longrightarrow NL} \quad \frac{M \longrightarrow N}{LM \longrightarrow LN}$$

3.2 PCF as a STTRS

PCF can be turned into a STTRS R_{PCF} with infinitely many function symbols. First, for each term M of type B , with free variables $x_1 \leq x_2 \leq \dots \leq x_n$ of types A_1, \dots, A_n , and x of type A , we introduce a function $\mathbf{abs}_{M,x}$ of \mathcal{F} , with type $A_1 \times \dots \times A_n \rightarrow (A \rightarrow B)$. Then, for each pair of terms M, N of type B , both with free variables among $x_1 \leq x_2 \leq \dots \leq x_n$ of types A_1, \dots, A_n , and x of type A , we introduce a function $\mathbf{ifz}_{M,N}$ of \mathcal{F} , with type $A_1 \times \dots \times A_n \rightarrow (\text{NAT} \rightarrow B)$. We also need function symbols for \mathbf{succ} , \mathbf{pred} and \mathbf{fix} . Now, the translation $\langle M \rangle$ is defined by induction on M :

$$\begin{aligned} \langle x \rangle &= x; \\ \langle M N \rangle &= (\langle M \rangle \langle N \rangle); \\ \langle \lambda x. M \rangle &= (\mathbf{abs}_{M,x} x_1 \dots x_n), \text{ if } FV(M) = x_1 \leq \dots \leq x_n; \\ \langle \mathbf{fix}^{((A \rightarrow B) \rightarrow A \rightarrow B) \rightarrow A \rightarrow B} \rangle &= \mathbf{fix}^{((A \rightarrow B) \rightarrow A \rightarrow B) \rightarrow A \rightarrow B}; \\ \langle \mathbf{n} \rangle &= \underbrace{\mathbf{s}(\mathbf{s}(\dots(\mathbf{s}(\mathbf{0})))\dots)}_{n \text{ times}} \\ \langle \mathbf{succ} \rangle &= \mathbf{succ}; \\ \langle \mathbf{pred} \rangle &= \mathbf{pred}; \\ \langle \mathbf{ifz} M N \rangle &= (\mathbf{ifz}_{M,N} x_1 \dots x_n), \text{ if } FV(M) \cup FV(N) = x_1 \leq \dots \leq x_n. \end{aligned}$$

A converse translation $[\cdot]$ can be easily defined.

Lemma 1 *If M is a PCF term of type A , then $\langle M \rangle$ is a well-typed term, of type A .*

The rules of R_{PCF} are the following:

$$\begin{aligned} & ((\mathbf{abs}_{M,x} x_1 \dots x_n) x) \rightarrow \langle M \rangle; \\ & (\mathbf{fix} x) \rightarrow (x (\mathbf{abs}_{yz,z} (\mathbf{fix} x))); \\ & (\mathbf{succ} x) \rightarrow (\mathbf{s} x); \\ & (\mathbf{pred} \mathbf{0}) \rightarrow \mathbf{0}; \\ & (\mathbf{pred} (\mathbf{s} x)) \rightarrow x; \\ & ((\mathbf{ifz}_{M,N} x_1 \dots x_n) \mathbf{0}) \rightarrow \langle M \rangle; \\ & ((\mathbf{ifz}_{M,N} x_1 \dots x_n) (\mathbf{s} x)) \rightarrow \langle N \rangle. \end{aligned}$$

A term is said to be *canonical* if either it is a value or it is in the form $((\mathbf{f} t_1 \dots t_n s_1 \dots s_m))$, where n is the arity of \mathbf{f} , t_1, \dots, t_n are values and s_1, \dots, s_m are themselves canonical. The following are technical intermediate results towards Theorem 7

Lemma 2 *For every PCF term M , $[\langle M \rangle] = M$.*

Proof : By induction on the structure of M . □

Lemma 3 *For every closed PCF term M , $\langle M \rangle$ is canonical. Moreover, if t is canonical and $t \rightarrow s$, then s is canonical.*

Proof : The fact $\langle M \rangle$ is always canonical can be proved by induction on the structure of M . The fact canonicity is preserved by reduction is a consequence of the adoption of a call-by-value notion of reduction. □

Lemma 4 *A canonical term t is a normal form iff $[t]$ is a normal form.*

Proof : Again, a simple induction of t . □

Lemma 5 *If t is canonical and $t \rightarrow s$, then $[t] \rightarrow [s]$.*

Lemma 6 *If $M \rightarrow N$, t is canonical and $[t] = M$, then $t \rightarrow s$, where $[s] = N$.*

Theorem 7 (Term Reducibility) *Let M be a closed PCF term. The following two conditions are equivalent:*

1. $M \rightarrow^* N$ where N is in normal form;
2. $\langle M \rangle \rightarrow^* t$ where $[t] = N$ and t is in normal form.

Proof : Suppose $M \rightarrow^n N$, where N is in normal form. Then, by applying Lemma 6, we obtain a term t such that $\langle M \rangle \rightarrow^n t$ and $[t] = N$. By Lemma 3, t is canonical and, by Lemma 4, it is in normal form. Now, suppose $\langle M \rangle \rightarrow^n t$ where $[t] = N$ and t is in normal form. By applying n times Lemma 5, we obtain $[\langle M \rangle] \rightarrow^n [t] = N$. But $[\langle M \rangle] = M$ by Lemma 2 and N is a normal form by Lemma 4, since $\langle M \rangle$ and t are canonical by Lemma 3. □

Even if the STTRS we have just defined is infinite and involves infinitely many function symbols, one can prove that any closed PCF program M of type $\text{NAT} \rightarrow \text{NAT}$ only needs a finite set of function symbols to be simulated, namely those function symbols corresponding to subterms of M .

In the following it will also be instructive to consider a source language which is less expressive than PCF: Gödel's T, equipped with weak reduction. For that, remove in the source language the constant \mathbf{fix} , and replace it with a constant \mathbf{rec} with type scheme $A \rightarrow (\text{NAT} \rightarrow A \rightarrow A) \rightarrow \text{NAT} \rightarrow A$ and the following reduction rules:

$$\begin{aligned} \mathbf{rec} x f \mathbf{0} & \rightarrow x; \\ \mathbf{rec} x f \mathbf{n} + \mathbf{1} & \rightarrow (f \mathbf{n} (\mathbf{rec} x f \mathbf{n})). \end{aligned}$$

We translate it to a STTRS R_{\top} similar to R_{PCF} : one adds a function symbol $\text{rec}^{A \rightarrow (\text{NAT} \rightarrow A \rightarrow A) \rightarrow \text{NAT} \rightarrow A}$ and the following new STTRS rules:

$$\begin{aligned} & (((\text{rec } x) f) \mathbf{0}) \rightarrow x; \\ & (((\text{rec } x) f) (\mathbf{s } y)) \rightarrow ((f y) (((\text{rec } x) f) y)). \end{aligned}$$

This encoding of system \top then enjoys properties similar to the encoding of PCF.

4 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 a complexity bound on programs of *first-order types* but using 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 `foldl`, 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 [1], which offers the advantage of simplicity. We thus build a theory of *higher-order polynomial interpretations* for STTRSs. It can be seen as a particular concrete instantiation of the methodology proposed in [2] for proving termination by interpretation.

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 $\mathbb{F}\text{SPOS}$ with products and functions. So, the whole process can be summarized by the following diagram:

$$\text{STTRSs} \xrightarrow{[\cdot]} \text{HOPs} \xrightarrow{[\cdot]} \mathbb{F}\text{SPOS}$$

4.1 Higher-Order Polynomials

Let us consider types built over a single base type \mathbf{N} :

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

$A^n \rightarrow B$ stands for the type

$$\underbrace{A \rightarrow \dots \rightarrow A}_n \rightarrow B.$$

Let C_P be the following set of constants:

$$C_P = \{+ : \mathbf{N}^2 \rightarrow \mathbf{N}, \times : \mathbf{N}^2 \rightarrow \mathbf{N}\} \cup \{\bar{n} : \mathbf{N} \mid n \in \mathbb{N}^*\}.$$

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

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

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 . *HOP contexts* (or simply contexts) are defined as HOPs but with the proviso that they must contain exactly one occurrence of a special constant \bullet^A (*hole*), for a type A . They are denoted as $\mathcal{C}, \mathcal{D} \dots$ If \mathcal{C} is a HOP context with hole \bullet^A , and M is a HOP of type A , then $\mathcal{C}\{M\}$ is the HOP obtained from \mathcal{C} by replacing the occurrence of \bullet^A by M and reducing to the β normal form. We assume given the usual set-theoretic interpretation of types and terms, denoted as $\llbracket A \rrbracket$ and $\llbracket M \rrbracket$: if M has type A and $FV(M) = \{x_1^{A_1}, \dots, x_n^{A_n}\}$, then $\llbracket M \rrbracket$ is a map from $\llbracket A_1 \rrbracket \times \dots \times \llbracket A_n \rrbracket$ to $\llbracket A \rrbracket$. We denote by \equiv the equivalence relation which identifies terms which denote the same function, *e.g.* we have: $\lambda x. (\bar{2} \times ((\bar{3} + x) + y)) \equiv \lambda x. (\bar{6} + (2 \times x + 2 \times y))$.

Noticeably, even if HOPs can be built using higher-order functions, the first order fragment only contains polynomials:

Lemma 8 *If M is a HOP of type $\mathbf{N}^n \rightarrow \mathbf{N}$ and such that $FV(M) = \{y_1 : \mathbf{N}, \dots, y_k : \mathbf{N}\}$, then the function $\llbracket M \rrbracket$ is a polynomial function.*

Proof : We proceed by induction on M :

- if $M = x$ or $M = +, \times$ or \bar{n} : the result is trivial;
- if $M = \lambda x^A.M_1^B$: then we have $A = N$ and the type B is of the form $N, \dots, N \rightarrow N$, so by i.h. on M_1 the property is true for M_1 , hence for M ;
- otherwise M is an application. As M is in β -normal form, there exists $n \geq 0$ such that:
 - M is of the form $M = (N M_1 \dots M_n)$,
 - and $N = x^A, +, \times$ or \bar{n} .

Now, if $N = +$ or \times , then for any $1 \leq i \leq n$ we have that M_i is of type N , so by i.h. on M_i it satisfies the property, therefore M represents a polynomial function. If $N = x^A$, then as x^A is free in M , by assumption we know that $A = N$, thus $n = 0$ and the property is valid. Similarly if $N = \bar{n}$.

This concludes the proof. \square

A *HOP substitution* θ is a map from variables to HOPs, with finite domain. We will simply speak of *substitution* if there is no ambiguity. For any HOP M and HOP substitution θ , $M\theta$ is the HOP defined in the expected way.

4.2 Semantic Interpretation.

Now, we consider a subcategory \mathbf{FSPOs} of the category \mathbf{SPOs} of strict partial orders as objects and *strictly monotonic* total functions as morphisms. Objects of \mathbf{FSPOs} are the following:

- \mathcal{N} is the domain of *strictly positive* integers, equipped with the natural strict order $\prec_{\mathcal{N}}$,
- 1 is the trivial order with one point;
- if σ, τ are objects, then $\sigma \times \tau$ is obtained by the product ordering,
- $\sigma \rightarrow \tau$ is the set of strictly monotonic total functions from σ to τ , equipped with the following strict order:

$$f \prec_{\sigma \rightarrow \tau} g \text{ if for any } a \text{ of } \sigma \text{ we have } f(a) \prec_{\tau} g(a).$$

We denote by \preceq_{τ} the reflexive closure of \prec_{τ} . \mathbf{FSPOs} is a subcategory of \mathbf{SET} with all the necessary structure to interpret types. $\llbracket A \rrbracket_{\prec}$ denotes the semantics of A as an object of \mathbf{FSPOs} . We choose to set $\llbracket \mathbf{N} \rrbracket_{\prec} = \mathcal{N}$. Notice that any element of $e \in \llbracket A \rrbracket_{\prec}$ can be easily mapped onto an element $e \downarrow$ of $\llbracket A \rrbracket$. What about terms? Actually, \mathbf{FSPOs} can again be shown to be sufficiently rich:

Proposition 9 *Let M be a HOP of type A with free variables $x_1^{A_1}, \dots, x_n^{A_n}$. Then for every $e \in \llbracket A_1 \times \dots \times A_n \rrbracket_{\prec}$, there is exactly one $f \in \llbracket A \rrbracket_{\prec}$ such that $f \downarrow = \llbracket M \rrbracket(e \downarrow)$. Moreover, this correspondence is strictly monotone and thus defines an element of $\llbracket A_1 \times \dots \times A_n \rightarrow A \rrbracket_{\prec}$ which we denote as $\llbracket M \rrbracket_{\prec}$.*

Proof : We proceed by induction on the construction of M :

- $M = x^A$: trivial;
- $M = c^A \in C_P$: this holds because as we have interpreted N as \mathbf{N}^* the terms $+$ and \times denote strictly monotonic functions (note that it would not have been the case for \times if we had interpreted N as \mathbf{N});
- $M = \lambda x.M'$: by definition we know that x is free in M' . Let us denote by $x_1^{A_1}, \dots, x_n^{A_n}$ the free variables of M' and w.l.o.g. assume $x = x_n$. By i.h. we have $\llbracket M' \rrbracket_{\prec} \in \llbracket A_1 \times \dots \times A_n \rightarrow A \rrbracket_{\prec}$. For (e_1, \dots, e_{n-1}) in $\llbracket A_1 \times \dots \times A_{n-1} \rrbracket_{\prec}$ we then consider the map f from $\llbracket A_n \rrbracket_{\prec}$ to $\llbracket A \rrbracket_{\prec}$ defined by $f(e_n) = \llbracket M' \rrbracket_{\prec}(e_1, \dots, e_n)$. As $\llbracket M' \rrbracket_{\prec} \in \llbracket A_1 \times \dots \times A_n \rightarrow A \rrbracket_{\prec}$ we have that $f \in \llbracket A_n \rightarrow A \rrbracket_{\prec}$ and satisfies the property. Suppose now that $f' \in \llbracket A_n \rightarrow A \rrbracket_{\prec}$ is defined in a similar way from $(e'_1, \dots, e'_{n-1}) \in \llbracket A_1 \times \dots \times A_{n-1} \rrbracket_{\prec}$ such that $(e_1, \dots, e_{n-1}) \prec (e'_1, \dots, e'_{n-1})$. Then we have that if $e_n \in \llbracket A_n \rrbracket_{\prec}$, then $(e_1, \dots, e_{n-1}, e_n) \prec (e'_1, \dots, e'_{n-1}, e_n)$, thus $\llbracket M' \rrbracket_{\prec}(e_1, \dots, e_{n-1}, e_n) \prec \llbracket M' \rrbracket_{\prec}(e'_1, \dots, e'_{n-1}, e_n)$. This shows that $f \prec f'$. Therefore we have obtained a strictly monotonic correspondence from $\llbracket A_1 \times \dots \times A_{n-1} \rrbracket_{\prec}$ to $\llbracket A_n \rightarrow A \rrbracket_{\prec}$, which completes the claim.

- $M = M_1^{A \rightarrow B} M_2^A$: this is the crucial case. By i.h. $\llbracket M_1 \rrbracket_{\prec}$ and $\llbracket M_2 \rrbracket_{\prec}$ have been defined. Denote by $x_1^{A_1}, \dots, x_n^{A_n}$ the free variables of M . Take $e = (e_1, \dots, e_n) \in \llbracket A_1 \times \dots \times A_n \rrbracket_{\prec}$. By abuse of notation we will simply write $\llbracket M_1 \rrbracket_{\prec}(e)$ instead of $\llbracket M_1 \rrbracket_{\prec}(e_{i_1}, \dots, e_{i_k})$ where x_{i_1}, \dots, x_{i_k} are the free variables of M_1 . Similarly for $\llbracket M_2 \rrbracket_{\prec}(e)$. Now we define f as $f = \llbracket M_1 \rrbracket_{\prec}(e)(\llbracket M_2 \rrbracket_{\prec}(e))$. We have that $f \in \llbracket B \rrbracket_{\prec}$ and $f \downarrow = \llbracket M \rrbracket_{\prec}(e \downarrow)$. Now let us show that this correspondence is strictly monotone. If $n = 0$ it is trivial, so let us assume $n \geq 1$. Take e, e' two elements of $\llbracket A_1 \times \dots \times A_n \rrbracket_{\prec}$ with $e \prec e'$. Let $f = \llbracket M_1 \rrbracket_{\prec}(e)(\llbracket M_2 \rrbracket_{\prec}(e))$ and $f' = \llbracket M_1 \rrbracket_{\prec}(e')(\llbracket M_2 \rrbracket_{\prec}(e'))$. Let us distinguish two subcases:

- if there exists i in $\{1, \dots, n\}$ such that $e_i \prec e'_i$ and $x_i \in FV(M_1)$:
then by i.h. on M_1 we have $\llbracket M_1 \rrbracket_{\prec}(e) \prec_{A \rightarrow B} \llbracket M_1 \rrbracket_{\prec}(e')$;
moreover by h.i. we have:
 $\llbracket M_2 \rrbracket_{\prec}(e) \preceq_A \llbracket M_2 \rrbracket_{\prec}(e')$ (note the non-strict ordering).
From these two inequalities, by definition of $\prec_{A \rightarrow B}$ we can deduce:
 $\llbracket M_1 \rrbracket_{\prec}(e)(\llbracket M_2 \rrbracket_{\prec}(e)) \prec_B \llbracket M_1 \rrbracket_{\prec}(e')(\llbracket M_2 \rrbracket_{\prec}(e'))$.
- Otherwise: we know that there exists i in $\{1, \dots, n\}$ such that $e_i \prec e'_i$, and x_i is free in M so it must be free in M_2 . So by i.h. on M_2 we have $\llbracket M_2 \rrbracket_{\prec}(e) \prec_A \llbracket M_2 \rrbracket_{\prec}(e')$; besides we know that $\llbracket M_1 \rrbracket_{\prec}(e) = \llbracket M_1 \rrbracket_{\prec}(e')$.
We know that $\llbracket M_1 \rrbracket_{\prec}(e)$ belongs to $\llbracket A \rightarrow B \rrbracket_{\prec}$ so it is strictly monotonic, so we deduce that:
 $\llbracket M_1 \rrbracket_{\prec}(e)(\llbracket M_2 \rrbracket_{\prec}(e)) \prec_B \llbracket M_1 \rrbracket_{\prec}(e')(\llbracket M_2 \rrbracket_{\prec}(e'))$.

So in both subcases we have concluded that $f \prec f'$, which completes the proof of the claim.

This concludes the proof. \square

Lemma 10 *Let M be a HOP and θ a HOP substitution with $FV(M) = \{x_1^{A_1}, \dots, x_n^{A_n}\}$ and $FV(M\theta) = \{y_1^{B_1}, \dots, y_m^{B_m}\}$. Then, for every $i \in \{1, \dots, m\}$ the function $f_i = \llbracket \theta(x_i) \rrbracket_{\prec}$ is a strictly monotonic map. Moreover, we have $\llbracket M\theta \rrbracket_{\prec} = \llbracket M \rrbracket_{\prec} \circ (f_1, \dots, f_m)$.*

Proof: For $i \in \{1, \dots, m\}$ we have $f_i = \llbracket \theta(x_i) \rrbracket_{\prec}$. We know that $\theta(x_i)$ is a HOP, and $FV(\theta(x_i)) \subseteq FV(M\theta)$. Let $\{y_{k_1}^{B_{k_1}}, \dots, y_{k_i}^{B_{k_i}}\}$ be the free variables of $\theta(x_i)$. Then f_i belongs to $\llbracket B_{k_1} \rrbracket_{\prec} \times \dots \times \llbracket B_{k_i} \rrbracket_{\prec} \rightarrow \llbracket A_i \rrbracket_{\prec}$. We can then prove the second statement by induction on M . As an illustration let us just examine here the base cases:

- The case where $M = c^A$ is trivial because $n = 0$ and $M\theta = M$.
- In the case where M is a variable we have $n = 1$ and $M = x_1^{A_1}$. Then $\llbracket M\theta \rrbracket_{\prec} = \llbracket \theta(x_1) \rrbracket_{\prec} = f_1 = id_{A_1} \circ f_1$.

This concludes the proof. \square

Applying the same substitution to two HOPs having the same type preserve the properties of the underlying interpretation:

Lemma 11 *If M^A, N^A are HOPs such that $FV(M) \subseteq FV(N)$, θ is a HOP substitution and if $\llbracket M \rrbracket_{\prec} \prec \llbracket N \rrbracket_{\prec}$, then $\llbracket M\theta \rrbracket_{\prec} \prec \llbracket N\theta \rrbracket_{\prec}$.*

4.3 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 $f \in \mathcal{C} \cup \mathcal{F}$, $[f]$ is a closed HOP, of type $\underline{A}_1, \dots, \underline{A}_n \rightarrow \underline{A}$. Now, for $t \in \mathcal{T}$ we define $[t]$ by induction on t :

- if $t \in \mathcal{X}$, then $[t]$ is f ;
- 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.

Lemma 12 *Let $t \in \mathcal{T}$ of type A and $FV(t) = \{y_1 : A_1, \dots, y_n : A_n\}$, then: $[t]$ is a HOP, of type \underline{A} , and $FV([t]) = \{y_1 : \underline{A}_1, \dots, y_n : \underline{A}_n\}$.*

Proof : This follows from the definition of $[t]$, by induction on t . \square

We extend the notion of assignments to contexts by setting: $[\bullet^A] = \bullet^{\underline{A}}$. So if \mathcal{C} is a context then $[\mathcal{C}]$ is a HOP context, and we have:

Lemma 13 *If \mathcal{C} is a context and t a term, then $[\mathcal{C}\{t\}] = [\mathcal{C}]\{[t]\}$.*

Now, if σ is a substitution, $[\sigma]$ is the HOP substitution defined as follows: for any variable x , $[\sigma](\underline{x}) = [\sigma(x)]$. We have:

Lemma 14 *If t is a term and σ a substitution, then: $[t\sigma] = [t]\theta$, where $\theta = [\sigma]$.*

Let us now consider the semantic interpretation. If $t \in \mathcal{T}$ of type A and $FV(t) = \{y_1 : A_1, \dots, y_n : A_n\}$, we will simply denote by $\llbracket t \rrbracket_{\prec}$ the element $\llbracket [t] \rrbracket_{\prec}$ of $\llbracket \underline{A}_1 \times \dots \times \underline{A}_n \rightarrow \underline{A} \rrbracket_{\prec}$.

Lemma 15 *Let M, N be HOPs such that $\llbracket M \rrbracket_{\prec} \prec \llbracket N \rrbracket_{\prec}$ and \mathcal{C} be a HOP context such that $\mathcal{C}\{M\}$ and $\mathcal{C}\{N\}$ are well-defined. Then we have that $\llbracket \mathcal{C}\{M\} \rrbracket_{\prec} \prec \llbracket \mathcal{C}\{N\} \rrbracket_{\prec}$. The same statement holds if we replace HOPs M, N by terms t, s and the HOP context by a context.*

Now, we say that an assignment $[\cdot]$ is a *higher polynomial interpretation (HOPI)* or simply a *polynomial interpretation* for R iff for any $l \rightarrow r \in R$, we have that $\llbracket r \rrbracket_{\prec} \prec \llbracket l \rrbracket_{\prec}$. Note that in the particular case where the program only contains first-order functions, this notion of higher-order polynomial interpretation coincides with the classical one for first-order rewrite systems. In the following we assume that $[\cdot]$ is a polynomial interpretation for the program P . A key property is the following, which tells you that the interpretation of terms strictly decreases along any reduction step:

Lemma 16 *If $s \rightarrow t$, then $\llbracket t \rrbracket_{\prec} \prec \llbracket s \rrbracket_{\prec}$.*

Proof : Suppose $s \rightarrow t$, then by definition there exists a rule $l \rightarrow r$ of R , a context \mathcal{C} and a substitution σ such that $s = \mathcal{C}\{l\sigma\}$ and $t = \mathcal{C}\{r\sigma\}$. As $[\cdot]$ is a polynomial interpretation we have $\llbracket r \rrbracket_{\prec} \prec \llbracket l \rrbracket_{\prec}$. We then get:

$$\begin{aligned} \llbracket r\sigma \rrbracket_{\prec} &= \llbracket [r\sigma] \rrbracket_{\prec}, && \text{by definition,} \\ &= \llbracket [r][\sigma] \rrbracket_{\prec}, && \text{by Lemma 14,} \\ &\prec \llbracket [l][\sigma] \rrbracket_{\prec}, && \text{by Lemma 11,} \\ &= \llbracket [l\sigma] \rrbracket_{\prec}, && \text{by Lemma 14 again,} \\ &= \llbracket [l\sigma] \rrbracket_{\prec}. \end{aligned}$$

Then by Lemma 15 (second statement) we get:

$$\llbracket \mathcal{C}\{r\sigma\} \rrbracket_{\prec} \prec \llbracket \mathcal{C}\{l\sigma\} \rrbracket_{\prec},$$

which ends the proof. \square

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

Proposition 17 *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 $\llbracket [t] \rrbracket_{\prec}$.*

Proof : It is sufficient to observe that by Lemma 16 any reduction step on t makes $\llbracket [t] \rrbracket_{\prec}$ decrease for \prec , and that as t is closed and of type \mathbf{N} the order \prec here is $\prec_{\mathcal{N}}$, which is the ordinary (strict) order on integers. \square

4.4 A Complexity Criterion

Proving a STTRS to have an interpretation is not enough to guarantee its time complexity to be polynomial. 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 $D_1 \times \dots \times D_n \rightarrow D_0$, where $n \geq 0$, is interpreted by a HOP $M_{\mathbf{c}}$ whose semantic interpretation $\llbracket M_{\mathbf{c}} \rrbracket_{\prec}$ is a polynomial function of the form:

$$P(y_1, \dots, y_n) = \sum_{i=1}^n y_i + \gamma_{\mathbf{c}}, \text{ with } \gamma_{\mathbf{c}} \geq 1.$$

Additivity ensures that the interpretation of first-order values is proportional to their size:

Lemma 18 *Let $[\cdot]$ be an additive assignment. Then there exists $\gamma \geq 1$ such that for any value v of type D , where D is a data type, we have $\llbracket v \rrbracket_{\prec} \leq \gamma \cdot |v|$.*

The base type W_n denotes the data-type of n -ary words, whose constructors are **empty** and $\mathbf{c}_1, \dots, \mathbf{c}_n$. A function $f : (\{0, 1\}^*)^m \rightarrow \{0, 1\}$ is said to be *representable* by a STTRS R if there is a function symbol \mathbf{f} of type $W_2^m \rightarrow W_2$ in R which computes f in the obvious way.

We are now ready to prove the main result about polynomial interpretations, namely that they enforce reduction lengths to be bounded in an appropriate way:

Theorem 19 *Let R be a STTRS with an additive polynomial interpretation $[\cdot]$. Consider \mathbf{g} a function symbol of type $W_2 \times \dots \times W_2 \rightarrow W_2$. We have: there exists a polynomial P such that, for any $w_1, \dots, w_n \in \{0, 1\}^*$, any reduction of $(\mathbf{g} \ \underline{w}_1 \dots \underline{w}_n)$ has length bounded by $P(|w_1|, \dots, |w_n|)$. This holds more generally for \mathbf{g} of type $D_1, \dots, D_n \rightarrow D$.*

Proof: By Prop. 17 we know that any reduction sequence has length bounded by: $\llbracket (\mathbf{g} \ \underline{w}_1 \dots \underline{w}_n) \rrbracket_{\prec} = \llbracket \mathbf{g} \rrbracket_{\prec} (\llbracket \underline{w}_1 \rrbracket_{\prec}, \dots, \llbracket \underline{w}_n \rrbracket_{\prec})$. By Lemma 8 there exists a polynomial function Q such that $\llbracket \mathbf{g} \rrbracket_{\prec}$ computes Q . Moreover by Lemma 18 there exists $\gamma \geq 1$ such that: $\llbracket \underline{w}_i \rrbracket_{\prec} \leq \gamma |w_i|$. So by defining P as the polynomial function such that $P(y_1, \dots, y_n) = Q(\gamma y_1, \dots, \gamma y_n)$, we have that the length of the reduction sequence is bounded by $P(|w_1|, \dots, |w_n|)$. \square

Corollary 20 *The functions on binary words representable by STTRSs admitting an additive polynomial interpretation are exactly the polytime functions.*

Proof : We have two inclusions to prove: from left to right (*complexity soundness*), and from right to left (*completeness*):

- **Complexity soundness.** Assume F is represented by a program \mathbf{g} with type $W_2 \times \dots \times W_2 \rightarrow W_2$ admitting an additive polynomial interpretation. Then by Theorem 19 we know that for any $w_1, \dots, w_n \in \{0, 1\}^*$, any reduction of $(\mathbf{g} \ \underline{w}_1 \dots \underline{w}_n)$ has a polynomial number of steps. By a result in [12], derivational complexity is an invariant cost model for TRSs, via graph rewriting. This result can be easily generalized to STTRSs.
- **Completeness.** Let F be a polytime function on binary words. It has been shown in [1] (Theorem 4, Sect. 4.2) that there exists a first-order rewrite system with an additive polynomial interpretation which computes F . Actually the rewrite systems considered in this paper are not assumed to be typed, but it can be checked that the rewrite systems used to simulate polynomial time Turing machines are typable and can be seen as simply typed term rewrite systems in our sense. Moreover when restricting to first-order typed rewrite systems, our notions of polynomial interpretation and of additive polynomial interpretation coincide with the notions they consider. Therefore F can be represented by a simply typed term rewriting system program with an additive polynomial interpretation.

This concludes the proof. \square

4.5 Examples

Consider the STTRS defined by the following rules:

$$\begin{aligned} (1) \quad & (\mathbf{map} \ f \ \mathbf{nil}) && \rightarrow \ \mathbf{nil} \\ (2) \quad & (\mathbf{map} \ f \ (\mathbf{cons} \ x \ xs)) && \rightarrow \ (\mathbf{cons} \ (f \ x) \ (\mathbf{map} \ f \ xs)) \end{aligned}$$

with the following types:

$$\begin{aligned} f &: D_1 \rightarrow D_2, & \mathbf{map} &: (D_1 \rightarrow D_2) \times L(D_1) \rightarrow L(D_2), \\ \mathbf{nil} &: L(D_i), & \mathbf{cons} &: D_i \times L(D_i) \rightarrow L(D_i) \quad \text{for } i = 1, 2. \end{aligned}$$

Here D_i and $L(D_i)$ for $i = 1, 2$ are base types. For simplicity we use just one **cons** and one **nil** notation for both types D_1, D_2 .

The interpretation below was given in [2] for proving termination, but here we show that it also gives a polynomial time bound. To simplify the reading of HOPs we use infix notations for $+$, omit some brackets (because anyway we have associativity and commutativity for the denotations) and the symbol \times , write $F(n)$ for the application of a function and k instead of \bar{k} . Now, we choose the following assignment of HOPs:

$$\begin{aligned} [\mathbf{nil}] &= 2 & : \ \mathbf{N} \\ [\mathbf{cons}] &= \lambda n. \lambda m. (n + m + 1) & : \ \mathbf{N} \rightarrow \mathbf{N} \rightarrow \mathbf{N} \\ [\mathbf{map}] &= \lambda F. \lambda n. nF(n) & : \ (\mathbf{N} \rightarrow \mathbf{N}) \rightarrow \mathbf{N} \rightarrow \mathbf{N} \end{aligned}$$

Instead of the last line we will often write $[\mathbf{map}](F, n) = nF(n)$.

We use as notations $[f] = F$, $[x] = n$, $[xs] = m$. We then get the following interpretations of terms:

$$\begin{aligned} [\mathbf{map} \ f \ \mathbf{nil}] &= 2F(2) \\ [\mathbf{map} \ f \ \mathbf{cons}(x, xs)] &= (n + m + 1)F(n + m + 1) \\ [\mathbf{cons}(f(x), (\mathbf{map} \ f \ xs))] &= 1 + F(n) + mF(m) \end{aligned}$$

One can check that the condition $\llbracket r \rrbracket_{\prec} \prec \llbracket l \rrbracket_{\prec}$ holds for (1) and (2). We thus have an additive polynomial interpretation for **map**, therefore Corollary 20 applies and we can conclude that for any f also satisfying the criterion, $(\mathbf{map} \ f)$ computes a polynomial time function.

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

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 obtained. In the sequel we will see how to overcome this issue.

5 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 rôle played by polynomial interpretations [13, 3]: (i) a termination condition, (ii) a condition enforcing a bound on the size of values occurring during the computation. In [3], 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.

$$\begin{array}{c}
\frac{\mathbf{f}^A \in \mathcal{NF}}{\Gamma \mid \Delta \vdash \mathbf{f} : A} \quad \frac{\mathbf{c}^A \in \mathcal{C}}{\Gamma \mid \Delta \vdash \mathbf{c} : A} \quad \frac{}{\Gamma \mid x : A, \Delta \vdash x : A} \quad \frac{}{x : D, \Gamma \mid \Delta \vdash x : D} \\
\\
\frac{\mathbf{f}^{A_1, \dots, A_n \rightarrow B} \in \mathcal{RF}, \text{ with arity } n \quad \Gamma \mid \emptyset \vdash s_i : A_i}{\Gamma \mid \Delta \vdash ((\mathbf{f} \ s_1 \dots s_n)) : B} \quad \frac{\Gamma \mid \Delta \vdash t : A_1 \times \dots \times A_n \rightarrow B \quad \Gamma \mid \Delta_i \vdash s_i : A_i}{\Gamma \mid \Delta, \Delta_1, \dots, \Delta_n \vdash (t \ s_1 \dots s_n) : B}
\end{array}$$

Figure 1: A Linear Type System for STTRS terms.

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 5.1 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 5.2 and 5.3.

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

5.1 The Termination Criterion

The termination criterion has two ingredients: a typing ingredient and a syntactic ingredient, expressed using the order \sqsubset . Is it restrictive for expressivity? Let us comment first on the syntactic ingredient:

- First consider the embedding of simply typed λ -calculus given by the restriction of the embedding of PCF of Section 3. The function symbols used are $\text{abs}_{M,x}$ for all typed term M and variable x . We define the order \sqsubset by:

$$\text{abs}_{M,x} \sqsubset \text{abs}_{N,y} \text{ if } \lambda y.N \text{ is a subterm of } M.$$

We write $\mathbf{f} \sqsubset t$ if $\mathbf{f} \sqsubset \mathbf{g}$ for any function \mathbf{g} in t . The STTRS encoding of M then satisfies the following condition, for any rule $((\mathbf{f} \ \overline{p}_1 \dots \overline{p}_k)) \rightarrow s$:

$$s \sqsubset \mathbf{f} \tag{1}$$

As the order \sqsubset defined is well-founded, this implies the termination of this STTRS program.

- Now consider System \mathbb{T} . We have seen that System \mathbb{T} with weak reduction can also be embedded into a STTRS. Forget about functions pred , succ , ifz for simplification, and consider the new function rec . We extend \sqsubset by setting: $\mathbf{f} \sqsubset \text{rec}$ for any function \mathbf{f} distinct from rec . Then in the STTRS encoding of a system \mathbb{T} term M , each rule $((\mathbf{f} \ \overline{p}_1 \dots \overline{p}_k)) \rightarrow s$ satisfies either condition 1 or the following condition 2: *there are a term r and sequences of patterns $\overline{q}_1, \dots, \overline{q}_k$ such that for any i, j , $q_{i,j}$ is subterm of $p_{i,j}$, there exist i_0, j_0 s.t. $q_{i_0, j_0} \neq p_{i_0, j_0}$ and*

$$s = r\{x / ((\mathbf{f} \ \overline{q}_1 \dots \overline{q}_k))\}, \text{ and } r \sqsubset \mathbf{f} \tag{2}$$

So the syntactic ingredient is fairly expressive, since it will allow to validate system \mathbb{T} programs.

As to the full termination criterion, including the typing ingredient, we believe it is general enough to embed Hofmann's system SLR [14], which is a restriction of system \mathbb{T} based on safe recursion and using linear types, and which characterizes the class of polytime functions.

Formally, the class \mathcal{F} needs to be split 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. The rules of a linear type system for STTRS terms are in Figure 1. A program *satisfies the termination criterion* if every rule $((\mathbf{f} \ \overline{p}_1 \dots \overline{p}_k)) \rightarrow s$ satisfies:

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

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

1. either $\mathbf{f} \in \mathcal{RF}$, there are a term r and sequences of patterns $\bar{q}_1, \dots, \bar{q}_k$ such that for any i, j , $q_{i,j}$ is subterm of $p_{i,j}$, there exist i_0, j_0 s.t. $q_{i_0, j_0} \neq p_{i_0, j_0}$ and $\Gamma \mid x : B, \Delta \vdash r : B$, $s = r\{x / ((\mathbf{f} \ \bar{q}_1 \dots \bar{q}_k))\}$, and $r \sqsubset \mathbf{f}$;
2. or $\mathbf{f} \in \mathcal{NF}$, s is typable, and $s \sqsubset \mathbf{f}$.

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.

By a standard reducibility argument, one get the following:

Theorem 21 *If a program satisfies the termination criterion, then any of its terms is strongly normalizing.*

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 $|v| \leq n$.

Lemma 22 *If $n \in \mathbb{N}$ is a bound of first-order values for t and $t \rightarrow^* s$, then n is also a bound of first-order values for s .*

Suppose a function symbol \mathbf{f} takes n base arguments. Then \mathbf{f} is said to have *base values bounded by a function $q : \mathbb{N}^n \rightarrow \mathbb{N}$* if $(\mathbf{f} \ t_1 \dots t_n)$ has $q(|t_1|, \dots, |t_n|)$ as a bound of its first-order values whenever t_1, \dots, t_n are first-order values. Given a term t , its *definitional depth* is the maximum, over any function symbol \mathbf{f} appearing in t , of the length of the longest descending \sqsubset -chain starting from \mathbf{f} . The definitional depth of t is denoted as $\partial(t)$. Given a function symbol \mathbf{f} , $\mathcal{R}(\mathbf{f})$ denotes the set of terms appearing in the right-hand side of rules for \mathbf{f} , not taking into account recursive calls. More formally, r belongs to $\mathcal{R}(\mathbf{f})$ if there is a rule $((\mathbf{f} \ \bar{p}_1 \dots \bar{p}_k)) \rightarrow s$ such that $s = r\{x / ((\mathbf{f} \ \bar{q}_1 \dots \bar{q}_k))\}$ (where x might not occur in r) and $r \sqsubset \mathbf{f}$. 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 \mathbf{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. It is defined formally by:

$$\begin{aligned} \|v\| &= 1, \text{ if } v \text{ is a first-order value,} \\ \|(t_0 \ t_1 \ \dots \ t_n)\| &= \sum_{i=0}^n \|t_i\|, \\ \|\alpha\| &= 1, \text{ if } \alpha = x, \mathbf{c} \text{ or } \mathbf{f}. \end{aligned}$$

For instance $\|(\mathbf{s} \ (\mathbf{s} \ x))\| = 3$ and $\|(\mathbf{s} \ (\mathbf{s} \ \mathbf{0}))\| = 1$. We define a rewrite relation \Rightarrow which is like \rightarrow , 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:

Lemma 23 *For every n and for every t , $\mathcal{TS}_t(n) \geq \|t\|$.*

Proof : A simple induction on t . □

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

Lemma 24 *If n is a bound of first-order values for t , and $t \Rightarrow s$, then $\mathcal{TS}_t(n) > \mathcal{TS}_s(n)$.*

Proof : Suppose that $t \Rightarrow s$ and let $((\mathbf{f} \ r_1 \dots r_m))$ be the redex fired in t to produce s . Then we can say that there is p such that

$$\begin{aligned} \mathcal{TS}_t(n) &= \mathcal{TS}_{((\mathbf{f} \ r_1 \dots r_m))}(n) + p; \\ \mathcal{TS}_s(n) &= \mathcal{TS}_q(n) + p. \end{aligned}$$

Let us now distinguish two cases:

- If $\mathbf{f} \in \mathcal{RF}$, then q is obtained by at most $m \cdot n$ rewrite steps from $((\mathbf{f} \ r_1 \dots r_m))$ and does not contain any instance of \mathbf{f} anymore. By an easy combinatorial argument, one realizes that, indeed,

$$\mathcal{TS}_q(n) \leq \left(\sum_{\substack{r_j \in \mathcal{HO} \\ j \leq \text{arity}(\mathbf{f})}} m \cdot n \cdot \mathcal{TS}_{r_j}(X) \right) + \left(\sum_{s \in \mathcal{R}(\mathbf{f})} m \cdot n \cdot \mathcal{TS}_s(X) \right)$$

which, by definition, is strictly smaller than $\mathcal{TS}_{((\mathbf{f} \ r_1 \dots r_m))}(n)$.

- if $\mathbf{f} \in \mathcal{NF}$, then q is such that $((\mathbf{f} \ r_1 \dots r_m)) \Rightarrow q$. As a consequence

$$\mathcal{TS}_{((\mathbf{f} \ r_1 \dots r_m))}(n) = 1 + \left(\sum_{1 \leq j \leq m} \mathcal{TS}_{r_j}(n) \right) + \left(\sum_{w \in \mathcal{R}(\mathbf{f})} \mathcal{TS}_w(n) \right)$$

while q , containing possibly at most one instance of each higher-order value in r_1, \dots, r_m , is such that:

$$\mathcal{TS}_q(n) \leq \left(\sum_{1 \leq j \leq m} \mathcal{TS}_{r_j}(n) \right) + \left(\sum_{w \in \mathcal{R}(\mathbf{f})} \mathcal{TS}_w(n) \right).$$

This concludes the proof. □

It is now easy to reach our goal:

Proposition 25 *Suppose that R satisfies the termination criterion. Moreover, suppose that \mathbf{f} has base values bounded by a function $q : \mathbb{N}^n \rightarrow \mathbb{N}$. Then, there is a polynomial $p : \mathbb{N} \rightarrow \mathbb{N}$ such that if t_1, \dots, t_n are first-order values and $(\mathbf{f} \ t_1 \dots t_n) \rightarrow^m s$, then $m, |s| \leq p(q(|t_1|, \dots, |t_n|))$.*

Proof : Let us first prove the following statement: if $(\mathbf{f} \ t_1 \dots t_n) \Rightarrow^m s$, then there is q polynomial such that $m, \|s\| \leq p(q(|t_1|, \dots, |t_n|))$ Actually, the polynomial we are looking for is precisely

$$q(X) = \left(\sum_{s \in \mathcal{R}(\mathbf{f})} n \cdot X \cdot \mathcal{TS}_s(X) \right) + n.$$

Indeed, observe that by definition

$$\mathcal{TS}_{(\mathbf{f} \ t_1 \dots t_n)}(X) = q(X)$$

and that, by lemmas 24 and 23, this is both a quantity that decreases at any reduction step and which bounds from above the collapsed size of any reduct of $(\mathbf{f} \ t_1 \dots t_n)$. Now, observe that:

- If m is a bound of first order values for t , then $|t| \leq m \cdot \|t\|$;
- If m is a bound of first order values for t , then the number of “real” \rightarrow -reduction steps corresponding to each \Rightarrow -reduction step from t is bounded by $m \cdot k$, where k is the maximum arity of function symbols in the underlying STTRS;
- Call-by-value is a strongly confluent rewrite relation, and as a consequence the possible number of reduction steps from a term does not depend on the specific reduction order. Similarly for the size of reducts.

This concludes the proof. \square

To convince yourself that linearity is needed to get a result like Proposition 25, consider the following STTRS, whose terms cannot be typed in our linear type systems:

$$\begin{array}{ll} ((\mathbf{comp} \ x \ y) \ z) & \rightarrow \quad (x(yz)) \\ (\mathbf{autocomp} \ x) & \rightarrow \quad (\mathbf{comp} \ x \ x) \\ (\mathbf{id} \ x) & \rightarrow \quad x \\ (\mathbf{expid} \ 0) & \rightarrow \quad \mathbf{id} \\ (\mathbf{expid} \ (s \ x)) & \rightarrow \quad (\mathbf{autocomp} \ (\mathbf{expid} \ x)) \end{array}$$

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

5.2 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:

$$\begin{array}{l} S ::= \mathbf{N} \mid S \multimap S \\ A ::= S \mid A \rightarrow A \end{array}$$

Types of the first (resp. second) grammar are called linear types (resp. types) and denoted as $R, S \dots$ (resp. $A, B, C \dots$).

The linear function type \multimap is a subtype of \rightarrow , i.e., one can define a relation \sqsubseteq between types by stipulating that $S \multimap R \sqsubseteq S \rightarrow R$ and by closing the rule above in the usual way, namely by imposing that $A \rightarrow B \sqsubseteq C \rightarrow D$ whenever $C \sqsubseteq A$ and $B \sqsubseteq D$.

We now consider the following new set of constructors:

$$\begin{aligned} D_P = \{ & + : \mathbf{N} \multimap \mathbf{N} \multimap \mathbf{N}, \max : \mathbf{N} \multimap \mathbf{N} \multimap \mathbf{N}, \\ & \times : \mathbf{N} \rightarrow \mathbf{N} \rightarrow \mathbf{N} \} \cup \{ \bar{n} : \mathbf{N} \mid n \in \mathbf{N}^* \}, \end{aligned}$$

and we consider the following grammar of Church-typed terms

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

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

- in $(M^{A \rightarrow B} N^A)^B$, $FV(N^A)$ does not contain any variable $x^{R \rightarrow S}$ with a linear type $R \rightarrow S$;
- in $(\lambda x^A. M^B)^{A \rightarrow B}$ the variable x^A occurs *at least once* in M^B ;
- in $(\lambda x^S. M^R)^{S \rightarrow R}$, the variable x^S occurs *exactly once* in M^R .

One can check that this class of Church-typed terms is preserved by β -reduction. A *higher-order max-polynomial* (HOMP) is a term defined above and which is in β -normal form. The interpretations $\llbracket M \rrbracket$, and the equivalence relation \equiv are defined in a similar way as before. We define the following objects and constructions on objects:

- \mathcal{N} is the domain of *strictly positive* integers, equipped with the natural order, denoted here $\leq_{\mathcal{N}}$,
- 1 is the trivial order with one point;
- 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.

Given an object σ of the category \mathbb{FPOS} and an element $e \in \sigma$, the *size* $|e|$ of e is defined as follows:

- If σ is \mathcal{N} , then $|e|$ is simply the natural number e itself;
- If $\sigma = \sigma_1 \times \dots \times \sigma_n$, then the size of $e = (e_1, \dots, e_n)$ exists if for any $i \in \{1, \dots, n\}$ the size $|e_i|$ exists, and in this case $|e| = \sum_{i=1}^n |e_i|$.
- If $\sigma = \tau \Rightarrow \rho$, then the size of e , if it exists, is the minimum natural number $|e|$ such that for every $f \in \tau$, $|e| \leq |e(f)| \leq |e| + |f|$.

We can now define a new construction on \mathbb{FPOS} objects: if σ, τ are objects, then $\sigma \multimap \tau$ is the restriction of the order $\sigma \Rightarrow \tau$ to the elements e which admit a size.

We denote by $\llbracket A \rrbracket_{\leq}$ the semantics of A as an object of \mathbb{FPOS} , where \mathbf{N} is mapped to \mathcal{N} , \rightarrow is mapped to \Rightarrow and \multimap to \multimap . As before, any element of $e \in \llbracket A \rrbracket_{\leq}$ can be mapped onto an element $e \downarrow$ of $\llbracket A \rrbracket$. We define $\llbracket M \rrbracket_{\leq}$ for any HOMP M in the natural way.

Proposition 26 *Let M be a closed HOMP of type A . Then $\llbracket M \rrbracket_{\leq} \in \llbracket A \rrbracket_{\leq}$. Moreover, if A is a linear type S , then the size of $\llbracket M \rrbracket_{\leq}$ is defined.*

Lemma 27 *If M is a HOMP of type $\mathbf{N}^m \rightarrow \mathbf{N}$ (with m arguments) and such that $FV(M) = \{y_1 : \mathbf{N}, \dots, y_k : \mathbf{N}\}$, then the function $\llbracket M \rrbracket$ is bounded by a polynomial and satisfies: $\forall i \in \{1, k+m\}$, $(\llbracket M \rrbracket(x_1, \dots, x_k))(x_{k+1}, \dots, x_{k+m}) \geq x_i$.*

Proof : We prove the statement by induction on M :

- If $M = x$ then the statement holds.
- If $M = \bar{n}, +, max$ or \times , then the statement is obviously also true (note that for \times we are using the fact that the base domain is \mathbb{N}^* and not \mathbb{N}).
- If M is an application, it can be written as $M = (\dots (M_0) M_1) \dots M_n$ where M_0 is not an application and $n \geq 1$. Moreover M_0 cannot be an abstraction since M is in β -normal form, and it cannot be a variable y since M can only have free variables of type \mathbf{N} . So $M_0 = \mathbf{c}$ for $\mathbf{c} = +, max$ or \times , and therefore $n \leq 2$. We obtain that the M_i s for $1 \leq i \leq 2$ are of type \mathbf{N} hence also satisfy the hypothesis, and thus by i.h. they satisfy the claim. Therefore the claim is also valid for M .
- Finally the only possibility left is $M = \lambda x. M_1$. By definition of HOMP we know then that x is a free variable of M_1 of type \mathbf{N} , and as M_1 satisfies the hypothesis, by i.h. we know that M_1 satisfies the claim. Therefore the claim is valid for M .

This concludes the proof. □

The following will be useful to obtain the Subterm Property:

Lemma 28 *For every type A there is a closed HOMP of type A .*

Proof : By induction on A :

- If A is simply \mathbf{N} , then the required HOMP is simply $\bar{1}$;
- If A is $A_1 \mapsto_1 \dots A_n \mapsto_n \mathbf{N}$ (where \mapsto_i is either \rightarrow or \multimap), then the required HOMP is

$$\lambda x_1^{A_1} \dots \lambda x_n^{A_n} . (x_1 M_1^1 \dots M_1^{m_1}) + \dots + (x_n M_n^1 \dots M_n^{m_n})$$

where the HOMPs M_i^j exist by induction hypothesis. □

5.3 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 A by:

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

For instance if $A = (D_1 \rightarrow D_2) \rightarrow D_3$ we can take for B any of the types: $(\mathbf{N} \multimap \mathbf{N}) \rightarrow \mathbf{N}$, $(\mathbf{N} \rightarrow \mathbf{N}) \rightarrow \mathbf{N}$, etc. In the sequel we will write \underline{A} for any of these types B . Then $[\cdot]$ is extended inductively to all $t \in \mathcal{T}$ as for HOP assignments.

Lemma 29 *Let $t \in \mathcal{T}$ of type A and $FV(t) = \{y_1 : A_1, \dots, y_n : A_n\}$, then: $[t]$ is a HOMP, with a type \underline{A} , and $FV([t]) = \{y_1 : \underline{A}_1, \dots, y_n : \underline{A}_n\}$ for some types \underline{A}_i , $1 \leq i \leq n$.*

Additive HOMP assignments are defined just as additive HOP assignments.

Lemma 30 *Let $[\cdot]$ be an additive HOMP assignment. Then there exists $\gamma \geq 1$ such that for any value v of type D , where D is a data type, we have $\llbracket v \rrbracket_{\prec} \leq \gamma \cdot |v|$.*

Now, we say that an assignment $[\cdot]$ is a *quasi-interpretation for R* if for any rule $l \rightarrow r$ of R , it holds that $\llbracket l \rrbracket_{\leq} \geq \llbracket r \rrbracket_{\leq}$. Observe that contrarily to the case of polynomial interpretations these inequations are not strict, and moreover they are stated with respect to the new domains, taking into account the distinction between the two connectives \rightarrow and \multimap .

The interpretation of a term does not, like in the strict case, necessarily decrease along a reduction step. However, it cannot increase:

Lemma 31 *If $[\cdot]$ is a quasi-interpretation and if $t \rightarrow^* s$, then $\llbracket s \rrbracket_{\leq} \leq \llbracket t \rrbracket_{\leq}$.*

Proof : [Sketch] This can be done in a way analogous to what has been done for polynomial interpretations with Lemma 16, using intermediary lemmas for substitutions and contexts similar to lemmas 14 and 15, but it is actually easier because here we are not considering a strict order. □

The previous lemma, together with the possibility of forming HOMPs of arbitrary type (Lemma 28) implies the following, crucial, property:

Proposition 32 (Subterm Property) *Suppose that an STTRS R has an additive quasi-interpretation $[\cdot]$. Then, for every function symbol \mathbf{f} of arity n with base arguments, there is a polynomial $p : \mathbb{N}^n \rightarrow \mathbb{N}$ such that if $(\mathbf{f} t_1 \dots t_n) \rightarrow^* s$ and if s contains an occurrence of a base term r , then $|r| \leq p(|t_1|, \dots, |t_n|)$.*

Proof : Denote $t = (\mathbf{f} t_n \dots t_1)$. Its type A can be written as $A = A_1, \dots, A_k \rightarrow D$, where D is a base type. By Lemma 28, for any $i \in 1, \dots, k$ there is a closed HOMP M_i of type \underline{A}_i . Consider now $t' = (\mathbf{f} x_1 \dots x_n)$ where for $i \in 1, \dots, n$, x_i is a free variable of same type as t_i .

Then $M = [\mathbf{f} \ x_1 \dots x_n] M_1 \dots M_k$ is a HOMP of type N with free variables x_i of type N , for $i \in 1, \dots, n$. By Lemma 27 we deduce from that that there exists a polynomial q such that:

$$y_i \leq \llbracket M \rrbracket_{\leq}(y_1, \dots, y_n) \leq q(y_1, \dots, y_n), \text{ for } i = 1, \dots, n.$$

Moreover we have:

$$\begin{aligned} \llbracket ([t] \ M_1 \dots M_k) \rrbracket_{\leq} &= \llbracket ([t'] \ M_1 \dots M_k) \rrbracket_{\leq}(\llbracket [t] \rrbracket_{\leq}, \dots, \llbracket [t_n] \rrbracket_{\leq}) \\ &= \llbracket M \rrbracket_{\leq}(\llbracket [t_1] \rrbracket_{\leq}, \dots, \llbracket [t_n] \rrbracket_{\leq}) \\ &\leq q(\llbracket [t_1] \rrbracket_{\leq}, \dots, \llbracket [t_n] \rrbracket_{\leq}) \\ &\leq q(\alpha|t_1|, \dots, \alpha|t_n|) \end{aligned}$$

for some α , because $[\cdot]$ is an additive quasi-interpretation, thanks to Lemma 30. So finally there is a polynomial p such that:

$$\llbracket ([t] \ M_1 \dots M_k) \rrbracket_{\leq} \leq p(|t_1|, \dots, |t_n|).$$

Now, as $t \rightarrow^* s$, by Lemma 31 we have $\llbracket [t] \rrbracket_{\leq} \geq \llbracket [s] \rrbracket_{\leq}$. Therefore $\llbracket ([t] \ M_1 \dots M_k) \rrbracket_{\leq} \geq \llbracket [s] \ M_1 \dots M_k \rrbracket_{\leq}$. So we get: $\llbracket [s] \ M_1 \dots M_k \rrbracket_{\leq} \leq p(|t_1|, \dots, |t_n|)$. Besides, by assumption we know that s can be written as $s = s' \{y/r\}$. By Lemma 27 we have $\llbracket [s'] \ M_1 \dots M_k \rrbracket_{\leq}(y) \geq y$, because $([s'] \ M_1 \dots M_k)$ has type N and only one free variable y which is also of type N . Therefore we get $\llbracket [s] \ M_1 \dots M_k \rrbracket_{\leq} = \llbracket [s'] \ M_1 \dots M_k \rrbracket_{\leq}(\llbracket [r] \rrbracket_{\leq}) \geq \llbracket [r] \rrbracket_{\leq}$. So finally by combining the two inequalities we obtained we get $\llbracket [r] \rrbracket_{\leq} \leq \llbracket [s] \ M_1 \dots M_k \rrbracket_{\leq} \leq p(|t_1|, \dots, |t_n|)$. \square

And here is the main result of this Section:

Theorem 33 *If a program P has an additive quasi-interpretation, P satisfies the termination criterion and \mathbf{f} has arity n with base arguments, then there is a polynomial $p : \mathbb{N}^n \rightarrow \mathbb{N}$ such that if $(\mathbf{f} \ t_1 \dots t_n) \rightarrow^m s$, then $m, |s| \leq p(|t_1|, \dots, |t_n|)$. So if \mathbf{f} has a type $D_1, \dots, D_n \rightarrow D$ then it is Ptime.*

Proof : A consequence of Proposition 32 and of Proposition 25. \square

Notice how Theorem 33 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.

5.4 Examples

Consider the program `foldl` given by:

$$\begin{aligned} (1) \quad & \text{foldl } f \ b \ \text{nil} && \rightarrow \ b \\ (2) \quad & \text{foldl } f \ b \ (\text{cons } x \ xs) && \rightarrow \ (f \ x \ (\text{foldl } f \ b \ xs)) \end{aligned}$$

with types:

$$\begin{aligned} \text{foldl} & : (D \times E \rightarrow E) \times E \times L(D) \rightarrow E \\ f & : D \times E \rightarrow E \\ \text{nil} & : L(E) \\ \text{cons} & : E \times L(E) \rightarrow L(E) \end{aligned}$$

Now, we choose as assignment:

$$\begin{aligned} [\text{nil}] &= 1 && : \mathbf{N} \\ [\text{cons}](n, m) &= n + m + 1 && : \mathbf{N} \rightarrow \mathbf{N} \rightarrow \mathbf{N} \\ [\text{foldl}](\phi, p, n) &= p + n\phi(1, 1) && : (\mathbf{N} \multimap \mathbf{N} \multimap \mathbf{N}) \rightarrow \mathbf{N} \rightarrow \mathbf{N} \rightarrow \mathbf{N} \end{aligned}$$

Observe the \multimap in the type of the first argument of `foldl` which is the way to restrict the domain of arguments. We denote $[b] = p$, $[x] = n$, $[xs] = m$, $[f] = \phi$. We then obtain the following interpretations of terms:

$$\begin{aligned} [\text{foldl } f \ b \ \text{nil}] &= p + 1\phi(1, 1) \\ [b] &= p \\ [\text{foldl } f \ b \ \text{cons}(x, xs)] &= p + (n + m + 1)\phi(1, 1) \\ [f(x, (\text{foldl } f \ b \ xs))] &= \phi(n, (p + m\phi(1, 1))) \end{aligned}$$

The condition $\llbracket r \rrbracket_{\leq} \leq \llbracket l \rrbracket_{\leq}$ holds for (1), because we have:

$$p \leq p + \phi(1, 1)$$

holds for any p . As to rule (2) consider $\phi \in (\mathcal{N} \times \mathcal{N} \multimap \mathcal{N})$, n , p and m in \mathcal{N} . So we know that there exists $c \geq 0$ such that:

$$c \leq \phi(x_1, x_2) \leq x_1 + x_2 + c, \text{ for any } x_1, x_2 \in \mathcal{N}. \quad (*)$$

Then we have:

$$\begin{aligned} \phi(n, (p + m\phi(1, 1))) &\leq n + p + m\phi(1, 1) + c \quad \text{by } (*), \\ &\leq n\phi(1, 1) + p + m\phi(1, 1) + c, \\ &\leq p + (n + m + 1)\phi(1, 1). \end{aligned}$$

where for the two last steps we used $\phi(1, 1) \geq 1$ and $\phi(1, 1) \geq c$ (by $(*)$). So $\llbracket r \rrbracket_{\leq} \leq \llbracket l \rrbracket_{\leq}$ also holds for (2) and we have an additive quasi-interpretation.

As to the termination criterion, it is satisfied because in (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 33 and conclude that: if the termination criterion is satisfied by all functions, if $t^{D \times E \rightarrow E}$, b^E are terms and $[t]$ is a HOMP with type $\mathbf{N} \multimap \mathbf{N} \multimap \mathbf{N}$, then $(\mathbf{fold1} \ t \ b)$ is a polynomial time program of type $L(D) \rightarrow E$.

6 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, and to overcome 1. one possibility consists 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.

6.1 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 [3] (see also [15] for the relation with non-size-increasing, at first-order). We could examine whether the various flavours they consider on the termination criteria and the interpretations (e.g. sup-interpretations) could suggest ideas in our higher-order setting.

At higher-order, various approaches based on realizability have been used [14, 16, 17]. 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 other ICC systems. Several variants of system T based on restriction of recursion and linearity conditions [14, 18, 19] have been proposed which characterize polynomial time. Another system [20] with a linear type system for non-size-increasing computation offers more intensional expressivity. We believe we should be able to embed these systems in our approach, but leave this for future work. With respect to the first group of works, we offer the same benefit as [20], that is to say the nesting of recursions thanks to the distinction of non-size-increasing functions. With respect to [20], the advantages we bring are the use

of recursively defined functions generalizing the System \mathbb{T} recursor and a slightly more general handling of higher-order arguments, allowing for instance to validate the `foldl` program.

Some other works are based on type systems built out of variants of linear logic [21, 22, 23]. 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 [24].

7 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 ICC systems for higher-order languages. While up to now quite distinct techniques had been successful for providing expressive criteria for Ptime complexity at first-order and at higher-order respectively, our approach allows to take advantage simultaneously of 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 Ptime based on a termination criterion and on quasi-interpretations, which allows to give some sufficient conditions for programs built with higher-order functionals (like `foldl`) to work in bounded time. We think this setting should allow in future work to devise new systems for ensuring complexity bounds for higher-order languages, which would be more expressive algorithmically.

References

- [1] G. Bonfante, A. Cichon, J.-Y. Marion, and H. Touzet, “Algorithms with polynomial interpretation termination proof,” *J. Funct. Program.*, vol. 11, no. 1, pp. 33–53, 2001.
- [2] T. Yamada, “Confluence and termination of simply typed term rewriting systems,” in *Proceedings of RTA’01*, ser. LNCS, vol. 2051. Springer, 2001, pp. 338–352.
- [3] G. Bonfante, J.-Y. Marion, and J.-Y. Moyon, “Quasi-interpretations a way to control resources,” *Theor. Comput. Sci.*, vol. 412, no. 25, pp. 2776–2796, 2011.
- [4] D. A. Plaisted, “A recursively defined ordering for proving termination of term rewriting systems,” University of Illinois, Urbana, Illinois, Tech. Rep. R-78-943, September 1978.
- [5] N. Dershowitz, “Orderings for term-rewriting systems,” *Theoretical Computer Science*, vol. 17, no. 3, pp. 279–301, 1982.
- [6] D. Lankford, “On proving term rewriting systems are noetherian,” Louisiana Tech. University, Tech. Rep. MTP-3, 1979.
- [7] U. Dal Lago and S. Martini, “On constructor rewrite systems and the lambda-calculus,” in *ICALP (2)*, 2009, pp. 163–174.
- [8] J.-P. Jouannaud and M. Okada, “A computation model for executable higher-order algebraic specification languages,” in *LICS*, 1991, pp. 350–361.
- [9] J. W. Klop, V. van Oostrom, and F. van Raamsdonk, “Combinatory reduction systems: Introduction and survey,” *Theor. Comput. Sci.*, vol. 121, no. 1&2, pp. 279–308, 1993.
- [10] J.-P. Jouannaud and A. Rubio, “The higher-order recursive path ordering,” in *LICS*, 1999, pp. 402–411.
- [11] T. Aoto and T. Yamada, “Termination of simply typed term rewriting by translation and labelling,” in *Proceedings of RTA’03*, ser. LNCS, vol. 2706. Springer, 2003.

- [12] U. Dal Lago and S. Martini, “Derivational complexity is an invariant cost model,” in *FOPARA*, 2009, pp. 88–101.
- [13] J.-Y. Marion and J.-Y. Moyon, “Efficient First Order Functional Program Interpreter with Time Bound Certifications,” in *Proceedings of LPAR’00*, ser. LNAI, vol. 1955. Springer, 2000, pp. 25–42.
- [14] M. Hofmann, “A mixed modal/linear lambda calculus with applications to Bellantoni-Cook safe recursion,” in *Proceedings of CSL*, ser. LNCS, 1997, pp. 275–294.
- [15] R. Amadio, “Synthesis of max-plus quasi-interpretations,” *Fundamenta Informaticae*, vol. 65, pp. 29–60, 2005.
- [16] U. Dal Lago and M. Hofmann, “Realizability models and implicit complexity,” *Theor. Comput. Sci.*, vol. 412, no. 20, pp. 2029–2047, 2011.
- [17] A. Brunel and K. Terui, “Church \Rightarrow Scott = Ptime: an application of resource sensitive realizability,” in *Proceedings of workshop DICE*, ser. EPTCS, vol. 23, 2010, pp. 31–46.
- [18] S. J. Bellantoni, K.-H. Niggl, and H. Schwichtenberg, “Higher type recursion, ramification and polynomial time,” *Ann. Pure Appl. Logic*, vol. 104, no. 1-3, pp. 17–30, 2000.
- [19] U. Dal Lago, “The geometry of linear higher-order recursion,” in *LICS*, 2005, pp. 366–375.
- [20] M. Hofmann, “Linear types and non-size-increasing polynomial time computation,” *Information and Computation*, vol. 183, no. 1, pp. 57–85, 2003.
- [21] P. Baillot and K. Terui, “Light types for polynomial time computation in lambda calculus,” *Inf. Comput.*, vol. 207, no. 1, pp. 41–62, 2009.
- [22] M. Gaboardi and S. Ronchi Della Rocca, “A soft type assignment system for lambda - calculus,” in *Proceedings of CSL’07*, ser. LNCS, vol. 4646. Springer, 2007, pp. 253–267.
- [23] P. Baillot, M. Gaboardi, and V. Mogbil, “A polytime functional language from light linear logic,” in *ESOP*, 2010, pp. 104–124.
- [24] U. Dal Lago and M. Gaboardi, “Linear dependent types and relative completeness,” in *LICS*, 2011, pp. 133–142.