A CHARACTERIZATION OF BASIC FEASIBLE FUNCTIONALS THROUGH HIGHER-ORDER REWRITING AND TUPLE INTERPRETATIONS

PATRICK BAILLOT [a], UGO DAL LAGO [b], CYNTHIA KOP [c], AND DEIVID VALE [c]

[a] Univ. Lille, CNRS, Inria, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France
e-mail address: patrick.baillot@univ-lille.fr

[b] Università di Bologna, Inria
e-mail address: ugo.dallago@unibo.it

[c] Radboud University Nijmegen
e-mail address: c.kop@cs.ru.nl, deividvale@cs.ru.nl

Abstract. The class of type-two basic feasible functionals ($\mathtt{BFF}_2$) is the analogue of $\mathtt{FP}$ (polynomial time functions) for type-2 functionals, that is, functionals that can take (first-order) functions as arguments. $\mathtt{BFF}_2$ can be defined through Oracle Turing Machines with running time bounded by second-order polynomials. On the other hand, higher-order term rewriting provides an elegant formalism for expressing higher-order computation. We address the problem of characterizing $\mathtt{BFF}_2$ by higher-order term rewriting. Various kinds of interpretations for *first-order* term rewriting have been introduced in the literature for proving termination and characterizing first-order complexity classes. In this paper, we consider a recently introduced notion of cost–size tuple interpretations for higher-order term rewriting and see second order rewriting as ways of computing type-2 functionals. We then prove that the class of functionals represented by higher-order terms admitting polynomially bounded cost–size interpretations exactly corresponds to $\mathtt{BFF}_2$.

1. Introduction

Computational complexity classes — and in particular those relating to polynomial time and space [HS69, Cob65] — capture the concept of a feasible problem, and as such have been scrutinized with great care by the scientific community in the last fifty years. The fact that even apparently simple problems (such as nontrivial separation between those classes) remain open today has highlighted the need for a comprehensive study aimed at investigating the deep nature of computational complexity. The so-called implicit computational complexity [BC92, Lei91, Oit01, DLH11, BDBRDR18] fits into this picture, and is concerned

Key words and phrases: Basic Feasible Functions; Higher-Order Term Rewriting; Tuple Interpretation; Implicit Computational Complexity .

This work is supported by the NWO TOP project "Implicit Complexity through Higher-Order Rewriting", NWO 612.001.803/7571, the NWO VIDI project "Constrained Higher-Order Rewriting and Program Equivalence", NWO VI.Vidi.193.075, and the ERC CoG "Differential Program Semantics", GA 818616.

with characterizations of complexity classes based on tools from mathematical logic and the theory of programming languages.

One of the areas involved in this investigation is certainly that of term rewriting [Ter03], which has proved useful as a tool for the characterization of complexity classes. In particular, the class FP (i.e., of first-order functions computable in polynomial time) has been characterized through variations of techniques originally introduced for *termination*, e.g., the interpretation method [MN70, Lan79], path orders [Der82], or dependency pairs [GTS04]. Some examples of such characterizations can be found in [BW96, BCMT01, BMM11, AM13, BDL12].

After the introduction of FP, it became clear that the study of computational complexity also applies to *higher-order functionals*, which are functions that take not only data but also other functions as inputs. The pioneering work of Constable [Con73], Mehlhorn [Meh76], and Kapron and Cook [KC96] laid the foundations of the so-called higher-order complexity, which remains a prolific research area to this day. Some motivations for this line of work can be found for instance in computable analysis [KC12], NP search problems [BCE+98], and programming language theory [DR06].

There have been several proposals for a class of type-2 functionals that generalizes FP. However, the most widely accepted one is the class $\mathtt{BFF}_2$ of *type-two basic feasible functionals*. This class can be characterized based on function algebras, similar to Cobham-style, but it can also be described using Oracle Turing Machines. The class $\mathtt{BFF}_2$ was then the object of study by the research community, which over the years has introduced a variety of characterizations, e.g., in terms of programming languages with restricted recursion schemes [IRK01, DR06], typed imperative languages [HKMP20, HKMP22], and restricted forms of iteration in OTMs [KS18].

The studies cited above present structurally complex programming languages and logical systems, precisely due to the presence of higher-order functions. It is not currently known whether it is possible to give a characterization of $\mathtt{BFF}_2$ in terms of mainstream concepts of rewriting theory, although the latter has long been known to provide tools for the modeling and analysis of functional programs with higher-order functions [KvOvR93].

This paper goes precisely in this direction by showing that the interpretation method in the form studied by Kop and Vale [KV21, KV23] provides the right tools to characterize $\mathtt{BFF}_2$. More precisely, we consider a class of higher-order rewrite systems admitting cost–size tuple interpretations (with some mild upper-bound conditions on their cost and size components) and show that this class contains exactly the functionals in $\mathtt{BFF}_2$. Such a characterization could not have been obtained employing classical integer interpretations as e.g. in [BCMT01] because $\mathtt{BFF}_2$ crucially relies on some conditions both on *size* and on *time*. We believe that a benefit of this characterization is that it opens the way to effectively handling programs or executable specifications implementing $\mathtt{BFF}_2$ functions, in full generality. For instance, we expect that such a characterization could be integrated into rewriting-based tools for complexity analysis of term rewriting systems such as e.g., [AMS16].

Contributions. We consider simply-typed term rewriting systems (STRS for short) with partial application but no λ-abstraction. The contributions of this paper are as follows.

- We provide a compatibility theorem for cost–size interpretations of STRSs with respect to innermost reduction, whose proof is simpler than that of [KV23]; this captures the fact that cost–size interpretations provide safe upper bounds on the length of reduction sequences, and that the size interpretation of a term cannot increase.

- We propose a natural definition of computation of type-2 functionals by an STRS, which can be of more general interest than the specific characterization of the class $\mathtt{BFF}_2$ that we focus on in this paper; this is purely operational in nature and revolves around the use of rewrite rules modeling calls for an "oracle". This notion of computation is, intuitively, a rewriting counterpart for oracle Turing machines.
- We prove a soundness result, stating that any (orthogonal) STRS with a polynomially bounded cost–size interpretation computes a type-2 functional in $\mathtt{BFF}_2$; this proof uses, in particular, a term-graph rewriting argument.
- Conversely, we define an encoding of polynomial time Oracle Turing Machines in an STRS which shows that any type-2 functional in $\mathtt{BFF}_2$ can be computed by an STRS with a polynomially bounded cost–size interpretation.

Related Work. We describe here some related work about the topics addressed by this paper, namely implicit computational complexity, higher-order complexity classes, higher-order rewriting systems, and interpretations.

Implicit computational complexity refers to a line of work aiming at characterizing complexity classes without reference to machine models and explicit bounds on resources, but instead by relying on logical systems and programming language restrictions. It goes back to early work by Leivant [Lei91] and Bellantoni and Cook [BC92] and has used various methods coming in particular from recursion theory [LM94, Oit01], programming language restrictions [Jon01, KS17], linear logic [Gir98, Laf04, Bai11] and type systems [Hof03, LS16]. In the setting of term-rewriting it has taken advantage of contributions in the area of polynomial interpretations [BCMT98] and has provided a variety of characterizations for first-order complexity classes such as FP and PSPACE [MM00, BMM05, BW96, BCMT01, BMM11, AM13, BLM12].

The class of Basic Feasible Functionals BFF was introduced by Cook and Kapron [CK89] by means of bounded typed loop programs, and they showed that its type-2 restriction $\mathtt{BFF}_2$ coincides with a class that had been defined by Melhorn [Meh76]. They later provided a machine characterization of type-2 $\mathtt{BFF}_2$ by polynomial time Oracle Turing Machines (OTM) [KC91, KC96] which gave more confidence in the naturalness of this class. Several works then provided alternative characterizations of $\mathtt{BFF}_2$, in particular by restricted recursion schemes in some functional languages [IRK01, DR06], or typed imperative languages with insights coming from non-interference analysis [HKMP20, HKMP22], or by restricted forms of iteration in OTMs [KS18].

Higher-order interpretations have been introduced and investigated in [vdP96] in relation to termination issues but not to complexity classes. In [BDL12, BL16] a notion of higher-order polynomial interpretations was proposed which allowed to provide a characterization of the (first-order) class FP of polynomial time computable functions. However, the codomain considered was the domain $\mathbb{N}$ of natural numbers, not tuples, and this approach did not consider higher-order complexity. An investigation of higher-order complexity classes employing the higher-order interpretation method in the context of a pure higher-order functional language was proposed in [HP20]. However, this paper did not provide a characterization of the standard $\mathtt{BFF}_2$ class. Instead, it characterized a newly proposed class $\mathtt{SFF}_2$ (Safe Feasible Functionals) which is defined as the restriction of $\mathtt{BFF}_2$ to argument functions in FP (see Sect. 4.2 and the conclusion in [HP20]). Another related line of work is that of [FHHP15]: the authors consider a first order functional stream language, which allows to implement second order functionals and they study (first-order) interpretations for this language. In this way they define a subclass of $\mathtt{BFF}_2$ but do not characterize $\mathtt{BFF}_2$ itself. Closer to the present work, the paper [KV21] introduced the notion of tuple interpretation for higher-order rewriting systems and in particular the cost–size interpretations that we use here. Then [KV23] adapted this notion to the weak call-by-value strategy, which allowed for tighter bounds but also relaxed the approach by separating the cost and the size components.

Publication History. This paper is a revised and extended version of [BLKV24]. This version contains detailed proofs of the results and additional examples, which previously could not be added due to space constraints. The results in this paper appear in monograph form in the last author's PhD thesis [Val24, Chapter 6].

Outline of the Paper. The paper is organized as follows. We first provide some background on higher-order rewriting and simply typed term-rewriting system (STRS), and on type-two complexity (Section 2). We then recall the definition of cost–size interpretations and prove a compatibility theorem for cost–size interpretations of STRSs with respect to innermost reduction (Section 3). In Section 4 we state the main theorem of this paper, which says that the STRSs with polynomially bounded cost–size interpretations exactly characterize the $\mathtt{BFF}_2$ functionals. Section 5 is devoted to the proof of the first direction of this theorem, the soundness result. Section 6 contains the proof of the second direction, the completeness result. In Section 7 we conclude the paper and discuss future work.

Technical Overview. In this paper we see $\mathtt{BFF}_2$ (Definition 2.5) as the set of those type-2 functionals computed by an Oracle Turing Machine in polynomial time. We recall basic definitions of such theory in Section 2.2. The main result of this paper is a characterization of the class $\mathtt{BFF}_2$ via higher-order term rewriting.

In order to formally give a statement of this result we need to first establish some important notions such as the rewriting counterpart of an oracle (see Definition 4.2) and a computability notion for higher-order rewriting (see Definition 4.3). We state the main result in Theorem 4.5. It is proved in two parts. We first prove that if any term rewriting system in this class computes a higher-order functional, then this functional has to be in $\mathtt{BFF}_2$ (*soundness*). Conversely, we prove that all functionals in $\mathtt{BFF}_2$ are computed by this class of rewriting systems (*completeness*). We argue that the key ingredient towards achieving this characterization is the ability to split the dual notions of *cost* and *size* given by the usage of tuple interpretations.

Soundness at first seems straightforward. From Kop and Vale [KV23] we know that (call-by-value) higher-order rewrite systems that admit polynomial interpretations (with certain conditions on the interpretation of data constructors) satisfy the property that their runtime complexity is polynomially bounded. We could temptingly say if a term rewrite sytem computes (in the sense of Definition 4.3) a type-2 functional, it must do so in a polynomial number of steps, and hence the said functional must be in $\mathtt{BFF}_2$. However, this is not generally the case due to the size-explosion problem, i.e., in a polynomial number of steps we could iterate over copied data. We solve this issue by restricting the interpretation of data constructors in Definition 4.4 and by employing term graph rewriting (see Section 5.3). Additionally, we need to guarantee that polynomial interpretations alone are capable of controlling the size of the calls to the oracle. More precisely, such interpretations do not allow for unbounded repeated iteration of oracle calls and the size of the resulting oracle call is polynomially related to the size of its given input. This is established by the *Oracle Subterm Lemma* (see Lemma 5.4). We then prove soundness in Theorem 5.16.

To prove completeness we work on an encoding of polynomial time Oracle Turing Machines (OTMs) in STRSs. We proceed as follows: we encode machine configurations as terms and machine transitions as rewriting rules that rewrite such configuration terms. With such encoding, we can faithfully simulate transitions on an OTM as one or more rewriting steps on the corresponding rewrite system. The correctness of such simulation is the subject of Lemma 6.2. Notice however that simulating OTMs by STRSs is not enough. We need to do it in polynomially many steps. For this we provide a rewrite system that can fully simulate a run of an OTM in polynomially many steps, which is given by Theorem 6.4.

2. Preliminaries

In this section we present a brief overview of simply typed term rewriting and the basic notions on basic feasible functionals. We assume the reader to be familiar with concepts from rewriting theory [Ter03] and basic notions of computability and complexity theory [AB09].

2.1. Higher-Order Rewriting.

We roughly follow the definition of a simply-typed term rewriting system (STRS) [Kus01]: types are simple, consisting of either base or functional types; terms are applicative, meaning they may be partially applied but contain no lambda abstractions. We limit our interest to second-order STRSs where all rules have base type (which in practice means the left- and right-hand side of rules are fully applied terms). Reduction follows an innermost evaluation strategy. We make this intuition precise below.

First, let us define our notion of a type. Let $\mathbb{B}$ be a nonempty set of *base types* which are ranged over by ι, κ, ν. The set $\mathbb{T}(\mathbb{B})$ of *simple types* over $\mathbb{B}$ is defined by the grammar $\mathbb{T}(\mathbb{B}) := \mathbb{B} \mid \mathbb{T}(\mathbb{B}) \Rightarrow \mathbb{T}(\mathbb{B})$. Types from $\mathbb{T}(\mathbb{B})$ are ranged over by σ, τ, ρ. The $\Rightarrow$ type constructor is right-associative, so we write $\sigma \Rightarrow \tau \Rightarrow \rho$ for $(\sigma \Rightarrow (\tau \Rightarrow \rho))$. Hence, every type σ can be written as $\sigma_1 \Rightarrow \cdots \Rightarrow \sigma_n \Rightarrow \iota$ with $n \geq 0$. We may write such types as $\vec{\sigma} \Rightarrow \iota$. The *order* of a type is: $\mathtt{ord}(\iota) = 0$ for $\iota \in \mathbb{B}$ and $\mathtt{ord}(\sigma \Rightarrow \tau) = \max(1 + \mathtt{ord}(\sigma), \mathtt{ord}(\tau))$.

Then, signatures are sets of function symbols, each of them equipped with a type. Formally, a *signature* $\mathbb{F}$ is a triple $(\mathbb{B}, \Sigma, \mathtt{typeOf})$ where $\mathbb{B}$ is a set of base types, Σ is a nonempty set of symbols, and $\mathtt{typeOf}$ is a function from Σ to $\mathbb{T}(\mathbb{B})$.

Next, we define terms. For each type σ with $\mathtt{ord}(\sigma) \leq 1$, we assume given a set $\mathbb{X}_\sigma$ of countably many variables and assume that $\mathbb{X}_\sigma \cap \mathbb{X}_\tau = \emptyset$ if $\sigma \neq \tau$. We let $\mathbb{X}$ denote $\cup_\sigma \mathbb{X}_\sigma$ and assume that $\Sigma \cap \mathbb{X} = \emptyset$. The set $\mathsf{T}(\mathbb{F}, \mathbb{X})$ — of *terms* built from $\mathbb{F}$ and $\mathbb{X}$ — collects those expressions s for which a judgment $s : \sigma$ can be deduced using the following rules:

$$\text{(ax)}\ \frac{x \in \mathbb{X}_\sigma}{x : \sigma} \qquad \text{(f-ax)}\ \frac{\mathsf{f} \in \Sigma \qquad \mathtt{typeOf}(\mathsf{f}) = \sigma}{\mathsf{f} : \sigma} \qquad \text{(app)}\ \frac{s : \sigma \Rightarrow \tau \qquad t : \sigma}{(s\,t) : \tau}$$

As usual, application of terms is left-associative, so we write $s\,t\,u$ for $((s\,t)\,u)$. Let $\mathtt{vars}(s)$ be the set of variables occurring in s. A term s is *ground* if $\mathtt{vars}(s) = \emptyset$. The *head symbol* of a term $\mathsf{f}\,s_1 \cdots s_n$ is f. We say t is a *subterm* of s (written $s \trianglerighteq t$) if either (a) $s = t$, or (b) $s = s'\,s''$ and $s' \trianglerighteq t$ or $s'' \trianglerighteq t$. It is a *proper subterm* of s if $s \neq t$. For a term s, $\mathtt{pos}(s)$ is the set of *positions* in s: $\mathtt{pos}(x) = \mathtt{pos}(\mathsf{f}) = \{\sharp\}$ and $\mathtt{pos}(s\,t) = \{\sharp\} \cup \{1 \cdot u \mid u \in \mathtt{pos}(s)\} \cup \{2 \cdot u \mid u \in \mathtt{pos}(t)\}$. For $p \in \mathtt{pos}(s)$, the subterm $s|_p$ at position p is given by: $s|_\sharp = s$ and $(s_1\,s_2)|_{i \cdot p} = s_i|_p$.

In this paper, we require that for all $\mathsf{f} \in \Sigma$, $\mathtt{ord}(\mathtt{typeOf}(\mathsf{f})) \leq 2$, so w.l.o.g., $\mathsf{f} : (\vec{\iota_1} \Rightarrow \kappa_1) \Rightarrow \cdots \Rightarrow (\vec{\iota_k} \Rightarrow \kappa_k) \Rightarrow \nu_1 \Rightarrow \cdots \Rightarrow \nu_l \Rightarrow \iota$. Hence, in a fully applied term $\mathsf{f}\,s_1 \ldots s_k\,t_1 \ldots t_l$ we say the s_i are the arguments of type-1 and the t_j are the arguments of type-0 for f. We say a term *has type order* k if its type has order k. A *substitution* γ is a type-preserving map from variables to terms such that $\{x \in \mathbb{X} \mid \gamma(x) \neq x\}$ is finite. We extend γ to terms as usual: $x\gamma = \gamma(x)$, $\mathsf{f}\gamma = \mathsf{f}$, and $(s\,t)\gamma = (s\gamma)\,(t\gamma)$. A *context* C is a term with a single occurrence of a variable $\Box$; the term $C[s]$ is obtained by substituting $\Box$ by s.

Finally, we have all the ingredients needed to define rewriting rules and the dynamics they give rise to. A *rewrite rule* $\ell \to r$ is a pair of terms of the same type such that ℓ has a form $\mathsf{f}\,\ell_1 \cdots \ell_m$ with $\mathsf{f} \in \Sigma$, and $\mathtt{vars}(\ell) \supseteq \mathtt{vars}(r)$. It is *left-linear* if no variable occurs more than once in ℓ. A *simply-typed term rewriting system* $(\mathbb{F}, \mathcal{R})$ is a pair consisting of a signature $\mathbb{F}$ and a set of rewrite rules $\mathcal{R}$ over $\mathsf{T}(\mathbb{F}, \mathbb{X})$. In this paper, we require that all rules have *base* type; that is, for each rule $\ell \to r$, the type of ℓ (so also the type of r) is a base type. An STRS is *innermost orthogonal* if all rules are left-linear, and any two distinct rules $\ell_1 \to r_1, \ell_2 \to r_2$ are non-overlapping: there are no substitutions γ and δ such that $\ell_1\gamma = \ell_2\delta$. A *reducible expression* (redex) is a term of the form $\ell\gamma$ for a rule $\ell \to r$ and substitution γ. A term is in *normal form* if none of its subterms is a redex.

The *innermost rewrite relation* induced by $\mathcal{R}$ is defined as follows:

- $\ell\gamma \to_{\mathcal{R}} r\gamma$, if $\ell \to r \in \mathcal{R}$ and $\ell\gamma$ has no proper subterm that is a redex;
- $s\,t \to_{\mathcal{R}} u\,t$, if $s \to_{\mathcal{R}} u$ and $s\,t \to_{\mathcal{R}} s\,u$, if $t \to_{\mathcal{R}} u$.

We write $\to_{\mathcal{R}}^{+}$ for the transitive closure of $\to_{\mathcal{R}}$ and $\to_{\mathcal{R}}^{*}$ for the transitive-reflexive closure. An STRS $\mathcal{R}$ is *innermost terminating* if no infinite rewrite sequence $s \to_{\mathcal{R}} t \to_{\mathcal{R}} \ldots$ exists. It is *innermost confluent* if $s \to_{\mathcal{R}}^{+} t$ and $s \to_{\mathcal{R}}^{+} u$ implies that some v exists with $t \to_{\mathcal{R}}^{*} v$ and $u \to_{\mathcal{R}}^{*} v$. It is well-known that innermost orthogonality implies innermost confluence [MN98]. In this paper, we will typically drop the "innermost" adjective and simply refer to terminating/orthogonal/confluent STRSs.

Example 2.1. Let us consider a simple example of STRS. We start by defining the following signature $\mathbb{F}$. The base types are $\mathbb{B} = \{\mathsf{nat}\}$ and Σ includes $\mathsf{0} : \mathsf{nat}, \mathsf{s} : \mathsf{nat} \Rightarrow \mathsf{nat}$. We add rules for basic operations such as $\mathsf{add}, \mathsf{mult} : \mathsf{nat} \Rightarrow \mathsf{nat} \Rightarrow \mathsf{nat}$ and a higher-order function $\mathsf{fnProd} : (\mathsf{nat} \Rightarrow \mathsf{nat}) \Rightarrow \mathsf{nat} \Rightarrow \mathsf{nat} \Rightarrow \mathsf{nat}$. Finally, the set of rules $\mathcal{R}$ is given by:

$$\begin{array}{rclrcl} \mathsf{add}\,\mathsf{0}\,y & \to & y & \mathsf{add}\,(\mathsf{s}\,x)\,y & \to & \mathsf{s}\,(\mathsf{add}\,x\,y) \\ \mathsf{mult}\,\mathsf{0}\,y & \to & \mathsf{0} & \mathsf{mult}\,(\mathsf{s}\,x)\,y & \to & \mathsf{add}\,y\,(\mathsf{mult}\,x\,y) \\ \mathsf{fnProd}\,F\,\mathsf{0}\,y & \to & y & \mathsf{fnProd}\,F\,(\mathsf{s}\,x)\,y & \to & \mathsf{fnProd}\,F\,x\,(\mathsf{mult}\,y\,(F\,x)) \end{array}$$

Notice that this system is left-linear and non-overlapping; hence, it is innermost orthogonal, and therefore innermost confluent.

Hereafter, we write $\ulcorner \mathsf{n} \urcorner$ for the term $\mathsf{s}\,(\mathsf{s}\,(\ldots\,\mathsf{0}\ldots))$ with n ss.

Remark 2.2. In this paper, we have chosen to limit interest to second-order systems with base-type rules. The restriction to base-type rules is not so significant — since one may simply pad both sides of the rules with fresh variables until they have base type — but the order limitation is arguably more restrictive.

We impose this restriction because our primary goal is to identify a convenient class of term rewriting systems that characterizes $\mathtt{BFF}_2$ — *not* to find the most general class of term rewriting systems that may do so. Indeed, we have already derived a variation of the results in this work for call-by-value systems of unrestricted type order following the definitions of cost–size interpretation in [KV23] (see [Val24, Chapter 6]). Hence, we have every reason to believe that the type restriction could be dropped also in the current innermost setting, provided the "main" function F that we will identify in Definition 4.3 remains second-order, i.e., the oracle functions considered are still first-order in such an envisioned extension. We have not done so in this work because extending the definition of "interpretation" to higher-order innermost systems would add a lot of technical burden (as it also does for call-by-value reduction), and inhibit understanding of the methodology, for arguably little expressivity gain.

We also *suspect* that the results will extend if lambda-abstraction and beta reduction are permitted, but we have not confirmed this as it would require a larger change, primarily in the construction of Section 5.3. This is also not a major restriction, since abstractions in the right-hand side of rules can be replaced by fresh function symbols.

2.2. **Basic Feasible Functionals.** We assume familiarity with Turing Machines and basic notions of computability and complexity theory [AB09]. In this paper, we consider *deterministic multi-tape Turing Machines*. Those are, conceptually, machines consisting of a finite set of states, one or more (but a fixed number of) right-infinite *tapes* divided into cells. Each tape is equipped with a tape head that scans the symbols on the tape's cells and may write on it. The head can move to the left or right. A k-ary *Oracle Turing Machine* (OTM) is a deterministic multi-tape Turing Machine with at least $2k+1$ tapes: one main tape for (input/output), k designated *query* tapes ($\texttt{querytape}_i$ for $1 \leq i \leq k$) and k designated *answer* tapes ($\texttt{answertape}_i$). It also has k distinct *query states* q_i and k *answer states* a_i.

Let $W = \{0,1\}^*$. A computation with a k-ary OTM M requires k fixed *oracle functions* $f_1, \ldots, f_k : W \longrightarrow W$. We write $M_{\vec{f}}$ to denote a run of M with these functions. A run of $M_{\vec{f}}$ on w starts with w written on the main tape. It ends when the machine halts, and yields the word that is written on the main tape as output. As usual, we only consider machines that halt on all inputs. The computation proceeds as usual for non-query states. To query the value of f_i on w, the machine writes w on the corresponding query tape $\texttt{tape}_{2*i}$ and enters the query state q_i. Then, *in one step*, it transitions to the answer state a_i as follows:

(a) the query value w written in $\texttt{querytape}_i$, the query tape for f_i, is read;
(b) the contents of $\texttt{answertape}_i$, are changed to $f_i(w)$;
(c) the query value w is erased from $\texttt{querytape}_i$; and
(d) the head of $\texttt{answertape}_i$ is moved to its first symbol.

The *running time* of $M_{\vec{f}}$ on w is the number of steps used in the computation.

We say x is of type-0 whenever $x \in W$. A *type-1 function* is a mapping in $W \longrightarrow W$. A *type-2 functional* of rank (k,l) is a mapping in $(W \longrightarrow W)^k \longrightarrow W^l \longrightarrow W$.

Definition 2.3. We say an OTM M **computes** a type-2 functional Ψ of rank (k,l) iff for all type-1 functions $f_1, \ldots, f_k$ and $x_1, \ldots, x_l \in W$, whenever $M_{f_1,\ldots,f_k}$ is started with $x_1, \ldots, x_l$ written on its main tape (separated by blanks), it halts with $\Psi(f_1, \ldots, f_k, x_1, \ldots, x_l)$ written on its main tape.

Definition 2.4. Let $\{F_1, \ldots, F_k\}$ be a set of *type-1 variables* and $\{x_1, \ldots, x_l\}$ a set of *type-0 variables*. The set $\texttt{Pol}^2_{\mathbb{N}}[F_1, \ldots, F_k; x_1, \ldots, x_l]$ of **second-order polynomials** over $\mathbb{N}$ with indeterminates $F_1, \ldots, F_k, x_1, \ldots, x_l$ is generated by:

$$P, Q \coloneqq n \mid x \mid P + Q \mid P * Q \mid F(Q)$$

where $n \in \mathbb{N}$, $x \in \{x_1, \ldots, x_l\}$, and $F \in \{F_1, \ldots, F_k\}$.

Notice that such polynomial expressions can be naturally viewed as type-2 functionals, e.g., $P(F,x) = 3 * F(x) + x$ is a type-2 functional of rank $(1,1)$.

Given $w \in W$, we write $|w|$ for its length and define the length $|f|$ of $f : W \longrightarrow W$ as $|f| = \boldsymbol{\lambda} n. \max_{|y| \leq n} |f(y)|$. This allows us to define $\mathtt{BFF}_2$ as the class of functionals computable by OTMs with running time bounded by a second-order polynomial:

Definition 2.5. A type-2 functional Ψ is in $\mathtt{BFF}_2$ iff there exist an OTM M and a second-order polynomial P such that M computes Ψ and for all $\vec{f}$ and $\vec{x}$: the running time of $M_{f_1,\dots,f_k}$ on $x_1, \dots, x_l$ is at most $P(|f_1|, \dots, |f_k|, |x_1|, \dots, |x_l|)$.

3. Cost–Size Interpretations

In this section we define cost–size interpretations for the syntax of types and terms in the STRS format. The interpretations we develop differ from those in [KV23] in two key aspects: here we limit ourselves to second-order systems and the rewriting strategy is innermost instead of call-by-value. As a consequence, the notion of *cost* here is explicitly expressed as a function $\mathsf{cost}(\cdot)$ that inductively computes the total cost of reduction. Furthermore, these changes require that we prove a higher-order version of the innermost compatibility theorem.

3.1. The Interpretation of Types and Terms.

In cost–size interpretations, each base-type term is mapped to two different components: a *size* and a *cost*. Intuitively, the size component imposes a restriction on the shape of the normal form of a term; for example, by limiting the total number of function symbols in the normal form, or a weighted total where some symbols are weighed more heavily than others. Meanwhile, the cost component is an integer that bounds the number of steps needed to reach a normal form. By separating these two notions, we can obtain a more fine-grained complexity bound compared to the classical interpretation methods that map to for instance natural numbers [BL16, BCMT01, HP20]. For non-base type terms, the cost and size interpretations are represented by *functions*.

In this paper both the cost and size components must be elements of quasi-ordered sets, which we define formally as follows.

Size interpretation. For sets A and B, we write $A \longrightarrow B$ for the set of functions from A to B. A *quasi-ordered set* $(A, \sqsupseteq)$ consists of a nonempty set A and a reflexive and transitive relation $\sqsupseteq$ on A. For quasi-ordered sets $(A_1, \sqsupseteq_1)$ and $(A_2, \sqsupseteq_2)$, we write $A_1 \Longrightarrow A_2$ for the set of functions $f \in A_1 \longrightarrow A_2$ such that $f(x) \sqsupseteq_2 f(y)$ whenever $x \sqsupseteq_1 y$, i.e., $A_1 \Longrightarrow A_2$ is the space of *weakly monotonic* functions: functions that preserve the quasi-ordering.

For every $\iota \in \mathbb{B}$, let a quasi-ordered set $(\mathcal{S}_\iota, \sqsupseteq_\iota)$ be given. We extend this to $\mathbb{T}(\mathbb{B})$ by defining $\mathcal{S}_{\sigma \Rightarrow \tau} = (\mathcal{S}_\sigma \Longrightarrow \mathcal{S}_\tau, \sqsupseteq_{\sigma \Rightarrow \tau})$ where $f \sqsupseteq_{\sigma \Rightarrow \tau} g$ iff $f(x) \sqsupseteq_\tau g(x)$ for all $x \in \mathcal{S}_\sigma$. So $\mathcal{S}_{\sigma \Rightarrow \tau}$ is a set of weakly monotonic functionals, using pointwise comparison.

We assume given a fixed *size interpretation function* $\mathcal{J}^{\mathsf{s}}$, which maps each $\mathsf{f} \in \Sigma$ to some $\mathcal{J}^{\mathsf{s}}_{\mathsf{f}} \in \mathcal{S}_{\mathtt{typeOf(f)}}$. If we are additionally given a valuation α that maps each $x \in \mathbb{X}_\sigma$ to $\mathcal{S}_\sigma$, we can map each term $s : \sigma$ to an element of $\mathcal{S}_\sigma$ naturally as follows: (a) $[\![x]\!]^{\mathsf{s}}_\alpha = \alpha(x)$; (b) $[\![\mathsf{f}]\!]^{\mathsf{s}}_\alpha = \mathcal{J}^{\mathsf{s}}_{\mathsf{f}}$; (c) $[\![s\,t]\!]^{\mathsf{s}}_\alpha = [\![s]\!]^{\mathsf{s}}_\alpha([\![t]\!]^{\mathsf{s}}_\alpha)$.

(Note that the superscript $^{\mathsf{s}}$ always indicates *size*, while we will use $^{\mathsf{c}}$ for *cost*.)

Cost interpretation. For every type σ with $\mathtt{ord}(\sigma) \leq 2$, we define $\mathcal{C}_\sigma$ as follows:

(a) $\mathcal{C}_\kappa = \mathbb{N}$ for $\kappa \in \mathbb{B}$;
(b) $\mathcal{C}_{\iota \Rightarrow \tau} = \mathcal{S}_\iota \Longrightarrow \mathcal{C}_\tau$ for $\iota \in \mathbb{B}$, and
(c) $\mathcal{C}_{\sigma \Rightarrow \tau} = \mathcal{C}_\sigma \Longrightarrow \mathcal{S}_\sigma \Longrightarrow \mathcal{C}_\tau$ if $\mathtt{ord}(\sigma) = 1$.

For the weak monotonicity, we again use pointwise comparison: for $f, g \in \mathcal{C}_{\sigma \Rightarrow \tau}$ we say $f \geq g$ if $f(x) \geq g(x)$ for every $x \in \mathcal{C}_\sigma$.

We assume given a fixed *cost interpretation function* $\mathcal{J}^{\mathsf{c}}$, which maps each $\mathsf{f} \in \Sigma$ to some $\mathcal{J}^{\mathsf{c}}_{\mathsf{f}} \in \mathcal{C}_{\mathtt{typeOf}(\mathsf{f})}$; and for each type σ with $\mathtt{ord}(\sigma) \leq 1$, we assume given *valuations* $\alpha : \mathbb{X}_\sigma \longrightarrow \mathcal{S}_\sigma$ and $\zeta : \mathbb{X}_\sigma \longrightarrow \mathcal{C}_\sigma$. We then define, for terms of type order ≤ 1:

$$\begin{aligned} [\![x\, s_1 \cdots s_n]\!]^{\mathsf{c}}_{\alpha,\zeta} &= \zeta(x)([\![s_1]\!]^{\mathsf{s}}_\alpha, \ldots, [\![s_n]\!]^{\mathsf{s}}_\alpha) \\ [\![\mathsf{f}\, s_1 \cdots s_k\, t_1 \cdots t_n]\!]^{\mathsf{c}}_{\alpha,\zeta} &= \mathcal{J}^{\mathsf{c}}_{\mathsf{f}}([\![s_1]\!]^{\mathsf{c}}_{\alpha,\zeta}, [\![s_1]\!]^{\mathsf{s}}_\alpha, \ldots, [\![s_k]\!]^{\mathsf{c}}_{\alpha,\zeta}, [\![s_k]\!]^{\mathsf{s}}_\alpha, [\![t_1]\!]^{\mathsf{s}}_\alpha, \ldots, [\![t_n]\!]^{\mathsf{s}}_\alpha) \end{aligned}$$

This is well-defined under our assumptions that all variables have a type of order 0 or 1 (so the arguments to a variable have type order 0), and that we can write $\mathsf{f} : (\vec{\iota_1} \Rightarrow \kappa_1) \Rightarrow \cdots \Rightarrow (\vec{\iota_k} \Rightarrow \kappa_k) \Rightarrow \nu_1 \Rightarrow \cdots \Rightarrow \nu_l \Rightarrow \iota$ for each function symbol f. Observe also that $\mathcal{J}^{\mathsf{c}}_{\mathsf{f}}$, when fully applied, maps to a natural number. As the left- and right-hand sides of rules have base type, this definition assigns a natural number as the cost interpretation for both sides of each rule.

Intuitively, for a ground base-type term $\mathsf{f}\ s_1 \cdots s_k\, t_1 \cdots t_n$, $[\![\mathsf{f}\, s_1 \cdots s_k\, t_1 \cdots t_n]\!]^{\mathsf{c}}_{\alpha,\zeta}$ expresses the cost of the reduction to normal form *not including* the cost of first normalising each s_i and t_i. To compute the full reduction cost, we let

$$\mathsf{cost}(s)_{\alpha,\zeta} = \sum\{[\![t]\!]^{\mathsf{c}}_{\alpha,\zeta} \mid s \trianglerighteq t \text{ and } t \text{ is a non-variable term of base type}\}$$

In addition, in our proofs it is very convenient to have a second cost measure:

$$\mathsf{cost}^\star(s)_{\alpha,\zeta} = \sum\{[\![t]\!]^{\mathsf{c}}_{\alpha,\zeta} \mid s \trianglerighteq t \text{ and } t \text{ is a non-variable base-type term not in normal form}\}$$

The difference between cost and $\mathsf{cost}^\star$ is that subterms that are already in normal form may still contribute a non-zero part to the cost measure of a term, but not to $\mathsf{cost}^\star$.

Compatibility. A *cost–size interpretation* $\mathcal{F}$ for a second order signature $\mathbb{F} = (\mathbb{B}, \Sigma, \mathtt{typeOf})$ is a choice of a quasi-ordered set $\mathcal{S}_\iota$, for each $\iota \in \mathbb{B}$, along with cost- and size-interpretation functions $\mathcal{J}^{\mathsf{c}}$ and $\mathcal{J}^{\mathsf{s}}$ defined as above. Let $(\mathbb{F}, \mathcal{R})$ be an STRS over $\mathbb{F}$.

Definition 3.1. We say $(\mathbb{F}, \mathcal{R})$ is *compatible* with a cost–size interpretation if for any valuations α and ζ, we have (a) $[\![\ell]\!]^{\mathsf{c}}_{\alpha,\zeta} > \mathsf{cost}(r)_{\alpha,\zeta}$ and (b) $[\![\ell]\!]^{\mathsf{s}}_\alpha \sqsupseteq [\![r]\!]^{\mathsf{s}}_\alpha$, for all rules $\ell \to r$ in $\mathcal{R}$. In this case we say such cost–size interpretation *orients* all rules in $\mathcal{R}$.

As in all notions of interpretation where the underlying domain is that of the natural numbers, the compatibility check is, in general, bound to be undecidable. This is the case even if sound but complete techniques exist. In any case, a study about the possibility of automating our technique is outside the scope of this paper and is thus left to future work.

To avoid heavy notation, we will often omit the valuations α and ζ from subscripts when quantifying over them, and keep the quantification implicit. For example, instead of writing

$$\forall \alpha, \zeta.\ [\![\mathsf{fnProd}\, F\, (\mathsf{s}\, x)\, y]\!]^{\mathsf{c}}_{\alpha,\zeta} > \mathsf{cost}(\mathsf{fnProd}\, F\, x\, (\mathsf{mult}\, y\, (F\, x)))_{\alpha,\zeta}$$

we will use:

$$[\![\mathsf{fnProd}\, F\, (\mathsf{s}\, x)\, y]\!]^{\mathsf{c}} > \mathsf{cost}(\mathsf{fnProd}\, F\, x\, (\mathsf{mult}\, y\, (F\, x)))$$

When we do this, we will also leave α and ζ out of concreate interpretations; for example, instead of

$$[\![\mathsf{fnProd}\,F\,x\,y]\!]^{\mathsf{c}}_{\alpha,\zeta} = 3*\alpha(x)*\alpha(y)*\max(\alpha(F)(\alpha(x)),1)^{\alpha(x)+1}+\alpha(x)*\zeta(F)(\alpha(x))+2*\alpha(x)+1$$

we will use:

$$[\![\mathsf{fnProd}\,F\,x\,y]\!]^{\mathsf{c}}_{\alpha,\zeta} = 3*x*y*\max(F^{\mathsf{s}}(x),1)^{x+1}+x*F^{\mathsf{c}}(x)+2*x+1$$

Where $\alpha(x)$ is simply denoted as x for type-0 variables, and as x^{s} for type-1 variables; $\zeta(x)$ is always denoted as x^{c} (note that, when considering the cost interpretation of rules, $\zeta(x)$ only occurs for type-1 variables).

Example 3.2. Let us consider the signature of the STRS given in Example 2.1. We can interpret the base type nat as the natural quasi-order set $\mathcal{S}_{\mathsf{nat}} = (\mathbb{N},\geq)$. We need to give interpretations for the constructors of nat. Let us start by size.

$$\mathcal{J}^{\mathsf{s}}_{\mathsf{0}} = 0 \qquad\qquad \mathcal{J}^{\mathsf{s}}_{\mathsf{s}} = \boldsymbol{\lambda} x.x+1$$

This gives us $[\![\ulcorner \mathsf{n} \urcorner]\!]^{\mathsf{s}} = n$ for all $n \in \mathbb{N}$. So intuitively our interpretation is counting how many s symbols there are in the unary representation of a natural number, and setting this as its size. For the cost measure we do as follows.

$$\mathcal{J}^{\mathsf{c}}_{\mathsf{0}} = 0 \qquad\qquad \mathcal{J}^{\mathsf{c}}_{\mathsf{s}} = \boldsymbol{\lambda} x.0$$

This gives us $[\![\ulcorner \mathsf{n} \urcorner]\!]^{\mathsf{c}} = \mathsf{cost}(n) = 0$, which makes sense since the base type nat is intended to represent data values, from which no computation can be started. Here we see a first contrast with the classical interpretation method. By splitting cost and size we can faithfully encode complexity information into the interpretation.

Next, we give size and cost interpretation functions for add and mult. So we let $\mathcal{J}^{\mathsf{s}}_{\mathsf{add}} = \boldsymbol{\lambda} xy.x+y$ and $\mathcal{J}^{\mathsf{s}}_{\mathsf{mult}} = \boldsymbol{\lambda} xy.x*y$; then indeed $[\![\ell]\!]^{\mathsf{s}} \geq [\![r]\!]^{\mathsf{s}}$ for the first four rules of Example 2.1 (e.g., $[\![\mathsf{mult}\,(\mathsf{s}\,x)\,y]\!]^{\mathsf{s}} = (x+1)*y \geq y+(x*y) = [\![\mathsf{add}\,y\,(\mathsf{mult}\,x\,y)]\!]^{\mathsf{s}}$). For the cost interpretation functions, note that both add and mult are type-1, so we must choose functions that take the *sizes* of their arguments as input, and return a cost. Here, we choose $\mathcal{J}^{\mathsf{c}}_{\mathsf{add}} = \boldsymbol{\lambda} xy.x+1$ and $\mathcal{J}^{\mathsf{c}}_{\mathsf{mult}} = \boldsymbol{\lambda} xy.x*y+2*x+1$. To see why this makes sense, consider arguments in *normal form*: e.g., reducing $\mathsf{add}\ (\mathsf{s}^n\ \mathsf{0})\ (\mathsf{s}^m\ \mathsf{0})$ to normal form takes $n+1$ steps. We indeed have the required inequality $[\![\ell]\!]^{\mathsf{c}} > \mathsf{cost}(r)$ for the first four rules; for example:

$$\begin{array}{rcllcl}
[\![\mathsf{mult}\,(\mathsf{s}\,x)\,y]\!]^{\mathsf{c}} & = & [\![\mathsf{s}\,x]\!]^{\mathsf{s}} * [\![y]\!]^{\mathsf{s}} + 2*[\![\mathsf{s}\,x]\!]^{\mathsf{s}}+1 & & & \\
 & = & (x+1)*y+2*(x+1)+1 & & = & x*y+2*x+y+3 \\
 & > & & & & \\
\mathsf{cost}(\mathsf{add}\,y\,(\mathsf{mult}\,x\,y)) & = & [\![\mathsf{add}\,y\,(\mathsf{mult}\,x\,y)]\!]^{\mathsf{c}} + [\![\mathsf{mult}\,x\,y]\!]^{\mathsf{c}} & & & \\
 & = & (y+1) \quad + \quad (x*y+2*x+1) & & = & x*y+x*x+y+2
\end{array}$$

Regarding fnProd: note that the cost of evaluating a term $\mathsf{fnProd}\ F\ n\ m$ depends not only on the sizes of the arguments F, n and m, but also on the *behaviour* of F: if F is a function that computes its result in linear time in the size of the input, then $\mathsf{fnProd}\ F\ n\ m$ will reach a normal form much faster than if F operates in exponential time. The computation time is also affected by the size of the output that fnProd produces, as this is fed into mult. This is why $\mathcal{J}^{\mathsf{c}}_{\mathsf{fnProd}}$ takes both a cost function and a size function as argument. We can orient both rules by choosing $\mathcal{J}^{\mathsf{s}}_{\mathsf{fnProd}} = \boldsymbol{\lambda} Fxy.y*\max(F(x),1)^x$ and $\mathcal{J}^{\mathsf{c}}_{\mathsf{fnProd}} = \boldsymbol{\lambda} F^{\mathsf{c}}F^{\mathsf{s}}xy.3*x*y*\max(F^{\mathsf{s}}(x),1)^{x+1}+x*F^{\mathsf{c}}(x)+2*x+1$.

Notice that $\mathcal{J}^{\mathsf{c}}_{\mathsf{fnProd}}$ is not polynomial, but that is allowed in the general case.

Example 3.3. Consider an STRS with base types $\mathbb{B} = \{\mathsf{nat}\}$, function symbols $\mathsf{0} : \mathsf{nat}, \mathsf{s} : \mathsf{nat} \Rightarrow \mathsf{nat}, \mathsf{minus} : \mathsf{nat} \Rightarrow \mathsf{nat} \Rightarrow \mathsf{nat}$, and rules:

$$\mathsf{minus}\ (\mathsf{s}\ x)\ (\mathsf{s}\ y) \ \to\ \mathsf{minus}\ x\ y \qquad\qquad \mathsf{minus}\ x\ \mathsf{0} \ \to\ x$$

If we choose $\mathcal{J}^{\mathsf{s}}_{\mathsf{0}}$, $\mathcal{J}^{\mathsf{s}}_{\mathsf{s}}$, $\mathcal{J}^{\mathsf{c}}_{\mathsf{0}}$ and $\mathcal{J}^{\mathsf{c}}_{\mathsf{s}}$ like in Example 3.2, and $\mathcal{J}^{\mathsf{c}}_{\mathsf{minus}} = \boldsymbol{\lambda} xy.y$, then we have $\mathsf{cost}(\mathsf{minus}\ \mathsf{0}\ (\mathsf{s}\ \mathsf{0})) = 1$, while $\mathsf{cost}^\star(\mathsf{minus}\ \mathsf{0}\ (\mathsf{s}\ \mathsf{0})) = 0$. This is because $\mathsf{minus}\ \mathsf{0}\ (\mathsf{s}\ \mathsf{0})$ is in normal form, even though it is not fully built from constructors.

3.2. **The Compatibility Theorem.** We now prove the Innermost Compatibility Theorem for STRSs. The proof is analogous to that in [KV23] with some adaptations to the definitions of Section 3.1 which are possible here since our terms are applicative and second-order only; we also make some adaptations to the evaluation strategy being innermost rather than call-by-value, in particular through the use of the $\mathsf{cost}^\star(\cdot)$ function.

Theorem 3.4 (Innermost Compatibility)**.** Suppose $(\mathbb{F}, \mathcal{R})$ is an STRS compatible with a cost–size interpretation $\mathcal{F}$ (following Definition 3.1). Then for any valuations α and ζ we have $\mathsf{cost}^\star(s)_{\alpha,\zeta} > \mathsf{cost}^\star(t)_{\alpha,\zeta}$ and $[\![s]\!]^{\mathsf{s}}_{\alpha} \sqsupseteq [\![t]\!]^{\mathsf{s}}_{\alpha}$ whenever $s \to_{\mathcal{R}} t$.

Hence, if we find a suitable cost–size interpretation, which orients all the rules, then we have a *bound* on the derivation height of any given term: that is, the number of reduction steps we need to take to reduce any given term to normal form. For example, suppose $s_0 \to_{\mathcal{R}} s_1 \to_{\mathcal{R}} \dots \to_{\mathcal{R}} s_n$. Let α and ζ map all variables either to 0 or to a constant function $\boldsymbol{\lambda}\vec{x}.0$. Then $\mathsf{cost}^\star(s_0)_{\alpha,\zeta} \geq \mathsf{cost}^\star(s_1)_{\alpha,\zeta} + 1 \geq \dots \geq \mathsf{cost}^\star(s_n)_{\alpha,\zeta} + n$ — and therefore, necessarily $\mathsf{cost}^\star(s_0)_{\alpha,\zeta} \geq n$. This holds whether s_0 has base type or not, since $\mathsf{cost}^\star(\cdot)$ always maps to a natural number. Observing that always $\mathsf{cost}(s)_{\alpha,\zeta} \geq \mathsf{cost}^\star(s)_{\alpha,\zeta}$:

Corollary 3.5. Suppose $(\mathbb{F}, \mathcal{R})$ is an STRS compatible with a cost–size interpretation $\mathcal{F}$, and let s be a term with derivation height n. Then for any valuations α and ζ, $\mathsf{cost}(s)_{\alpha,\zeta} \geq n$.

In order to prove Theorem 3.4, we first stablish some useful lemmas. Recall that in this paper, all rules are of base type, i.e., they are fully applied. Since reduction is innermost, a rule may only be fired if the matching substitution (i.e., the substitution γ on the base case $\ell\gamma \to r\gamma$), maps all variables to irreducible terms. So in the lemmas below, without loss of generality, we restrict ourselves to this type of substitutions and notice that $\mathsf{cost}^\star(x\gamma) = 0$ for any variable x.

Lemma 3.6. For all terms $s\,t$ with t of base type: $[\![s\,t]\!]^{\mathsf{s}}_{\alpha} = [\![s]\!]^{\mathsf{s}}_{\alpha}([\![t]\!]^{\mathsf{s}}_{\alpha})$ and $[\![s\,t]\!]^{\mathsf{c}}_{\alpha,\zeta} = [\![s]\!]^{\mathsf{c}}_{\alpha,\zeta}([\![t]\!]^{\mathsf{s}}_{\alpha})$ for all α, ζ.

Proof. The former statement holds by definition. The latter holds by a straightforward case analysis on the form of s: it holds both if $s = x\,s_1 \cdots s_n$ and $s = \mathsf{f}\,s_1 \cdots s_k\,t_1 \cdots t_n$. For the f case, recall that we fixed $\mathsf{f} : (\vec{\iota_1} \Rightarrow \kappa_1) \Rightarrow \dots \Rightarrow (\vec{\iota_k} \Rightarrow \kappa_k) \Rightarrow \nu_1 \Rightarrow \dots \Rightarrow \nu_l \Rightarrow \iota$ as the general type for function symbols f, and since t has base type, all higher-type arguments to f are necessarily already supplied. Hence, $[\![s]\!]^{\mathsf{c}}_{\alpha,\zeta}$ is indeed well-defined. ☐

Given a valuation α and substitution γ, we denote the γ-extention of α by α^γ as the valuation defined by $\alpha^\gamma(x) = [\![x\gamma]\!]^{\mathsf{s}}_{\alpha}$. Let us start with some substitution lemmata.

Lemma 3.7. Let γ be a substitution mapping all variables to irreducible terms and α be a valuation. Then, for any term s, $[\![s\gamma]\!]^{\mathsf{s}}_{\alpha} = [\![s]\!]^{\mathsf{s}}_{\alpha^\gamma}$.

Proof. By induction on the structure of s.

- If s is a variable, we have $[\![x\gamma]\!]^{\mathsf{s}}_{\alpha} = \alpha^{\gamma}(x) = [\![x]\!]^{\mathsf{s}}_{\alpha^{\gamma}}$.
- If $s = t\,u$ is an application, we have

$$\begin{aligned}[\![(t\,u)\gamma]\!]^{\mathsf{s}} &= [\![t\gamma]\!]^{\mathsf{s}}_{\alpha}([\![u\gamma]\!]^{\mathsf{s}}_{\alpha}) \\ &\overset{IH}{=} [\![t]\!]^{\mathsf{s}}_{\alpha^{\gamma}}([\![u]\!]^{\mathsf{s}}_{\alpha^{\gamma}}) = [\![t\,u]\!]^{\mathsf{s}}_{\alpha^{\gamma}}\end{aligned}$$

□

We can now prove the *size* part of Theorem 3.4 which is stated as the lemma below.

Lemma 3.8. Let $(\mathbb{F}, \mathcal{R})$ be an STRS compatible with a cost–size interpretation following Definition 3.1, and let s, t be terms of the same type such that $s \to_{\mathcal{R}} t$. Then $[\![s]\!]^{\mathsf{s}}_{\alpha} \sqsupseteq [\![t]\!]^{\mathsf{s}}_{\alpha}$ holds for all α.

Proof. By induction on the form of s.

In the base case, we have $s \to_{\mathcal{R}} t$ by $\ell\gamma \to r\gamma$. Then we combine the substitution lemma (Lemma 3.7) with the compatibility requirement for size, i.e., $[\![\ell]\!]^{\mathsf{s}}_{\alpha} \sqsupseteq [\![r]\!]^{\mathsf{s}}_{\alpha}$, as follows:

$$[\![\ell\gamma]\!]^{\mathsf{s}}_{\alpha} = [\![\ell]\!]^{\mathsf{s}}_{\alpha^{\gamma}} \sqsupseteq [\![r]\!]^{\mathsf{s}}_{\alpha^{\gamma}} = [\![r\gamma]\!]^{\mathsf{s}}_{\alpha}$$

In the application case, $s = s_1\,s_2$ and either $t = t_1\,s_2$ with $s_1 \to_{\mathcal{R}} t_1$ or $t = s_1\,t_2$ with $s_2 \to_{\mathcal{R}} t_2$. In the first case, $[\![s_1]\!]^{\mathsf{s}}_{\alpha} \sqsupseteq [\![t_1]\!]^{\mathsf{s}}_{\alpha}$ by the induction hypothesis, which by definition gives $[\![s]\!]^{\mathsf{s}}_{\alpha} = [\![s_1]\!]^{\mathsf{s}}_{\alpha}([\![s_2]\!]^{\mathsf{s}}_{\alpha}) \sqsupseteq [\![t_1]\!]^{\mathsf{s}}_{\alpha}([\![s_2]\!]^{\mathsf{s}}_{\alpha}) = [\![t]\!]^{\mathsf{s}}_{\alpha}$. In the second case, $[\![s_2]\!]^{\mathsf{s}}_{\alpha} \sqsupseteq [\![t_2]\!]^{\mathsf{s}}_{\alpha}$ by the induction hypothesis and $[\![s]\!]^{\mathsf{s}}_{\alpha} = [\![s_1]\!]^{\mathsf{s}}_{\alpha}([\![s_2]\!]^{\mathsf{s}}_{\alpha}) \sqsupseteq [\![s_1]\!]^{\mathsf{s}}_{\alpha}([\![t_2]\!]^{\mathsf{s}}_{\alpha}) = [\![t]\!]^{\mathsf{s}}_{\alpha}$ because by definition, $[\![s_1]\!]^{\mathsf{s}}_{\alpha}$ is a *weakly monotonic* function. □

Let us move on to cost versions of substitution lemmata. First, notice that we cannot directly define a γ-extention for cost valuations. Indeed, $[\![\cdot]\!]^{\mathsf{c}}_{\alpha,\zeta}$ also depends on a size valuation α. So given a size valuation α, we write ζ^{γ}_{α} to denote the valuation ζ^{γ}_{α} with $\zeta^{\gamma}_{\alpha}(x) = [\![\gamma(x)]\!]^{\mathsf{c}}_{\alpha,\zeta}$.

Lemma 3.9. Given cost–size valuations α, γ and a term s of type order at most 1. Then $[\![s\gamma]\!]^{\mathsf{c}}_{\alpha,\zeta} = [\![s]\!]^{\mathsf{c}}_{\alpha^{\gamma},\zeta^{\gamma}_{\alpha}}$.

Proof. By induction on the size of s. We consider two cases:

- For the first case, we get $s = x\,s_1 \dots s_n$, and
  - If $n = 0$, we have $[\![x\gamma]\!]^{\mathsf{c}}_{\alpha,\zeta} = \zeta^{\gamma}_{\alpha}(x) = [\![x]\!]^{\mathsf{c}}_{\alpha^{\gamma},\zeta^{\gamma}_{\alpha}}$ by definition.
  - If $n > 0$, we have

$$\begin{aligned}[\![(x\,s_1 \dots s_n)\gamma]\!]^{\mathsf{c}}_{\alpha,\zeta} &= [\![(x\gamma)\,(s_1\gamma) \dots (s_n\gamma)]\!]^{\mathsf{c}}_{\alpha,\zeta} \\ &\overset{\text{Lemma 3.6}}{=} [\![x\gamma]\!]^{\mathsf{c}}_{\alpha,\zeta}([\![s_1\gamma]\!]^{\mathsf{s}}_{\alpha}, \dots, [\![s_n\gamma]\!]^{\mathsf{s}}_{\alpha}) \\ &\overset{\text{Lemma 3.7}}{=} [\![x\gamma]\!]^{\mathsf{c}}_{\alpha,\zeta}([\![s_1]\!]^{\mathsf{s}}_{\alpha^{\gamma}}, \dots, [\![s_n]\!]^{\mathsf{s}}_{\alpha^{\gamma}}) \\ &= \zeta^{\gamma}_{\alpha}(x)([\![s_1]\!]^{\mathsf{s}}_{\alpha^{\gamma}}, \dots, [\![s_n]\!]^{\mathsf{s}}_{\alpha^{\gamma}}) \\ &= [\![(x\,s_1 \dots s_n)]\!]^{\mathsf{c}}_{\alpha^{\gamma},\zeta^{\gamma}_{\alpha}}\end{aligned}$$

- For the second case we have $s = \mathsf{f}\,s_1 \dots s_k\,t_1 \dots t_n$. Recall that we fixed $\mathsf{f} : (\vec{\iota_1} \Rightarrow \kappa_1) \Rightarrow \dots \Rightarrow (\vec{\iota_k} \Rightarrow \kappa_k) \Rightarrow \nu_1 \Rightarrow \dots \Rightarrow \nu_l \Rightarrow \iota$ as the general type for f. Hence, since we consider s of type order at most 1, f must take at least k arguments, and $0 \leq n \leq l$.

$$\begin{aligned}&[\![(\mathsf{f}\,s_1 \dots s_k\,t_1 \dots t_n)\gamma]\!]^{\mathsf{c}}_{\alpha,\zeta} \\ &= [\![\mathsf{f}\,(s_1\gamma) \dots (s_k\gamma)\,(t_1\gamma) \dots (t_n\gamma)]\!]^{\mathsf{c}}_{\alpha,\zeta} \\ &= \mathcal{J}^{\mathsf{c}}_{\mathsf{f}}([\![s_1\gamma]\!]^{\mathsf{c}}_{\alpha,\zeta}, [\![s_1\gamma]\!]^{\mathsf{s}}_{\alpha}, \dots, [\![s_k\gamma]\!]^{\mathsf{c}}_{\alpha,\zeta}, [\![s_k\gamma]\!]^{\mathsf{s}}_{\alpha}, [\![t_1\gamma]\!]^{\mathsf{s}}_{\alpha}, \dots, [\![t_n\gamma]\!]^{\mathsf{s}}_{\alpha})\end{aligned}$$

$$\overset{\text{IH, Lemma 3.7}}{=} \mathcal{J}_{\mathsf{f}}^{\mathsf{c}}([\![s_1]\!]^{\mathsf{c}}_{\alpha^\gamma,\zeta^\gamma_\alpha}, [\![s_1]\!]^{\mathsf{s}}_{\alpha^\gamma}, \dots, [\![s_k]\!]^{\mathsf{c}}_{\alpha^\gamma,\zeta^\gamma_\alpha}, [\![s_k]\!]^{\mathsf{s}}_{\alpha^\gamma}, [\![t_1]\!]^{\mathsf{s}}_{\alpha^\gamma}, \dots, [\![t_n]\!]^{\mathsf{s}}_{\alpha^\gamma})$$
$$= [\![\mathsf{f}\, s_1 \dots s_k\, t_1 \dots t_n]\!]^{\mathsf{c}}_{\alpha^\gamma,\zeta^\gamma_\alpha} \qquad \square$$

Next, we connect the relationship between the two cost functions we defined.

Lemma 3.10. For any term s of type order 0 or 1, and any substitution γ such that all $\gamma(x)$ are in normal form, we have that $\mathsf{cost}(s)_{\alpha^\gamma,\zeta^\gamma_\alpha} \geq \mathsf{cost}^\star(s\gamma)_{\alpha,\zeta}$.

Proof. We again consider two cases:

- For the first case, let $s = x\, s_1 \dots s_n$. If $n = 0$ then $\mathsf{cost}(x)_{\alpha^\gamma,\zeta^\gamma_\alpha} = 0$ by definition, and since we assumed that $\gamma(x)$ is in normal form, also $\mathsf{cost}^\star(x\gamma)_{\alpha,\zeta} = 0$. If $n > 0$ and s has base type, then $\mathsf{cost}(s)_{\alpha^\gamma,\zeta^\gamma_\alpha} = \zeta^\gamma_\alpha(x)([\![s_1]\!]^{\mathsf{s}}_{\alpha^\gamma,\zeta^\gamma_\alpha}, \dots, [\![s_n]\!]^{\mathsf{s}}_{\alpha^\gamma,\zeta^\gamma_\alpha}) + \Sigma_{i=1}^n \mathsf{cost}(s_i)_{\alpha^\gamma,\zeta^\gamma_\alpha} = [\![\gamma(x)]\!]^{\mathsf{c}}_{\alpha,\zeta}([\![s_1\gamma]\!]^{\mathsf{s}}_{\alpha}, \dots, [\![s_n\gamma]\!]^{\mathsf{s}}_{\alpha}) + \Sigma_{i=1}^n \mathsf{cost}(s_i)_{\alpha^\gamma,\zeta^\gamma_\alpha}$ by Lemmas 3.7 and 3.9. By Lemma 3.6 and the induction hypothesis this $\geq [\![(x\gamma)\,(s_1\gamma)\cdots(s_n\gamma)]\!]^{\mathsf{c}} + \Sigma_{i=1}^n \mathsf{cost}^\star(s_i\gamma)_{\alpha,\zeta}$. Since $x\gamma$ is in normal form, either this is exactly $\mathsf{cost}^\star(s\gamma)$, or $\mathsf{cost}^\star(s\gamma) = 0$ and we are done regardless. If $n > 0$ and s does not have base type, we complete quickly with the induction hypothesis.
- For the second case, let $s = \mathsf{f}\, s_1 \dots s_k\, t_1 \dots t_n$ We have two cases whether $s\gamma$ is in normal form or not. In the first case, $\mathsf{cost}^\star(s\gamma)_{\alpha,\zeta} = 0$ and certainly $\mathsf{cost}(s)_{\alpha^\gamma,\zeta^\gamma_\alpha} \geq 0$. For the second case, s is not in normal form.

  If s has base type, then:

$$\begin{aligned}
\mathsf{cost}(s)_{\alpha^\gamma,\zeta^\gamma_\alpha} &= \mathsf{cost}(\mathsf{f}\, s_1 \dots s_k\, t_1 \dots t_n)_{\alpha^\gamma,\zeta^\gamma_\alpha} \\
&= [\![s]\!]^{\mathsf{c}}_{\alpha^\gamma,\zeta^\gamma_\alpha} + \sum_{i=1}^k \mathsf{cost}(s_i)_{\alpha^\gamma,\zeta^\gamma_\alpha} + \sum_{j=1}^n \mathsf{cost}(t_j)_{\alpha^\gamma,\zeta^\gamma_\alpha} \\
&\overset{\text{Lemma 3.9}}{=} [\![s\gamma]\!]^{\mathsf{c}}_{\alpha,\zeta} + \sum_{i=1}^k \mathsf{cost}(s_i)_{\alpha^\gamma,\zeta^\gamma_\alpha} + \sum_{j=1}^n \mathsf{cost}(t_j)_{\alpha^\gamma,\zeta^\gamma_\alpha} \\
&\overset{IH}{\geq} [\![s\gamma]\!]^{\mathsf{c}}_{\alpha,\zeta} + \sum_{i=1}^k \mathsf{cost}^\star(s_i\gamma)_{\alpha,\zeta} + \sum_{j=1}^n \mathsf{cost}^\star(t_j\gamma)_{\alpha,\zeta} \\
&= \mathsf{cost}^\star(s\gamma)_{\alpha,\zeta}
\end{aligned}$$

  If not, then:

$$\begin{aligned}
\mathsf{cost}(s)_{\alpha^\gamma,\zeta^\gamma_\alpha} &= \sum_{i=1}^k \mathsf{cost}(s_i)_{\alpha^\gamma,\zeta^\gamma_\alpha} + \sum_{j=1}^n \mathsf{cost}(t_j)_{\alpha^\gamma,\zeta^\gamma_\alpha} \\
&\overset{IH}{\geq} \sum_{i=1}^k \mathsf{cost}^\star(s_i\gamma)_{\alpha,\zeta} + \sum_{j=1}^n \mathsf{cost}^\star(t_j\gamma)_{\alpha,\zeta} \\
&= \mathsf{cost}^\star(s\gamma)_{\alpha,\zeta}
\end{aligned} \qquad \square$$

Lemma 3.11. Let $(\mathbb{F}, \mathcal{R})$ be an STRS compatible with a cost–size interpretation following Definition 3.1 and s, t be type-1 terms of the same type, such that $s \to_{\mathcal{R}} t$. Then we have that $[\![s]\!]^{\mathsf{c}}_{\alpha,\zeta} \geq [\![t]\!]^{\mathsf{c}}_{\alpha,\zeta}$.

Proof. By induction on the size of s. There are two cases.

- First, $s = x\, s_1 \dots s_n$. We can write $x : \iota_1 \Rightarrow \cdots \Rightarrow \iota_k \Rightarrow \kappa$ and, since s is type-1 (so *not* of base type), $n < k$. Then $t = x\, s_1 \dots s_i' \dots s_n$ with $s_i \to_{\mathcal{R}} s_i'$. Hence, by Lemma 3.8 and monotonicity of $\zeta(x)$:

$$\begin{aligned}\llbracket x\, s_1 \dots s_i \dots s_n \rrbracket^{\mathsf{c}}_{\alpha,\zeta} &= \zeta(x)(\llbracket s_1 \rrbracket^{\mathsf{s}}_{\alpha}, \dots, \llbracket s_i \rrbracket^{\mathsf{s}}_{\alpha}, \dots, \llbracket s_n \rrbracket^{\mathsf{s}}_{\alpha}) \\ &\sqsupseteq \zeta(x)(\llbracket s_1 \rrbracket^{\mathsf{s}}_{\alpha}, \dots, \llbracket s_i' \rrbracket^{\mathsf{s}}_{\alpha}, \dots, \llbracket s_n \rrbracket^{\mathsf{s}}_{\alpha}) \\ &= \llbracket x\, s_1 \dots s_i' \dots s_n \rrbracket^{\mathsf{c}}_{\alpha,\zeta}\end{aligned}$$

- For the second part, $s = \mathsf{f}\, s_1 \cdots s_k\, t_1 \cdots t_n$, recall that rules are of base type, and therefore reduction does not occur at head position. Hence, either $t = \mathsf{f}\, s_1 \cdots s_i' \cdots s_k\, t_1 \cdots t_n$ with $s_i \to_{\mathcal{R}} s_i'$, or $t = \mathsf{f}\, s_1 \cdots s_k\, t_1 \cdots t_i' \cdots t_n$ with $t_i \to_{\mathcal{R}} t_i'$.

  In the first case,

$$\begin{aligned} & \llbracket \mathsf{f}\, s_1 \dots s_i \dots s_k\, t_1 \dots t_n \rrbracket^{\mathsf{c}}_{\alpha,\zeta} \\ = \ & \mathcal{J}^{\mathsf{c}}_{\mathsf{f}}(\llbracket s_1 \rrbracket^{\mathsf{c}}_{\alpha,\zeta}, \llbracket s_1 \rrbracket^{\mathsf{s}}_{\alpha}, \dots, \llbracket s_i \rrbracket^{\mathsf{c}}_{\alpha,\zeta}, \llbracket s_i \rrbracket^{\mathsf{s}}_{\alpha}, \dots, \llbracket s_k \rrbracket^{\mathsf{c}}_{\alpha,\zeta}, \llbracket s_k \rrbracket^{\mathsf{s}}_{\alpha}, \llbracket t_1 \rrbracket^{\mathsf{s}}_{\alpha}, \dots, \llbracket t_n \rrbracket^{\mathsf{s}}_{\alpha}) \\ \geq \ & \mathcal{J}^{\mathsf{c}}_{\mathsf{f}}(\llbracket s_1 \rrbracket^{\mathsf{c}}_{\alpha,\zeta}, \llbracket s_1 \rrbracket^{\mathsf{s}}_{\alpha}, \dots, \llbracket s_i \rrbracket^{\mathsf{c}}_{\alpha,\zeta}, \llbracket s_i \rrbracket^{\mathsf{s}}_{\alpha}, \dots, \llbracket s_k \rrbracket^{\mathsf{c}}_{\alpha,\zeta}, \llbracket s_k \rrbracket^{\mathsf{s}}_{\alpha}, \llbracket t_1 \rrbracket^{\mathsf{s}}_{\alpha}, \dots, \llbracket t_n \rrbracket^{\mathsf{s}}_{\alpha}) \\ = \ & \llbracket \mathsf{f}\, s_1 \dots s_i' \dots s_k\, t_1 \dots t_n \rrbracket^{\mathsf{c}}_{\alpha,\zeta}\end{aligned}$$

  because $\llbracket s_i \rrbracket^{\mathsf{s}}_{\alpha} \sqsupseteq \llbracket s_i' \rrbracket^{\mathsf{s}}$ by Lemma 3.8, $\llbracket s_i \rrbracket^{\mathsf{c}}_{\alpha,\zeta} \geq \llbracket s_i' \rrbracket^{\mathsf{c}}_{\alpha,\zeta}$ by the induction hypothesis, and $\mathcal{J}^{\mathsf{c}}_{\mathsf{f}}$ is weakly monotonic. For the second case, we only need Lemma 3.8 and monotonicity, not the induction hypothesis. □

Finally, we are ready to prove the innermost compatibility theorem.

Theorem 3.4 (Innermost Compatibility)**.** Suppose $(\mathbb{F}, \mathcal{R})$ is an STRS compatible with a cost–size interpretation $\mathcal{F}$ (following Definition 3.1). Then for any valuations α and ζ we have $\mathsf{cost}^\star(s)_{\alpha,\zeta} > \mathsf{cost}^\star(t)_{\alpha,\zeta}$ and $\llbracket s \rrbracket^{\mathsf{s}}_{\alpha} \sqsupseteq \llbracket t \rrbracket^{\mathsf{s}}_{\alpha}$ whenever $s \to_{\mathcal{R}} t$.

Proof. The second part of the theorem is given by Lemma 3.8. The first part follows by induction on the reduction $s \to_{\mathcal{R}} t$.

- For the base case, we have by Lemmas 3.9 and 3.10 that

$$\mathsf{cost}^\star(\ell\gamma)_{\alpha,\zeta} = \llbracket \ell\gamma \rrbracket^{\mathsf{c}}_{\zeta} = \llbracket \ell \rrbracket^{\mathsf{c}}_{\zeta^\gamma_\alpha} > \mathsf{cost}(r)_{\alpha^\gamma,\zeta^\gamma_\alpha} \geq \mathsf{cost}^\star(r\gamma)_{\alpha,\zeta}$$

- For the application case with a variable root symbol, we have that $x\, t_1 \dots t_i \dots t_n \to_{\mathcal{R}} x\, t_1 \dots t_i' \dots t_n$ with $t_i \to_{\mathcal{R}} t_i'$. By induction we get $\mathsf{cost}^\star(t_i)_{\alpha,\zeta} > \mathsf{cost}^\star(t_i')_{\alpha,\zeta}$, and by Lemma 3.8 we have $\llbracket t_i \rrbracket^{\mathsf{s}}_{\alpha} \sqsupseteq \llbracket t_i' \rrbracket^{\mathsf{s}}_{\alpha}$. Then, if s has base type:

$$\begin{aligned} & \mathsf{cost}^\star(x\, t_1 \dots t_i \dots t_n)_{\alpha,\zeta} \\ &= \llbracket x\, t_1 \dots t_i \dots t_n \rrbracket^{\mathsf{c}}_{\alpha,\zeta} + \sum_{j=1}^{n} \mathsf{cost}^\star(t_j)_{\alpha,\zeta} \\ &= \zeta(x)(\llbracket t_1 \rrbracket^{\mathsf{s}}_{\alpha}, \dots, \llbracket t_i \rrbracket^{\mathsf{s}}_{\alpha}, \dots, \llbracket t_n \rrbracket^{\mathsf{s}}_{\alpha}) + \Big(\sum_{\substack{j=1\dots n \\ j \neq i}} \mathsf{cost}^\star(t_j)_{\alpha,\zeta} \Big) + \mathsf{cost}^\star(t_i)_{\alpha,\zeta} \\ &\geq \zeta(x)(\llbracket t_1 \rrbracket^{\mathsf{s}}_{\alpha}, \dots, \llbracket t_i' \rrbracket^{\mathsf{s}}_{\alpha}, \dots, \llbracket t_n \rrbracket^{\mathsf{s}}_{\alpha}) + \Big(\sum_{\substack{j=1\dots n \\ j \neq i}} \mathsf{cost}^\star(t_j)_{\alpha,\zeta} \Big) + \mathsf{cost}^\star(t_i)_{\alpha,\zeta}, \\ &> \zeta(x)(\llbracket t_1 \rrbracket^{\mathsf{s}}_{\alpha}, \dots, \llbracket t_i' \rrbracket^{\mathsf{s}}_{\alpha}, \dots, \llbracket t_n \rrbracket^{\mathsf{s}}_{\alpha}) + \Big(\sum_{\substack{j=1\dots n \\ j \neq i}} \mathsf{cost}^\star(t_j)_{\alpha,\zeta} \Big) + \mathsf{cost}^\star(t_i')_{\alpha,\zeta}\end{aligned}$$

$$= \mathsf{cost}^\star(x\, t_1 \dots t_i' \dots t_n)_{\alpha,\zeta}$$

If s does not have base type, we have a similar reasoning without the component $[\![x\, t_1 \dots t_i \dots t_n]\!]^{\mathsf{c}}_{\alpha,\zeta}$.

- For the application case with a function root symbol where the reduction is done in a base-type argument, we have that $\mathsf{f}\, s_1 \dots s_k\, t_1 \dots t_i \dots t_n \to_{\mathcal{R}} \mathsf{f}\, s_1 \dots s_k\, t_1 \dots t_i' \dots t_n$ with $t_i \to_{\mathcal{R}} t_i'$. Let us write $\vec{s}$ for $s_1 \dots s_k$ and $\mathsf{c}(s)$ for $\sum_{j=1}^{k} \mathsf{cost}^\star(s_i)_{\alpha,\zeta}$ below. We also abuse notation and write $[\![\vec{s}]\!]^{\mathsf{c}}_{\alpha,\zeta}, [\![\vec{s}]\!]^{\mathsf{s}}_{\alpha}$ for $[\![s_1]\!]^{\mathsf{c}}_{\alpha,\zeta}, [\![s_1]\!]^{\mathsf{s}}_{\alpha}, \dots, [\![s_k]\!]^{\mathsf{c}}_{\alpha,\zeta}, [\![s_k]\!]^{\mathsf{s}}_{\alpha}$. If s has base type:

$$\begin{aligned}
&\mathsf{cost}^\star(\mathsf{f}\, \vec{s}\, t_1 \dots t_i \dots t_n)_{\alpha,\zeta}\\
&= [\![\mathsf{f}\, \vec{s}\, t_1 \dots t_i \dots t_n]\!]^{\mathsf{c}}_{\alpha,\zeta} + \mathsf{c}(s) + \sum_{j=1}^{n} \mathsf{cost}^\star(t_j)_{\alpha,\zeta}\\
&= \mathcal{J}^{\mathsf{c}}_{\mathsf{f}}([\![\vec{s}]\!]^{\mathsf{c}}_{\alpha,\zeta}, [\![\vec{s}]\!]^{\mathsf{s}}_{\alpha}, [\![t_1]\!]^{\mathsf{s}}_{\alpha}, \dots, [\![t_i]\!]^{\mathsf{s}}_{\alpha}, \dots, [\![t_n]\!]^{\mathsf{s}}_{\alpha}) + \mathsf{c}(s) + \sum_{j=1}^{n} \mathsf{cost}^\star(t_j)_{\alpha,\zeta}\\
&\geq \mathcal{J}^{\mathsf{c}}_{\mathsf{f}}([\![\vec{s}]\!]^{\mathsf{c}}_{\alpha,\zeta}, [\![\vec{s}]\!]^{\mathsf{s}}_{\alpha}, [\![t_1]\!]^{\mathsf{s}}_{\alpha}, \dots, [\![t_i']\!]^{\mathsf{s}}_{\alpha}, \dots, [\![t_n]\!]^{\mathsf{s}}_{\alpha}) + \mathsf{c}(s) + \sum_{j=1}^{n} \mathsf{cost}^\star(t_j)_{\alpha,\zeta}\\
&> \mathsf{cost}^\star(\mathsf{f}\, \vec{s}\, t_1 \dots t_i' \dots t_n)_{\alpha,\zeta}
\end{aligned}$$

where in the last step we use $\mathsf{cost}^\star(t_i)_{\alpha,\zeta} > \mathsf{cost}^\star(t_i')_{\alpha,\zeta}$, given by the IH. If s does not have base type, the reasoning is similar, only omitting the component $[\![\mathsf{f}\, \vec{s}\, t_1 \dots t_i \dots t_n]\!]^{\mathsf{c}}_{\alpha,\zeta}$.

- For the application case with a function root symbol where the reduction is done in a higher-type argument, we have that $\mathsf{f}\, s_1 \dots s_i \dots s_k\, t_1 \dots t_n \to_{\mathcal{R}} \mathsf{f}\, s_1 \dots s_i' \dots s_k\, t_1 \dots t_n$ with $s_i \to_{\mathcal{R}} s_i'$. Recall that by IH we get $\mathsf{cost}^\star(s_i)_{\alpha,\zeta} > \mathsf{cost}^\star(s_i')_{\alpha,\zeta}$. If s does not have base type, we conclude in a similar reasoning to the ones used above, simply counting the $\mathsf{cost}^\star(\cdot)$ for all arguments. If s does have base type, note that s_i and s_i' are type-1 terms, so we can apply Lemma 3.11 to obtain $[\![s_i]\!]^{\mathsf{c}} \geq [\![s_i']\!]^{\mathsf{c}}$. With this in hand we reason as follows:

$$\begin{aligned}
&\mathsf{cost}^\star(\mathsf{f}\, s_1 \dots s_i \dots s_k\, \vec{t})_{\alpha,\zeta}\\
&= \mathcal{J}^{\mathsf{c}}_{\mathsf{f}}([\![s_1]\!]^{\mathsf{c}}_{\alpha,\zeta}, [\![s_1]\!]^{\mathsf{s}}_{\alpha}, \dots, [\![s_i]\!]^{\mathsf{c}}_{\alpha,\zeta}, [\![s_i]\!]^{\mathsf{s}}_{\alpha} \dots, [\![s_k]\!]^{\mathsf{c}}_{\alpha,\zeta}, [\![s_k]\!]^{\mathsf{s}}_{\alpha}, [\![\vec{t}]\!]^{\mathsf{s}}_{\alpha})\\
&\qquad + (\sum_{j=1\dots k, j\neq i} \mathsf{cost}^\star(s_j)_{\alpha,\zeta}) + \mathsf{cost}^\star(s_i)_{\alpha,\zeta} + (\sum_{j=1}^{n} \mathsf{cost}^\star(t_j)_{\alpha,\zeta})
\end{aligned}$$

By monotonicity of $\mathcal{J}^{\mathsf{c}}_{\mathsf{f}}$ and $[\![s_i]\!]^{\mathsf{c}}_{\alpha,\zeta} \geq [\![s_i']\!]^{\mathsf{c}}_{\alpha,\zeta}$, $[\![s_i]\!]^{\mathsf{s}}_{\alpha} \sqsupseteq [\![s_i']\!]^{\mathsf{s}}_{\alpha}$, we get

$$\begin{aligned}
&\geq \mathcal{J}^{\mathsf{c}}_{\mathsf{f}}([\![s_1]\!]^{\mathsf{c}}_{\alpha,\zeta}, [\![s_1]\!]^{\mathsf{s}}_{\alpha}, \dots, [\![s_i']\!]^{\mathsf{c}}_{\alpha,\zeta}, [\![s_i']\!]^{\mathsf{s}}_{\alpha} \dots, [\![s_k]\!]^{\mathsf{c}}_{\alpha,\zeta}, [\![s_k]\!]^{\mathsf{s}}_{\alpha}, [\![\vec{t}]\!]^{\mathsf{s}}_{\alpha})\\
&\qquad + (\sum_{j=1\dots k, j\neq i} \mathsf{cost}^\star(s_j)_{\alpha,\zeta}) + \mathsf{cost}^\star(s_i)_{\alpha,\zeta} + (\sum_{j=1}^{n} \mathsf{cost}^\star(t_j)_{\alpha,\zeta})\\
&> \mathcal{J}^{\mathsf{c}}_{\mathsf{f}}([\![s_1]\!]^{\mathsf{c}}_{\alpha,\zeta}, [\![s_1]\!]^{\mathsf{s}}_{\alpha}, \dots, [\![s_i']\!]^{\mathsf{c}}_{\alpha,\zeta}, [\![s_i']\!]^{\mathsf{s}}_{\alpha} \dots, [\![s_k]\!]^{\mathsf{c}}_{\alpha,\zeta}, [\![s_k]\!]^{\mathsf{s}}_{\alpha}, [\![\vec{t}]\!]^{\mathsf{s}}_{\alpha})\\
&\qquad + (\sum_{j=1\dots k, j\neq i} \mathsf{cost}^\star(s_j)_{\alpha,\zeta}) + \mathsf{cost}^\star(s_i')_{\alpha,\zeta} + (\sum_{j=1}^{n} \mathsf{cost}^\star(t_j)_{\alpha,\zeta})\\
&= \mathsf{cost}^\star(\mathsf{f}\, s_1 \dots s_i' \dots s_k\, \vec{t})_{\alpha,\zeta}
\end{aligned}$$

□

4. From Higher-Order Rewriting to $\mathtt{BFF}_2$ and Back

The main result of this paper roughly states that $\mathtt{BFF}_2$ consists exactly of those type-2 functionals computed by an STRS compatible with a cost–size tuple interpretation, whose main function is polynomially bounded. To formally state this result, we must first define what it means for an STRS to *compute* a type-2 functional and define precisely the class of cost–size interpretations we are interested in. The challenge is that such an STRS is required to do what it is supposed to do *for every* input, and the inputs not only consist of words, but also of functions on words. We thus have to find a way to encode type-1 functions seen as inputs to a type-2 program.

Indeed, let us start by encoding words in W as terms. We let $\mathsf{bit}, \mathsf{word} \in \mathbb{B}$ and introduce symbols $\mathsf{o}, \mathsf{i} : \mathsf{bit}$ and $[] : \mathsf{word}$, $:: : \mathsf{bit} \Rightarrow \mathsf{word} \Rightarrow \mathsf{word}$ (where $::$ will be used in infix notation). Then for instance 001 is encoded as the term $\mathsf{o}::(\mathsf{o}::(\mathsf{i}::[]))$. In practice, we usually employ the cleaner list-like notation $[\mathsf{o}; \mathsf{o}; \mathsf{i}]$. Let $\underline{\mathsf{w}}$ denote the term encoding of a word w.

Example 4.1 (Implementing Binary Addition)**.** Let us implement binary addition. For this purpose, we consider binary sequences written in *little-endian* format, i.e., the least significant digit is at the head of the list. So the decimal number 6 is written as `011` in little-endian notation. To start, we will need the following logical operations on bit symbols.

$$\begin{array}{cccc} \mathsf{o} \,\mathsf{xor}\, \mathsf{o} \to \mathsf{o} & \mathsf{i} \,\mathsf{xor}\, \mathsf{i} \to \mathsf{o} & \mathsf{o} \,\mathsf{xor}\, \mathsf{i} \to \mathsf{i} & \mathsf{i} \,\mathsf{xor}\, \mathsf{o} \to \mathsf{i} \\ \mathsf{o} \,\mathsf{and}\, \mathsf{o} \to \mathsf{o} & \mathsf{i} \,\mathsf{and}\, \mathsf{i} \to \mathsf{i} & \mathsf{o} \,\mathsf{and}\, \mathsf{i} \to \mathsf{o} & \mathsf{i} \,\mathsf{and}\, \mathsf{o} \to \mathsf{o} \\ \mathsf{o} \,\mathsf{or}\, \mathsf{o} \to \mathsf{o} & \mathsf{i} \,\mathsf{or}\, \mathsf{i} \to \mathsf{i} & \mathsf{o} \,\mathsf{or}\, \mathsf{i} \to \mathsf{i} & \mathsf{i} \,\mathsf{or}\, \mathsf{o} \to \mathsf{i} \end{array}$$

The rules defining $\mathsf{aux} : \mathsf{word} \Rightarrow \mathsf{word} \Rightarrow \mathsf{bit} \Rightarrow \mathsf{word}$ below compute the bitwise addition of two binary numbers, given a carrying value as the third argument. This is done by recursing over the input lists.

$$\begin{array}{l} \mathsf{aux}\ []\,[]\,\mathsf{o} \to [] \\ \mathsf{aux}\ []\,[]\,\mathsf{i} \to \mathsf{i}::[] \\ \mathsf{aux}\ (a::as)\,[]\,acc \to (a \,\mathsf{xor}\, acc)::\mathsf{aux}\ as\,[]\,(a \,\mathsf{and}\, acc) \\ \mathsf{aux}\ []\,(b::bs)\,acc \to (b \,\mathsf{xor}\, acc)::\mathsf{aux}\ []\,bs\,(b \,\mathsf{and}\, acc) \\ \mathsf{aux}\ (a::as)\,(b::bs)\,acc \to ((a \,\mathsf{xor}\, b) \,\mathsf{xor}\, acc)::\mathsf{aux}\ as\,bs\,(((a \,\mathsf{xor}\, b) \,\mathsf{and}\, acc) \,\mathsf{or}\, (a \,\mathsf{and}\, b)) \end{array}$$

Finally, we write the addition of binary numbers as the rule $x +_{\mathsf{B}} y \to \mathsf{aux}\,x\,y\,\mathsf{o}$. We can orient all rules by setting the following interpretation:

$$\begin{array}{lll} \mathcal{J}^{\mathsf{s}}_{\mathsf{o}} = \mathcal{J}^{\mathsf{s}}_{\mathsf{i}} = \mathcal{J}^{\mathsf{s}}_{[]} = 0 & \mathcal{J}^{\mathsf{c}}_{\mathsf{o}} = \mathcal{J}^{\mathsf{c}}_{\mathsf{i}} = \mathcal{J}^{\mathsf{c}}_{[]} = 0 & \\ \mathcal{J}^{\mathsf{s}}_{::} = \boldsymbol{\lambda} xy.1 + y & \mathcal{J}^{\mathsf{c}}_{::} = \boldsymbol{\lambda} xy.0 & \\ \mathcal{J}^{\mathsf{s}}_{op} = \boldsymbol{\lambda} xy.0 & \mathcal{J}^{\mathsf{c}}_{op} = \boldsymbol{\lambda} xy.1 & \text{for } op \in \{\mathsf{xor}, \mathsf{and}, \mathsf{or}\} \\ \mathcal{J}^{\mathsf{s}}_{\mathsf{aux}} = \boldsymbol{\lambda} xya.1 + \max(x, y) & \mathcal{J}^{\mathsf{c}}_{\mathsf{aux}} = \boldsymbol{\lambda} xya.1 + 7 * \max(x, y) & \\ \mathcal{J}^{\mathsf{s}}_{+_{\mathsf{B}}} = \boldsymbol{\lambda} xy.1 + \max(x, y) & \mathcal{J}^{\mathsf{c}}_{+_{\mathsf{B}}} = \boldsymbol{\lambda} xy.2 + 7 * \max(x, y) & \end{array}$$

This interpretation assigns cost 0 to all ground constructor terms: ground terms that are exclusively built from the constructor symbols i, o, $[]$ and $::$. The size of a binary word is its length. The cost of adding two numbers is bounded by $2 + 7 * \langle\text{the length of the longest}\rangle$: this accounts for iterating through the list, and doing 6 bit operations for each step.

Next, we encode type-1 functions as a possibly infinite set of one-step rewrite rules.

Definition 4.2. Consider a type-1 function $f : W \longrightarrow W$ and let $\mathsf{S}_f : \mathsf{word} \Rightarrow \mathsf{word}$ be a fresh function symbol. A set of rules $\mathcal{R}_f$ **defines** a function $f : \mathbb{N} \longrightarrow \mathbb{N}$ by way of the symbol S_f if and only if $\mathcal{R}_f$ contains exactly the rules $\mathsf{S_f}\,\underline{\mathsf{n}} \to \underline{\mathsf{m}}$ where $n, m \in \mathbb{N}$ and $m = f(n)$.

Intuitively, this infinite set of rules is the rewriting counterpart of an oracle f. Indeed, in a single rewrite step $\mathsf{S_f}\,\underline{\mathsf{x}}$ rewrites to the value $\underline{\mathsf{f(x)}}$. Henceforth, we assume given that our STRS $(\mathbb{F}, \mathcal{R})$ at hand is such that $\mathbb{F}$ contains $\mathsf{o}, \mathsf{i}, [], ::$ typed as above and a distinguished symbol $\mathsf{F} : (\mathsf{word} \Rightarrow \mathsf{word})^k \Rightarrow \mathsf{word}^l \Rightarrow \mathsf{word}$. Given type-1 functions $f_1, \dots, f_k$, we write $\mathbb{F}_{\vec{f}}$ for $\mathbb{F}$ extended with function symbols $\mathsf{S}_{f_i} : \mathsf{word} \Rightarrow \mathsf{word}$, with $1 \leq i \leq k$, and let $\mathcal{R}_{+\vec{f}} = \mathcal{R} \cup \bigcup_{i=1}^{k} \mathcal{R}_f$. Now we can define the notion of type-2 computability for such STRSs.

Definition 4.3. Let $(\mathbb{F}, \mathcal{R})$ be an STRS. We say that F **computes** the type-2 functional Ψ in $(\mathbb{F}, \mathcal{R})$ iff for all type-1 functions $f_1, \dots, f_k$ and all $w_1, \dots, w_l \in W$, $\mathsf{F}\,\mathsf{S}_{f_1} \cdots \mathsf{S}_{f_k}\,\underline{\mathsf{w_1}} \cdots \underline{\mathsf{w_l}} \to^{+}_{\mathcal{R}_{+\vec{f}}} \underline{\mathsf{u}}$, where $u = \Psi(f_1, \dots, f_k, w_1, \dots, w_l)$.

Next, we define what we mean by polynomially bounded interpretation.

Definition 4.4. We say that an STRS $(\mathbb{F}, \mathcal{R})$ **admits** a polynomially bounded interpretation iff $(\mathbb{F}, \mathcal{R})$ is compatible with a cost–size interpretation such that:

- $\mathcal{S}_{\mathsf{word}} = (\mathbb{N}, \geq)$;
- $\mathcal{J}^{\mathsf{c}}_{\mathsf{o}} = \mathcal{J}^{\mathsf{c}}_{\mathsf{i}} = \mathcal{J}^{\mathsf{c}}_{[]} = 0$, $\mathcal{J}^{\mathsf{c}}_{::} = \boldsymbol{\lambda} xy.0$, and $\mathcal{J}^{\mathsf{s}}_{::} = \boldsymbol{\lambda} xy.x + y + c$ for some $c \geq 1$;
- $\mathcal{J}^{\mathsf{c}}_{\mathsf{F}}$ is bounded by a polynomial in $\mathtt{Pol}^2_{\mathbb{N}}[F^{\mathsf{c}}_1, F^{\mathsf{s}}_1, \dots, F^{\mathsf{c}}_k, F^{\mathsf{s}}_k; x_1, \dots, x_l]$.

Note that the condition on $\mathcal{J}^{\mathsf{s}}_{::}$ corresponds to what is called *additive* quasi-interpretation in [BMM11]. Finally, we can formally state our main result.

Theorem 4.5. A type-2 functional Ψ is in $\mathtt{BFF}_2$ if and only if there exists a finite orthogonal STRS $(\mathbb{F}, \mathcal{R})$ such that the distinguished symbol F computes Ψ in $(\mathbb{F}, \mathcal{R})$ and $\mathcal{R}$ admits a polynomially bounded cost–size interpretation.

Note that the polynomial bound *only* refers to the cost-size interpretation for F. Indeed, except for o, i, $[]$, and $::$ there are no requirements on the interpretations for other symbols. We will prove this result in two parts. First, we prove soundness in Section 5 which states that every type-2 functional computed by an STRS as above is in $\mathtt{BFF}_2$. Then in Section 6 we prove completeness, which states that every functional in $\mathtt{BFF}_2$ can be computed by such an STRS. In order to simplify proofs, we only consider type-2 functions of rank (1,1). We claim that the results can be easily generalized, but the proofs become more tedious when handling multiple arguments.

Example 4.6. Let us consider the type-2 functional defined by $\Psi \coloneqq \boldsymbol{\lambda} f x. \sum_{i<|x|} f(i)$, where $|x|$ refers to the length of the word x. Notice that Ψ adds all $f(i)$ over each word $i \in W$ whose value (as a binary number) is smaller than the length of x. This functional was proved to lie in $\mathtt{BFF}_2$ in [IRK01], where the authors used an encoding of Ψ as a BTLP_2 program. We can encode Ψ as an STRS as follows. We expand on the STRS from Example 4.1. Let us also include the type nat and data constructors $\mathsf{0}$ and s from Example 2.1, and consider ancillary symbols $\mathsf{lengthOf} : \mathsf{word} \Rightarrow \mathsf{nat}$ and $\mathsf{toBin} : \mathsf{nat} \Rightarrow \mathsf{word}$, defined by the following rules:

$$\mathsf{lengthOf}\,[] \to \mathsf{0} \qquad \mathsf{lengthOf}\,(a::as) \to \mathsf{s}\,(\mathsf{lengthOf}\,as)$$
$$\mathsf{toBin}\,\mathsf{0} \to [] \qquad \mathsf{toBin}\,(\mathsf{s}\,n) \to (\mathsf{toBin}\,n) +_{\mathsf{B}} (\mathsf{i}::[])$$

The former computes the length of a given word and the latter converts a number from unary to binary representation (using the binary addition symbol $+_{\mathsf{B}} : \mathsf{word} \Rightarrow \mathsf{word} \Rightarrow \mathsf{word}$, whose rules were given in Example 4.1). Then Ψ is computed by:

$$\begin{aligned} \mathsf{compute}\, F\, 0\, acc &\to acc \\ \mathsf{compute}\, F\, (\mathsf{s}\, i)\, acc &\to \mathsf{compute}\, F\, i\, (acc +_{\mathsf{B}} F(\mathsf{toBin}\, i)) \\ \mathsf{start}\, F\, x &\to \mathsf{compute}\, F\, (\mathsf{lengthOf}\, x)\, [] \end{aligned}$$

That is, if we want to compute $\Psi(f, x)$ we simply reduce the term $\mathsf{start}\, \mathsf{S}_f\, \underline{\mathsf{x}}$ to normal form. By Theorem 4.5, to show that this system is in $\mathtt{BFF}_2$ via our rewriting formalism we need to exhibit a cost–size tuple interpretation for it that satisfies Definition 4.4. The interpretation functions for o, i, $[]$ and $::$ from Example 4.1 satisfy the requirements (with $c = 1$); in addition let us set the following interpretations:

$$\begin{array}{cccc} \mathcal{J}^{\mathsf{s}}_{0} = 0 & \mathcal{J}^{\mathsf{c}}_{0} = 0 & \mathcal{J}^{\mathsf{s}}_{\mathsf{s}} = \boldsymbol{\lambda} x.1 + x & \mathcal{J}^{\mathsf{c}}_{\mathsf{s}} = \boldsymbol{\lambda} x.0 \\ \mathcal{J}^{\mathsf{s}}_{\mathsf{lengthOf}} = \boldsymbol{\lambda} x.x & \mathcal{J}^{\mathsf{c}}_{\mathsf{lengthOf}} = \boldsymbol{\lambda} x.1 + x & \mathcal{J}^{\mathsf{s}}_{\mathsf{toBin}} = \boldsymbol{\lambda} x.1 + x & \mathcal{J}^{\mathsf{c}}_{\mathsf{toBin}} = \boldsymbol{\lambda} x.7 * x^2 + 3 * x + 1 \end{array}$$

$$\begin{aligned} \mathcal{J}^{\mathsf{s}}_{\mathsf{compute}} &= \boldsymbol{\lambda} F j a.j + \max(a, F(j)) \\ \mathcal{J}^{\mathsf{c}}_{\mathsf{compute}} &= \boldsymbol{\lambda} F^{\mathsf{c}} F^{\mathsf{s}} j a.4 * j^3 + 7 * j * \max(a, F^{\mathsf{s}}(j)) + j * F^{\mathsf{c}}(j) + 1 \\ \mathcal{J}^{\mathsf{s}}_{\mathsf{start}} &= \boldsymbol{\lambda} F x.x + F(x) \\ \mathcal{J}^{\mathsf{c}}_{\mathsf{start}} &= \boldsymbol{\lambda} F^{\mathsf{c}} F^{\mathsf{s}} x.4 * x^3 + 7 * x * F^{\mathsf{s}}(x) + x * F^{\mathsf{c}}(x) + x + 3 \end{aligned}$$

This orients all rules, and $\mathcal{J}^{\mathsf{c}}_{\mathsf{start}}$ is a polynomial.

5. Soundness

In order to prove soundness, let us consider a fixed finite orthogonal STRS $\mathcal{R}$ with a distinguished function symbol F which admits a polynomially bounded cost–size interpretation, that computes a type-2 functional Ψ. We proceed to show that Ψ is in $\mathtt{BFF}_2$ roughly as follows.

(1) Since $\mathcal{R}$ computes Ψ and admits a polynomially bounded interpretation, we show that so does the extended system $\mathcal{R}_{+f}$ (Definition 4.3). The restriction on $\mathcal{J}^{\mathsf{s}}_{::}$ (Definition 4.4) implies that $[\![\mathsf{F}\, \mathsf{S}_f\, \underline{\mathsf{w}}]\!]^{\mathsf{c}}$ is bounded by a second-order polynomial over $|f|, |w|$. We show this in Lemma 5.1. By compatibility (Corollary 3.5), we can do at most polynomially many steps when reducing $\mathsf{F}\, \mathsf{S}_f\, \underline{\mathsf{w}}$.
(2) The cost polynomial restricts the size of any input that the function variable F is applied to (e.g., a cost bound of $3 + F^{\mathsf{c}}(m)$ implies that F is never called on a term with size interpretation $> m$). This is the subject of Lemma 5.4.
(3) Using the observations above, we then show that by graph rewriting we can simulate $\mathcal{R}_{+f}$ and compute each $\mathcal{R}_{+f}$-reduction step in polynomial time on an OTM. This guarantees that Ψ is in $\mathtt{BFF}_2$, Theorem 5.16.

5.1. **Interpreting The Extended STRS, Polynomially.** Our first goal is to provide a polynomially bounded cost–size interpretation to the extended system $\mathcal{R}_{+f}$. We start with the observation that the size interpretation of words in W is proportional to their length. Indeed, since $\mathcal{J}^{\mathsf{s}}_{::} = \boldsymbol{\lambda} xy.x + y + c$ (Definition 4.4) let $\mu := \max(\mathcal{J}^{\mathsf{s}}_{\mathsf{o}}, \mathcal{J}^{\mathsf{s}}_{\mathsf{i}}) + c$ and $\nu := \mathcal{J}^{\mathsf{s}}_{[]}$. Consequently, for all $w \in W$:

$$|w| \leq [\![\underline{\mathsf{w}}]\!]^{\mathsf{s}} \leq \mu * |w| + \nu \tag{5.1}$$

Recall that by Definition 4.2 the extended system $\mathcal{R}_{+f}$ has possibly infinitely many rules of the form $\mathsf{S}_f\,\underline{\mathsf{w}} \to \underline{\mathsf{f(w)}}$. Such rules S_f represent calls for an oracle to compute f in a single step. Thus, we set their cost to 1. The size should be given by the length of the oracle output, taking the overhead of interpretation into account. Hence, we obtain:

$$\mathcal{J}^{\mathsf{c}}_{\mathsf{S}_f} = \boldsymbol{\lambda} x.1 \qquad \mathcal{J}^{\mathsf{s}}_{\mathsf{S}_f} = \boldsymbol{\lambda} x.\mu * |f|(x) + \nu$$

Recall that $|f|(n) = \max_{|y| \leq n} |f(y)|$. Hence, $|f|$ is weakly monotonic, and therefore so is $\mathcal{J}^{\mathsf{s}}_{\mathsf{S}_f}$. This orients the rules in $\mathcal{R}_f$ because $[\![\mathsf{S}_f\,\underline{\mathsf{w}}]\!]^{\mathsf{c}} = 1 > 0 = \mathsf{cost}(\underline{\mathsf{f(w)}})$, and $[\![\mathsf{S}_f\,\underline{\mathsf{w}}]\!]^{\mathsf{s}} = \mu * |f|([\![\underline{\mathsf{w}}]\!]^{\mathsf{s}}) + \nu \geq \mu * |f|(|w|) + \nu \geq \mu * |f(w)| + \nu$ by definition of $|f|$, which is superior or equal to $[\![\underline{\mathsf{f(w)}}]\!]^{\mathsf{s}}$.

Lemma 5.1. There exists a second-order polynomial D so that $D(|f|, |w|)$ bounds the derivation height of $\mathsf{F}\,\mathsf{S}_f\,\underline{\mathsf{w}}$ for any $f \in W \longrightarrow W$ and $w \in W$.

Proof. As $\mathcal{J}^{\mathsf{c}}_{\mathsf{F}}$ is bounded by a second-order polynomial $\boldsymbol{\lambda} F^c F^s x.P$, we can let $D(F, n) := P(\boldsymbol{\lambda} x.1, \boldsymbol{\lambda} x.\mu * F(x) + \nu, \mu * n + \nu)$. Then D is a second-order polynomial, and $D(|f|, |w|) \geq \mathcal{J}^{\mathsf{c}}_{\mathsf{F}}(\mathcal{J}^{\mathsf{c}}_{\mathsf{S}_f}, \mathcal{J}^{\mathsf{s}}_{\mathsf{S}_f}, [\![\underline{\mathsf{w}}]\!]^{\mathsf{s}}) = \mathsf{cost}(\mathsf{F}\,\mathsf{S}_f\,\underline{\mathsf{w}})$ (we omit α and ζ here because there are no variables in $\mathsf{F}\,\mathsf{S}_f\,\underline{\mathsf{w}}$, and therefore the valuations are not relevant to the cost value). By Corollary 3.5, this serves as a bound on the derivation height of $\mathsf{F}\,\mathsf{S}_f\,\underline{\mathsf{w}}$. □

Notice that this lemma does not imply that Ψ is in BFF$_2$. It only guarantees that there is a polynomial bound to the *number of rewriting steps* for such systems. However, it does not immediately follow that this number is a reasonable bound for the actual computational cost of simulating a reduction on an OTM. Consider for example a rule $\mathsf{f}\,(\mathsf{s}\,n)\,t \to \mathsf{f}\,n\,(\mathsf{c}\,t\,t)$. Every step doubles the size of the term. A naive implementation — which copies the duplicated term in each step — would take exponential time. Moreover, a single step using the oracle can create a very large output, which is not considered part of the cost of the reduction, even though an OTM would be unable to use it without first fully reading it. Therefore, in order to prove soundness, we show how to realize a reasonable implementation of rewriting w.r.t. OTMs. In essence, we will show that (1) oracle calls are not problematic in the presence of polynomially bounded interpretations, and (2) we can handle duplication with an appropriate representation of rewriting. This is very much in the style of what has been done for first-order rewriting and the λ-calculus in the past [AM10, DLM12, AL16].

5.2. **Bounding The Oracle Input.** We first show that calling the oracle along a computation does not introduce an exponential overhead along the way. More precisely, we will show that there exists a second-order polynomial B such that if an oracle call $\mathsf{S}_f\,\underline{\mathsf{x}}$ occurs anywhere along the reduction $\mathsf{F}\,\mathsf{S}_f\,\underline{\mathsf{w}} \to^+_{\mathcal{R}} \underline{\mathsf{v}}$, then $|x| \leq B(|f|, |w|)$. From this, we know that the growth of the overall term size during an oracle call is at most $|f|(B(|f|, |w|))$.

Let P again be any polynomial bounding $\mathcal{J}^{\mathsf{c}}_{\mathsf{F}}$. Since P is a second-order polynomial, each occurrence of a sub-expression $F^c(E)$ in P is itself a second-order polynomial, and so is E. Let us enumerate these arguments as $E_1, \ldots, E_n$. We can then construct the new polynomial Q as follows:

$$Q \coloneqq \sum_i E_i \quad \text{where occurrences of } F^c(E_j) \text{ inside } E_i \text{ are replaced by } 1$$

The idea here is that the polynomial Q sums up all those expressions E_i (with $1 \leq i \leq n$) given to F^{c} as argument. We do this because intuitively, if during the computation a word $\underline{\mathsf{v}}$ is ever given to the oracle, then the cost for the oracle computation must be accounted for in $\mathcal{J}^{\mathsf{c}}_{\mathsf{F}}$. Hence, $[\![v]\!]^{\mathsf{s}}$ must be bounded by some E_i, so certainly by the sum of all E_i. We can safely replace occurrences of $F^{\mathsf{c}}(E_j)$ that occur inside another E_i by 1 because F^{c} will be instantiated by $\mathcal{J}^{\mathsf{c}}_{\mathsf{S}_f} = \boldsymbol{\lambda} x.1$ (or, in Lemma 5.3, by a function that maps specific input to 1).

Due to this replacement, F^{c} does not occur in Q, so Q is only parametrized by F^{s} and some type-0 variable x. It is also possible for Q to be a constant polynomial, or to be parametrized only by a type-0 variable. Its final shape depends on the arguments provided to the F^{c} in P. We let $B(G, y) := Q(\boldsymbol{\lambda} z.\mu * G(z) + \nu, \mu * y + \nu)$.

Example 5.2. Let us illustrate the construction of such polynomial Q. Consider the following polynomial P.

$$P = \boldsymbol{\lambda} F^c F^s x.x * F^{\mathsf{c}}(3 + F^{\mathsf{s}}(9 * x)) + F^{\mathsf{c}}(12) * F^{\mathsf{c}}(3 + x * F^{\mathsf{c}}(2)) + 5,$$

then Q is built by adding each of those arguments given to F^{c}. We get

$$\begin{aligned} Q &= 3 + F^{\mathsf{s}}(9 * x) + 12 + 3 + x * 1 + 2 \\ &= 20 + F^{\mathsf{s}}(9 * x) + x \end{aligned}$$

Finally, we construct the polynomial B.

$$\begin{aligned} B(G, x) &= 20 + \mu * G(9 * (\mu * x + \nu)) + \nu + (\mu * x + \nu) \\ &= 20 + 2 * \nu + G(9 * \mu * x + 9 * \nu) + \mu * x \end{aligned}$$

Now B gives an upper bound to the argument values for F^{c} that are considered: if a function differs from $\mathcal{J}^{\mathsf{c}}_{\mathsf{S}_f}$ only on argument values greater than $B(|f|, |w|)$, then we can use it in P and obtain the same result. Formally:

Lemma 5.3. Fix f, w. Let $G : \mathbb{N} \longrightarrow \mathbb{N}$ with $G(z) = \mathcal{J}^{\mathsf{c}}_{\mathsf{S}_f}(z)$ if $z \leq B(|f|, |w|)$. Then $P(G, \mathcal{J}^{\mathsf{s}}_{\mathsf{S}_f}, [\![\underline{\mathsf{w}}]\!]^{\mathsf{s}}) = P(\mathcal{J}^{\mathsf{c}}_{\mathsf{S}_f}, \mathcal{J}^{\mathsf{s}}_{\mathsf{S}_f}, [\![\underline{\mathsf{w}}]\!]^{\mathsf{s}})$.

This is proved by induction on the form of P, using that G is never applied on arguments larger than $B(|f|, |w|)$. Lemma 5.3 is used in the following key result:

Lemma 5.4 (Oracle Subterm Lemma)**.** Let $f : W \longrightarrow W$ be a type-1 function and $w \in W$. If $\mathsf{F}\,\mathsf{S}_f\,\underline{\mathsf{w}} \rightarrow^*_{\mathcal{R}_{+f}} C[\mathsf{S}_f\,\underline{\mathsf{x}}]$ for some context C, then $|x| \leq B(|f|, |w|)$.

Proof. By way of a contradiction, suppose there exist f, w, and x such that $\mathsf{F}\,\mathsf{S}_f\,\underline{\mathsf{w}} \rightarrow^*_{\mathcal{R}_{+f}} C[\mathsf{S}_f\,\underline{\mathsf{x}}]$ for some context C, and $|x| > B(|f|, |w|)$. Let us now construct an alternative oracle: let $\mathsf{0} : \mathsf{nat}, \mathsf{s} : \mathsf{nat} \Rightarrow \mathsf{nat}, \mathsf{S}'_f : \mathsf{word} \Rightarrow \mathsf{word}$ and $\mathsf{helper} : \mathsf{nat} \Rightarrow \mathsf{nat} \Rightarrow \mathsf{nat}$, and for $N := D(|f|, |w|)$, let $\mathcal{R}'_{f,w}$ be given by:

$$\begin{array}{lllll} \mathsf{S}'_f\,\underline{\mathsf{x}} & \rightarrow & \underline{\mathsf{f(x)}} & \text{if } |x| \leq B(|f|, |w|) & \mathsf{helper}\,\mathsf{0}\,y \rightarrow y \\ \mathsf{S}'_f\,\underline{\mathsf{x}} & \rightarrow & \mathsf{helper}\,\ulcorner \mathsf{N} \urcorner\,\underline{\mathsf{f(x)}} & \text{otherwise} & \mathsf{helper}\,(\mathsf{s}\,x)\,y \rightarrow \mathsf{helper}\,x\,y \end{array}$$

Where $\ulcorner \mathsf{N} \urcorner$ is the unary number encoding of N, as introduced in Section 2.1. Notice that by definition the rules for S'_f will produce $\underline{\mathsf{f}(\mathsf{x})}$ in a single step if $|x| \leq B(|f|, |w|)$ but they will take $N+2$ steps otherwise. Also observe that S_f and S'_f behave the same; that is, $\mathsf{S}_f\,\underline{\mathsf{x}}$ and $\mathsf{S}'_f\,\underline{\mathsf{x}}$ have the same normal form on any input $\underline{\mathsf{x}}$. We extend the interpretation function of the original signature with:

$$\mathcal{J}^{\mathsf{c}}_{\mathsf{S}'_f} = \boldsymbol{\lambda} x. \begin{cases} 1 & \text{if } x \leq B(|f|, |n|) \\ N+2 & \text{if } x > B(|f|, |n|) \end{cases} \qquad \mathcal{J}^{\mathsf{s}}_{\mathsf{S}'_f} = \mathcal{J}^{\mathsf{s}}_{\mathsf{S}_f}$$

$$\mathcal{J}^{\mathsf{c}}_{\mathsf{helper}} = \boldsymbol{\lambda} xy.x+1 \qquad \mathcal{J}^{\mathsf{s}}_{\mathsf{helper}} = \boldsymbol{\lambda} xy.y \qquad \mathcal{J}^{\mathsf{s}}_{\mathsf{0}} = 0 \qquad \mathcal{J}^{\mathsf{s}}_{\mathsf{s}} = \boldsymbol{\lambda} x.x+1$$

We easily see that this orients all rules in $\mathcal{R}_{f,w}$. Then, by Lemma 5.3, $\mathsf{cost}(\mathsf{F}\,\mathsf{S}'_f\,\underline{\mathsf{w}}) \leq P(\mathcal{J}^{\mathsf{c}}_{\mathsf{S}'_f}, \mathcal{J}^{\mathsf{s}}_{\mathsf{S}'_f}, [\![\underline{\mathsf{w}}]\!]^{\mathsf{s}}) = P(\mathcal{J}^{\mathsf{c}}_{\mathsf{S}_f}, \mathcal{J}^{\mathsf{s}}_{\mathsf{S}_f}, [\![\underline{\mathsf{w}}]\!]^{\mathsf{s}}) \leq D(|f|, |w|) = N$. Yet, as we have $\mathsf{F}\,\mathsf{S}_f\,\underline{\mathsf{w}} \to^*_{\mathcal{R}+f} C[\mathsf{S}_f\,\underline{\mathsf{x}}]$, we also have $\mathsf{F}\,\mathsf{S}_f\,\underline{\mathsf{w}} \to_{\mathcal{R}\cup\mathcal{R}'_{f,w}} C'[\mathsf{S}'_f\,\underline{\mathsf{x}}]$, where C' is obtained from C by replacing all occurrences of S_f by S'_f. Since $|x| > B(|f|, |w|)$ by assumption, the reduction $\mathsf{F}\,\mathsf{S}'_f\,\underline{\mathsf{w}} \to^*_{\mathcal{R}\cup\mathcal{R}'_{f,w}} C[\mathsf{S}'_f\,\underline{\mathsf{w}}] \to^*_{\mathcal{R}\cup\mathcal{R}_{f,w'}} C[\underline{\mathsf{f}(\mathsf{x})}]$ takes strictly more than N steps, contradicting Corollary 3.5. □

5.3. **Graph Rewriting.** Lemma 5.1 guarantees that if $\mathcal{R}$ is compatible with a suitable interpretation, then at most polynomially many $\mathcal{R}_{+f}$-steps can be performed starting in $\mathsf{F}\,\mathsf{S}_f\,\underline{\mathsf{w}}$. However, as observed in Section 5.1, this does not yet imply that a type-2 functional computed by an STRS with such an interpretation is in BFF. To simulate a reduction on an OTM, we must find a representation whose size does not increase too much in any given step. The answer is *graph rewriting*.

Definition 5.5. A **term graph** for a signature Σ is a tuple $(V, \mathtt{label}, \mathtt{succ}, \Lambda)$ with V a finite nonempty set of vertices; $\Lambda \in V$ a designated vertex called the *root*; $\mathtt{label} : V \longrightarrow \Sigma \cup \{@\}$ a partial function with @ fresh; and $\mathtt{succ} : V \longrightarrow V^*$ a total function such that $\mathtt{succ}(v) = v_1 v_2$ when $\mathtt{label}(v) = @$ and $\mathtt{succ}(v) = \varepsilon$ otherwise. We view this as a directed graph, with an edge from v to v' if $v' \in \mathtt{succ}(v)$, and require that this graph is *acyclic* (i.e., there is no path from any v to itself). Given term graph G, we will often directly refer to $V_G, \mathtt{label}_G$, etc.

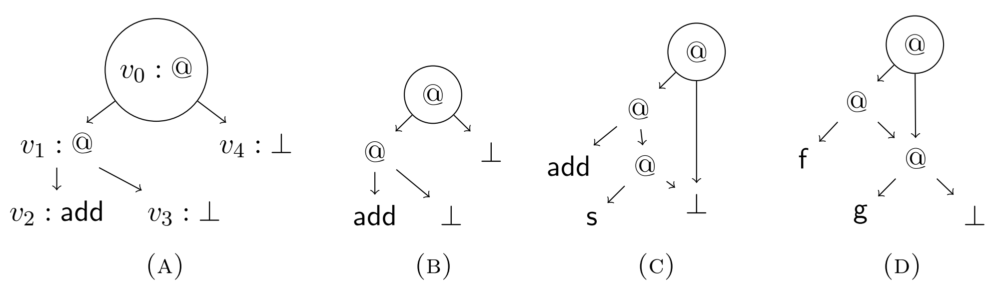

FIGURE 1. A term graph, its simplified version, and two graphs with sharing

Term graphs can be denoted visually in an intuitive way. For example, using Σ from Example 2.1, the graph with $V = \{v_0, \ldots, v_4\}$, $\mathtt{label} = \{v_0, v_1 \mapsto @,\ v_2 \mapsto \mathsf{add}\}$, $\mathtt{succ} = \{v_0 \mapsto v_1 v_4,\ v_1 \mapsto v_2 v_3,\ v_3, v_4, v_5 \mapsto \varepsilon\}$ amd $\Lambda = v_0$ is pictured in Fig. 1a. We use $\bot$ to indicate unlabeled vertices and a circle for Λ. We will typically omit vertex names, as done in Fig. 1b. Note that the definition allows multiple vertices to have the same vertex as successor; these successor vertices with in-degree > 1 are *shared*. Two examples are denoted in Figures 1c and 1d.

Each term has a natural representation as a tree. Formally, for a term s we let $[s]_{\mathbb{G}} = (\mathtt{pos}(s), \mathtt{label}, \mathtt{succ}, \sharp)$ where $\mathtt{label}(p) = @$ if $s|_p = s_1 s_2$ and $\mathtt{label}(p) = \mathsf{f}$ if $s|_p = \mathsf{f}$; $\mathtt{label}(p)$ is not defined if $s|_p$ is a variable; and $\mathtt{succ}(p) = (1 \cdot p)(2 \cdot p)$ if $s|_p = s_1\, s_2$ and $\mathtt{succ}(p) = \varepsilon$ otherwise. Essentially, $[s]_{\mathbb{G}}$ maintains the positioning structure of s and forgets variable names. For example, Figure 1b denotes both $[\mathsf{add}\, x\, y]_{\mathbb{G}}$ and $[\mathsf{add}\, x\, x]_{\mathbb{G}}$.

Our next step is to *reduce* term graphs using rules. We limit interest to *left-linear* rules, which includes all rules in $\mathcal{R}_{+f}$ (as $\mathcal{R}$ is orthogonal, and the rules in $\mathcal{R}_f$ are ground). To define reduction, we will need some helper definitions.

Definition 5.6. Let $G = (V, \mathtt{label}, \mathtt{succ}, \Lambda)$ and $v \in V$. The **subgraph** $\mathsf{reach}(G, v)$ of G with root v is the term graph $(V', \mathtt{label}', \mathtt{succ}', v)$ where V' contains those nodes $v' \in V$ such that there exists a path from v to v' and $\mathtt{label}', \mathtt{succ}'$ are respectively the restrictions of $\mathtt{label}$ and $\mathtt{succ}$ to V'.

Definition 5.7. A **homomorphism** between two term graphs G and H is a function $\phi : V_G \longrightarrow V_H$ with $\phi(\Lambda_G) = \Lambda_H$, and for $v \in V_G$ such that $\mathtt{label}_G(v)$ is defined, $\mathtt{label}_H(\phi(v)) = \mathtt{label}_G(v)$ and $\mathtt{succ}_H(\phi(v)) = \phi(v_1) \dots \phi(v_k)$ when $\mathtt{succ}_G(v) = v_1 \dots v_k$. (If $\mathtt{label}_G(v)$ is undefined, $\mathtt{label}_H(\phi(v))$ and $\mathtt{succ}_H(\phi(v))$ may be anything.)

Definition 5.8. A **redex** in G is a triple (ρ, v, ϕ) consisting of some rule $\rho = \ell \to r \in \mathcal{R}_{+f}$, a vertex v in V_G, and a homomorphism $\phi : [\ell]_{\mathbb{G}} \longrightarrow \mathsf{reach}(G, v)$.

Lemma 5.9. Let $(\ell \to r, v, \phi)$ be a redex in G, $x \in \mathtt{vars}(\ell)$, and let a_x be the corresponding vertex in $[\ell]_{\mathbb{G}}$. Then $v \neq \phi(a_x)$, nor is there a path from $\phi(a_x)$ to v.

Proof. Note that ℓ is not a variable, so there is a (non-empty) path from the root of $[\ell]_{\mathbb{G}}$ to each a_x; hence, by definition of homomorphism, there is a path from $\phi(\Lambda_{[\ell]_{\mathbb{G}}}) = v$ to each $\phi(a_x)$. Since G is acyclic, this means that $\phi(a_x)$ cannot be v, nor can there be a path from $\phi(a_x)$ back to v. □

Definition 5.10. Let G be a term graph and v_1, v_2 vertices in G such that no path exists from v_2 to v_1. The **redirection** of v_1 to v_2 is the term graph $G[v_1 \gg v_2] \equiv (V_G, \mathtt{label}_G, \mathtt{succ}'_G, \Lambda'_G)$ with $\mathtt{succ}'_G$ and Λ'_G given by:

$$\mathtt{succ}'_G(v)_i = \begin{cases} v_2, & \text{if } \mathtt{succ}_G(v)_i = v_1 \\ \mathtt{succ}_G(v)_i, & \text{otherwise} \end{cases} \qquad \Lambda'_G = \begin{cases} v_2 & \text{if } \Lambda_G = v_1 \\ \Lambda_G & \text{otherwise} \end{cases}$$

That is, we replace every reference to v_1 by a reference to v_2, which does not introduce any cycles because there is no path from v_2 to v_1. With these definitions in hand, we can define *contraction* of term graphs:

Definition 5.11. Let G be a term graph, and (ρ, v, ϕ) a redex in G with $\rho \in \mathcal{R}_{+f}$, such that no other vertex v' in $\mathsf{reach}(G, v)$ admits a redex (so v is an *innermost redex position*). Denote a_x for the position of variable x in ℓ, and recall that a_x is a vertex in $[\ell]_{\mathbb{G}}$. By left-linearity, a_x is unique for $x \in \mathtt{vars}(\ell)$. The **contraction** of (ρ, v, ϕ) in G is the term graph J produced after the following steps: H (building), I (redirection), and J (garbage collection).

(building): Let $H = (V_H, \mathtt{label}_H, \mathtt{succ}_H, \Lambda_G)$ where:

- $V_H = V_G \uplus \{\overline{p} \in \mathtt{pos}(r) \mid r|_p \text{ is not a variable}\}$ ($\uplus$ means disjoint union);
- for $v \in V_G$: $\mathtt{label}_H(v) = \mathtt{label}_G(v)$ and $\mathtt{succ}_H(v) = \mathtt{succ}_G(v)$

- for $p \in V_H$ with $r|_p$ not a variable:
  - $\mathtt{label}_H(\overline{p}) = \mathsf{f}$ if $r|_p = \mathsf{f}$ and $\mathtt{label}_H(\overline{p}) = @$ otherwise
  - $\mathtt{succ}_H(\overline{p}) = \varepsilon$ if $r|_p = \mathsf{f}$; otherwise, $\mathtt{succ}_H(\overline{p}) = \psi(1 \cdot p)\psi(2 \cdot p)$
    Here, $\psi(q) = \overline{q}$ if $r|_q$ is not a variable; if $r|_q = x$ then $\psi(q) = \phi(a_x)$.

(redirection): If r is a variable x (so $H = G$), then let $I = G[v \gg \phi(a_x)]$. Otherwise, let $I = H[v \gg \overline{\sharp}]$, so with all references to v redirected to the root vertex for r. (Note that in the first case, there is no path from $\phi(a_x)$ to v by Lemma 5.9; and in the second case, every path from $\overline{\sharp}$ into V_G passes through some $\phi(a_y)$, so again by Lemma 5.9 there is no path from $\overline{\sharp}$ to v, and therefore the redirection is allowed.)

(garbage collection): Let $J := \mathsf{reach}(I, \Lambda_I)$ (so remove unreachable vertices).

We then write $G \rightsquigarrow J$ in one step, and $G \rightsquigarrow^n J$ for the n-step reduction.

We illustrate this with two examples. First, we aim to rewrite the graph of Fig. 2a with a rule $\mathsf{add}\,\mathsf{0}\,y \to y$ at vertex v. Since the right-hand side is a variable, the building phase does nothing. The result of the redirection phase is given in Fig. 2b, and the result of the garbage collection in Fig. 2c.

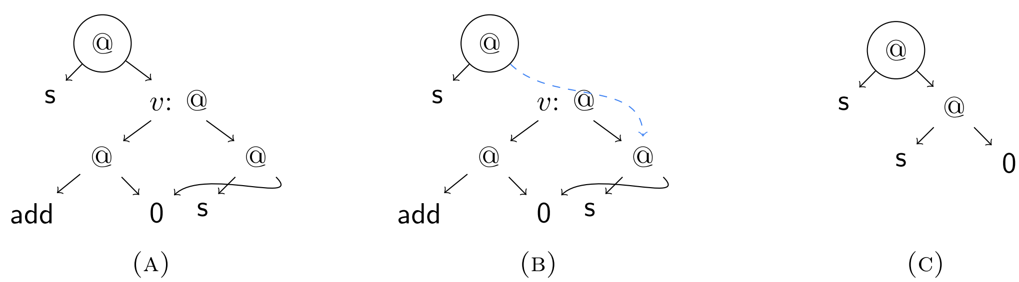

FIGURE 2. Reducing a graph with the rule $\mathsf{add}\,\mathsf{0}\,y \to y$

Second, we consider a reduction of the graph in Fig. 3a by $\mathsf{mult}\,(\mathsf{s}\,x)\,y \to \mathsf{add}\,y\,(\mathsf{mult}\,x\,y)$. Unlike the previous example, this graph has sharing. Fig. 3b shows the result of the building phase, with the vertices and edges added during this phase in red. Redirection sets the root to the squared node (the root of the right-hand side), and the result after garbage collection is in Fig. 3c.

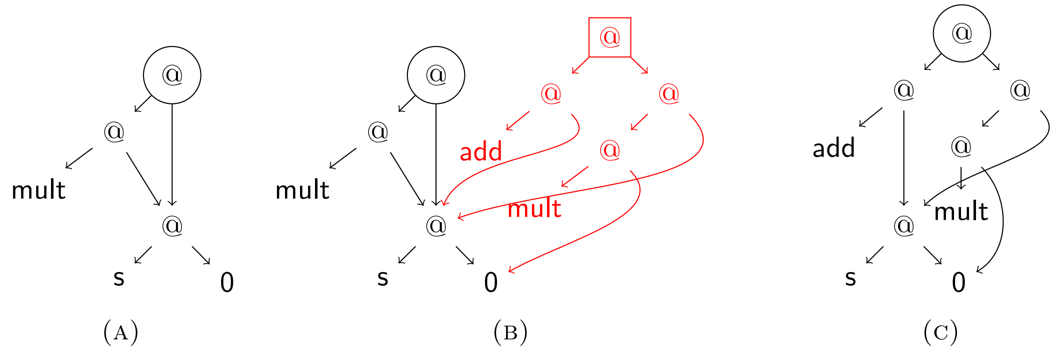

FIGURE 3. Reducing a term graph with substantial sharing

Note that, even when a term graph G is not a tree, we can find a corresponding term: we assign a variable $var(v)$ to each unlabeled vertex v in G, and let:

$$\theta(v) = \begin{cases} \theta(v_1)\,\theta(v_2) & \text{if } \texttt{label}(v) = @ \text{ and } \texttt{succ}(v) = v_1 v_2 \\ \mathsf{f} & \text{if } \texttt{label}(v) = \mathsf{f} \\ var(v) & \text{if } \texttt{label}(v) \text{ is undefined} \end{cases}$$

Then we may define $[G]_{\mathbb{G}}^{-1} = \theta(\Lambda_G)$. For a linear term (that is, one in which no variable occurs more than once) , clearly $[[s]_{\mathbb{G}}]_{\mathbb{G}}^{-1} = s$ modulo variable renaming. We make the following observation:

Lemma 5.12. Assume given a term graph G such that there is a path from Λ_G to every vertex in V_G, and let $[G]_{\mathbb{G}}^{-1} = s$. If $G \rightsquigarrow H$ then $[G]_{\mathbb{G}}^{-1} \rightarrow_{\mathcal{R}_{+f}}^{+} [H]_{\mathbb{G}}^{-1}$. Moreover, if $s \rightarrow_{\mathcal{R}_{+f}} t$ for some t, then there exists H such that $G \rightsquigarrow H$.

Consequently, if $\rightarrow_{\mathcal{R}_{+f}}$ is terminating, then so is $\rightsquigarrow$; and if $[s]_{\mathbb{G}} \rightsquigarrow^n G$ for some ground term s then $s \rightarrow_{\mathcal{R}_{+f}}^{*} [G]_{\mathbb{G}}^{-1}$ in at least n steps. Notice that if G does not admit any redex, then $[G]_{\mathbb{G}}^{-1}$ is in normal form. Moreover, since $\mathcal{R}_{+f} = \mathcal{R} \cup \mathcal{R}_f$ is orthogonal (as $\mathcal{R}$ is orthogonal and the $\mathcal{R}_f$ rules are non-overlapping) and therefore confluent, this is the *unique* normal form of s. We conclude:

Corollary 5.13. If $[\mathsf{F}\,\mathsf{S}_f\,\underline{\mathsf{w}}]_{\mathbb{G}} \rightsquigarrow^n G$, then $n \leq D(|f|, |w|)$; and if G cannot be reduced by $\rightsquigarrow$, then $[G]_{\mathbb{G}}^{-1} = \underline{\Psi(\mathsf{f}, \mathsf{w})}$.

5.4. **Bringing Everything Together.** We are now ready to complete the soundness proof following the recipe at the start of the section. Towards the third bullet point, we make the following observation.

Lemma 5.14. There is a constant a such that, whenever $G \rightsquigarrow H$ by a rule in $\mathcal{R}$, then $|H| \leq |G| + a$, where $|G|$ denotes the total number of nodes in the graph G.

Proof. In a step using a rule $\ell \rightarrow r$, the number of nodes in the graph can be increased at most by $|[r]_{\mathbb{G}}|$. As there are only finitely many rules in $\mathcal{R}$, we can let a be the number of nodes in the largest graph for a right-hand side r. □

To see that graph rewriting with S_f can be implemented in an efficient way, we observe that the size of any intermediate graph in the reduction $[\mathsf{G}\,\underline{\mathsf{w}}]_{\mathbb{G}} \rightarrow_{\mathcal{R}}^{+} [q]_{\mathbb{G}}$ is polynomially bounded by a second-order polynomial over $|f|, |w|$:

Lemma 5.15. There is a second-order polynomial Q such that if $[\mathsf{F}\,\mathsf{S}_f\,\underline{\mathsf{w}}]_{\mathbb{G}} \rightsquigarrow^* H$, then $|H| \leq Q(|f|, |w|)$.

Proof. Let $Q(F, x) := x + D(F, x) * (a + F(B(F, x)))$, where D is the polynomial from Lemma 5.1, a is the constant from Lemma 5.14, and B is the polynomial from Section 5.2. This suffices, because there are at most $D(|f|, |w|)$ steps (Lemma 5.1, Corollary 5.13), each of which increases the graph size by at most $\max(a, |f|(B(|f|, |w|)))$ according to Lemmas 5.4 and 5.14. (Note that, by the Oracle Subterm lemma, B takes the size $|w|$ of the original input w as its second argument, *not* the current size of the term.) □

All in all, we are finally ready to prove the *soundness* side of the main theorem:

Theorem 5.16 (Soundness)**.** Let $\mathcal{R}$ be a finite orthogonal STRS admitting a polynomially bounded interpretation. If F computes a type-2 functional Ψ, then $\Psi \in \mathtt{BFF}$.

Proof. Given $(\mathbb{F}, \mathcal{R})$, we can construct an OTM M so that for a given $f \in W \longrightarrow W$, the machine M_f executed on $w \in W$ computes the normal form of $\mathsf{F}\,\mathsf{S}_f\,\underline{\mathsf{w}}$ under $\to_{\mathcal{R}_{+f}}$ using graph rewriting. We omit the details of the construction, but observe:

- that we can represent each graph in polynomial space in the size of the graph; simply, we can make use of an adjacency-matrix representation of the graph;
- that we simulate any rewriting step that does not call the oracle (so using a rule in $\mathcal{R}$) following the contraction algorithm we defined in Definition 5.11, which is clearly feasible to do in polynomial time in the size of the graph; of course this could in principle involve the creation of new nodes and edges, but the amount of them is statically bounded by the size of the underlying set or rewrite rules (excluding those involving the oracle).
- and that each oracle call (implemented in rewriting by a $\mathcal{R}_f$-step $\mathsf{S}_f\,\underline{\mathsf{x}} \to \underline{\mathsf{y}}$) is resolved by copying $\underline{\mathsf{x}}$ to the query tape, transitioning to the query state, and from the answer state copying $\underline{\mathsf{y}}$ from the answer tape to the main tape. By Lemma 5.4 this is doable in polynomial time in $|f|, |w|$ and the graph size.

By Lemma 5.15, graph sizes are bounded by a polynomial over $|f|, |w|$, so using the above reasoning, the same holds for the cost of each reduction step. In summary: the total cost of M_f running on w is bounded by a second-order polynomial in terms of $|f|$ and $|w|$. As M_f simulates $\mathcal{R}_{+f}$ via graph rewriting and $\mathcal{R}_{+f}$ computes Ψ, M also computes Ψ. By Definition 2.5, Ψ is in $\mathtt{BFF}_2$. □

6. Completeness

Recall from Section 4 that to prove completeness we have to show the following: if a given type-2 functional Ψ is in $\mathtt{BFF}_2$, then there exists an orthogonal STRS computing Ψ and admitting a polynomially bounded interpretation. In this section, we prove this implication by providing an encoding of polynomial time Oracle Turing Machines as STRSs that admit a polynomially bounded interpretation.

The encoding is divided into three steps. In Section 6.1, we define the function symbols that allow us to encode any possible machine configuration as terms. In Section 6.2, we encode machine transitions as reduction rules that rewrite configuration terms. This allows us to simulate one transition in an OTM as one or more rewriting steps on the corresponding rewrite system. Lastly, we design an STRS which simulates a complete execution of an OTM in polynomially many steps. Achieving this polynomial bound is non-trivial and is done in Section 6.3 and Section 6.4.

Henceforth, we assume given a fixed OTM M, and a second-order polynomial P_M, such that M operates in time P_M. For simplicity, we assume the machine has only three tapes (one input/output tape, one query tape, one answer tape); that each non-oracle transition only operates on one tape (i.e., reading/writing and moving the tape head); and that we only have tape symbols $\{0, 1, \mathtt{B}\}$.

6.1. **Encoding Machine Configurations as Terms.** Recall from Section 4 that we have $\mathsf{o}, \mathsf{i} : \mathsf{bit}$, $:: : \mathsf{bit} \Rightarrow \mathsf{word} \Rightarrow \mathsf{word}$ and $[] : \mathsf{word}$, which are the basic constructors for encoding binary words as terms. To represent a (partial) tape, we also introduce $\mathsf{b} : \mathsf{bit}$ for the blank symbol. Now for instance a tape with content $011\mathtt{B}01\mathtt{BB}\cdots$ (followed by infinitely many blanks) may be represented as the list $[\mathsf{o};\mathsf{i};\mathsf{i};\mathsf{b};\mathsf{o};\mathsf{i}]$ of type word. We may also add an arbitrary number of blanks at the end of the representation; e.g., $[\mathsf{o};\mathsf{i};\mathsf{i};\mathsf{b};\mathsf{o};\mathsf{i};\mathsf{b};\mathsf{b}]$.

We can think of a *tape configuration* — the combination of a tape and the position of the tape head — as a finite word $w_1 \dots w_{p-1} \# w_p w_{p+1} \dots w_k$ (followed by infinitely many blanks). Here, the tape's head is reading the symbol w_p. We can split this tape into two components: the *left* word $w_1 \dots w_{p-1}$, and the *right* word $w_p \dots w_k$. To represent a tape configuration as a term, we introduce three symbols:

$$\mathsf{L} : \mathsf{word} \Rightarrow \mathsf{left} \qquad \mathsf{R} : \mathsf{word} \Rightarrow \mathsf{right} \qquad \mathsf{split} : \mathsf{left} \Rightarrow \mathsf{right} \Rightarrow \mathsf{tape}$$

Here, L, R hold the content[1] of the left and right split of the tape, respectively. For convenience in rewriting transitions, later on, we encode the left side of the split in reverse order. More precisely, we encode the configuration $w_1 \dots w_{p-1} \# w_p w_{p+1} \dots w_k$ as the term

$$\mathsf{split}\,(\mathsf{L}\,[w_{p-1};\dots;w_2;w_1])\,(\mathsf{R}\,[w_p;\dots;w_{k-1};w_k])$$

The symbol currently being read is the first element of the list below R; in case of $\mathsf{R}\,[]$, this symbol is $\mathtt{B}$. For a concrete example, a tape configuration $1\mathtt{B}0\#10$ is represented by: $\mathsf{split}\,(\mathsf{L}\,[\mathsf{o};\mathsf{b};\mathsf{i}])\,(\mathsf{R}\,[\mathsf{i};\mathsf{o}])$. Since we have assumed an OTM with three tapes, a configuration of the machine at any moment is a tuple (q, t_1, t_2, t_3), with q a state and t_1, t_2, t_3 tape configurations. To represent machine configurations, we introduce, for each state q, a symbol $\mathsf{q} : \mathsf{tape} \Rightarrow \mathsf{tape} \Rightarrow \mathsf{tape} \Rightarrow \mathsf{config}$. Thus, a configuration (q, t_1, t_2, t_3) is represented by a term $\mathsf{q}\, T_1\, T_2\, T_3$.

Example 6.1. The initial configuration for a machine M_f on input w is a tuple of the form $(q_0, \#w, \#\mathtt{B}, \#\mathtt{B})$. This is represented by the term

$$\mathsf{initial}(w) := \mathsf{q}_0\,(\mathsf{split}\,(\mathsf{L}\,[])\,(\mathsf{R}\,\underline{\mathsf{w}}))\,(\mathsf{split}\,(\mathsf{L}\,[])\,(\mathsf{R}\,[]))\,(\mathsf{split}\,(\mathsf{L}\,[])\,(\mathsf{R}\,[]))$$

To interpret the symbols from this section, we let $(\mathcal{S}_\iota, \sqsupseteq_\iota) := (\mathbb{N}, \geq)$ for all ι, let $\mathcal{J}^{\mathsf{c}}_{\mathsf{f}} = \boldsymbol{\lambda} x_1 \dots x_m.0$ whenever f takes m arguments, and for the sizes:

$$\begin{array}{lllllllll} \mathcal{J}^{\mathsf{s}}_{\mathsf{o}} & = & 0 & \mathcal{J}^{\mathsf{s}}_{[]} & = & 0 & \mathcal{J}^{\mathsf{s}}_{::} & = & \boldsymbol{\lambda} xy.x+y+1 \\ \mathcal{J}^{\mathsf{s}}_{\mathsf{i}} & = & 0 & \mathcal{J}^{\mathsf{s}}_{\mathsf{L}} & = & \boldsymbol{\lambda} x.x & \mathcal{J}^{\mathsf{s}}_{\mathsf{split}} & = & \boldsymbol{\lambda} xy.x+y \\ \mathcal{J}^{\mathsf{s}}_{\mathsf{b}} & = & 0 & \mathcal{J}^{\mathsf{s}}_{\mathsf{R}} & = & \boldsymbol{\lambda} x.x & \mathcal{J}^{\mathsf{s}}_{\mathsf{q}} & = & \boldsymbol{\lambda} xyz.x+y \qquad \text{(for all states } q\text{)} \end{array}$$

Hence, we satisfy the size component of the requirements in Definition 4.4. We have $[\![\underline{\mathsf{w}}]\!]^{\mathsf{s}} = |w|$; the size of a tape configuration $w_1 \dots w_{p-1} \# w_p \dots w_k$ is k, and the size of a configuration is the size of its first and second tapes combined. We do *not* include the third tape, as it does not directly affect either the result yielded by the final configuration (this is read from the first tape), nor the size of a word the oracle f is applied on.

[1] While we technically do not need these two constructors (we could have $\mathsf{split} : \mathsf{word} \Rightarrow \mathsf{word} \Rightarrow \mathsf{tape}$), they serve to make configurations more human-readable.

6.2. **Encoding Machine Transitions as Rules.** A single step in an OTM can either be an oracle call (a transition from the `query` state to the `answer` state), or a traditional step: we assume that an OTM M has a fixed set $\mathcal{T}$ of *transitions* $q \xRightarrow[t]{r/i,\ d} l$ where q is the *input state*, l the *output state*, $t \in \{1,2,3\}$ the tape considered (recall that we have assumed that a non-oracle transition only operates on one tape), $r, i \in \{0, 1, \mathtt{B}\}$ respectively the symbol being read and the symbol being written, and $d \in \{L, R\}$ the direction for the read head of tape t to move. We will model the computation of M as rules that simulate the small step semantics of the machine. Let us describe such rules as follows:

- To encode a single transition, let $\mathsf{step} : (\mathsf{word} \Rightarrow \mathsf{word}) \Rightarrow \mathsf{config} \Rightarrow \mathsf{config}$. For any transition of the form $q \xRightarrow[1]{r/i,\ L} l$ (so a transition operating on tape 1, and moving left), we introduce a rule (where we write $\underline{0} = \mathsf{o}$, $\underline{1} = \mathsf{i}$, $\underline{\mathtt{B}} = \mathsf{b}$):
$$\mathsf{step}\, F\, (\mathtt{q}\, (\mathsf{split}\, (\mathsf{L}\, (x{::}y))\, (\mathsf{R}\, (\underline{\mathsf{r}}{::}z)))\, u\, v) \to \mathtt{l}\, (\mathsf{split}\, (\mathsf{L}\, y)\, (\mathsf{R}\, (x{::}\underline{\mathsf{i}}{::}z)))\, u\, v$$
Moreover, for transitions $q \xRightarrow[1]{\mathtt{B}/w,\ L} l$ (so where $\mathtt{B}$ is read), we add a rule:
$$\mathsf{step}\, F\, (\mathtt{q}\, (\mathsf{split}\, (\mathsf{L}\, (x{::}y))\, (\mathsf{R}\, []))\, u\, v) \to \mathtt{l}\, (\mathsf{split}\, (\mathsf{L}\, y)\, (\mathsf{R}\, (x{::}\underline{\mathsf{i}}{::}[])))\, u\, v$$
These rules respectively handle the steps where a tape configuration is changed from $u_1 \dots u_{p-1} u_p \# r u_{p+2} \dots u_k$ to $u_1 \dots u_{p-1} \# u_p i u_{p+2} \dots u_k$, and where a tape configuration is changed from $u_1 \dots u_k \#$ to $u_1 \dots \# u_k i$.
- Transitions where $d = R$, or on the other two tapes, are encoded similarly.
- Next, we encode oracle calls. Recall that to query the machine for the value of f at u, we write u on the second tape, move its head to the leftmost position, and enter the query state. Then, the content of this tape is erased and the image of f over u is written in the third tape. Visually, this step is represented as:
$$(\mathtt{query}, \langle \text{tape}_1 \rangle, v_1 \dots v_p \# \underline{\mathtt{u}}\mathtt{B} \dots, \langle \text{tape}_3 \rangle) \rightsquigarrow (\mathtt{answer}, \langle \text{tape}_1 \rangle, \#\mathtt{B}, \#\underline{\mathsf{f}(\mathsf{u})})$$
This is implemented by the following rules:
$$\begin{aligned} \mathsf{step}\, F\, (\mathtt{query}\, t_1\, (\mathsf{split}\, x\, (\mathsf{R}\, y))\, t_3) \quad \to \quad & \mathtt{answer}\, t_1\, (\mathsf{split}\, (\mathsf{L}\, [])\, (\mathsf{R}\, [])) \\ & (\mathsf{split}\, (\mathsf{L}\, [])\, (\mathsf{R}\, (F\, (\mathsf{clean}\, y)))) \end{aligned}$$
$$\begin{array}{rclrcl} \mathsf{clean}\, (\mathsf{o}{::}x) & \to & \mathsf{o}{::}(\mathsf{clean}\, x) & \mathsf{clean}\, (\mathsf{b}{::}x) & \to & [] \\ \mathsf{clean}\, (\mathsf{i}{::}x) & \to & \mathsf{i}{::}(\mathsf{clean}\, x) & \mathsf{clean}\, [] & \to & [] \end{array}$$
Here, $\mathsf{clean} : \mathsf{word} \Rightarrow \mathsf{word}$ turns a word that may have blanks in it into a bitstring, by reading until the next blank; for instance replacing $[\mathsf{o}; \mathsf{i}; \mathsf{b}; \mathsf{i}]$ by $[\mathsf{o}; \mathsf{i}]$.

The various step rules and the clean rules defined above are non-overlapping because we consider *deterministic* OTMs. Additionally, they are also left-linear. We can orient such rules as follows:
$$\begin{array}{rclrcl} \mathcal{J}^{\mathsf{s}}_{\mathsf{clean}} & = & \boldsymbol{\lambda} x.x & \mathcal{J}^{\mathsf{c}}_{\mathsf{clean}} & = & \boldsymbol{\lambda} x.x + 1 \\ \mathcal{J}^{\mathsf{s}}_{\mathsf{step}} & = & \boldsymbol{\lambda} F x.x + 1 & \mathcal{J}^{\mathsf{c}}_{\mathsf{step}} & = & \boldsymbol{\lambda} F^c F^s x.F^c(x) + x + 2 \end{array}$$
Note that $\mathcal{J}^{\mathsf{s}}_{\mathsf{step}}$ is so simple because the size of a configuration does not include the size of the answer tape. From these simulation rules, the following result is straightforward.

Lemma 6.2. Let M_f be an OTM and C, D be machine configurations of M_f such that $C \rightsquigarrow D$. Then $\mathsf{step}\, \mathsf{S}_f\, [C] \to^+_{\mathcal{R}} [D]$, where $[C]$ is the term encoding of C.

6.3. **A Bound on the Number of Steps.** To generalize from performing a single step of the machine to tracing a full computation on the machine level, the natural idea would be to define rules such as:

$$\begin{array}{rcll} \mathsf{execute}\,F\,(\mathsf{q}\,x\,y\,z) & \to & \mathsf{execute}\,F\,(\mathsf{step}(\mathsf{q}\,x\,y\,z)) & \text{for } \mathsf{q} \neq \mathsf{end} \\ \mathsf{execute}\,F\,(\mathsf{end}\,(\mathsf{split}\,(\mathsf{L}\,x)\,(\mathsf{R}\,w))\,y\,z) & \to & \mathsf{clean}\,w & \end{array}$$

Then, reducing $\mathsf{execute}\,\mathsf{S}_f\,\mathsf{initial}(w)$ to normal form simulates a full OTM execution of M_f on input w. Unfortunately, this rule does not admit an interpretation, as it may be non-terminating. A solution could be to give $\mathsf{execute}$ an additional argument $\ulcorner\mathsf{N}\urcorner$ suggesting an execution in at most N steps; this argument would ensure termination, and could be used to find an interpretation.

The challenge, however, is to compute a bound on the number of steps in the OTM: the obvious thought is to compute $P_M(|f|, |w|)$, but this cannot in general be done in polynomial time because the STRS does not have access to $|f|$: since $|f|(i) = \max\{x \in \mathbb{N} \mid |x| \leq i\}$, there are exponentially many choices for x.

To solve this, and following [KC96, Proposition 2.3], we observe that it suffices to know a bound for $f(x)$ for only those x on which the oracle is actually questioned. That is, for $A \subseteq W$, let $|f|_A = \boldsymbol{\lambda} n.\max\{|f(x)| \mid x \in A \wedge |x| \leq n\}$. Then:

Lemma 6.3. Suppose an OTM M_f runs in time bounded by $P_M(|f|, |w|)$ on input w. If M_f transitions in N steps from its initial state to some configuration C, calling the oracle only on words in $A \subseteq W$, then $N \leq P_M(|f|_A, |w|)$.

Proof sketch. We construct f' with $f'(x) = 0$ if $x \notin A$ and $f'(x) = f(x)$ if $x \in A$. Then $|f'| = |f|_A$, and $M_{f'}$ runs the same on input w as M_f does. □

Now, for A encoded as a term A (using symbols $\emptyset : \mathsf{set}$, $\mathsf{scons} : \mathsf{word} \Rightarrow \mathsf{set} \Rightarrow \mathsf{set}$), we can compute $|f|_A$ using the rules below, where we use unary integers as in Example 2.1 ($\mathsf{0} : \mathsf{nat}, \mathsf{s} : \mathsf{nat} \Rightarrow \mathsf{nat}$), and defined symbols $\mathsf{len} : \mathsf{word} \Rightarrow \mathsf{nat}$, $\mathsf{max} : \mathsf{nat} \Rightarrow \mathsf{nat} \Rightarrow \mathsf{nat}$, $\mathsf{limit} : \mathsf{word} \Rightarrow \mathsf{nat} \Rightarrow \mathsf{word}$, $\mathsf{retif} : \mathsf{word} \Rightarrow \mathsf{nat} \Rightarrow \mathsf{word} \Rightarrow \mathsf{word}$, $\mathsf{tryapply} : (\mathsf{word} \Rightarrow \mathsf{word}) \Rightarrow \mathsf{word} \Rightarrow \mathsf{nat} \Rightarrow \mathsf{nat}$, $\mathsf{tryall} : (\mathsf{word} \Rightarrow \mathsf{word}) \Rightarrow \mathsf{set} \Rightarrow \mathsf{nat} \Rightarrow \mathsf{nat}$. By design, $\mathsf{retif}\,\underline{\mathsf{x}}\,\ulcorner\mathsf{n}\urcorner\,\underline{\mathsf{y}}$ reduces to $\underline{\mathsf{y}}$ if $|x| \leq n$ and to $[]$ otherwise; $\mathsf{tryapply}\,\mathsf{S}_f\,\underline{\mathsf{x}}\,\ulcorner\mathsf{n}\urcorner$ reduces to the unary encoding of $|F|_{\{x\}}(n)$ and $\mathsf{tryall}\,\mathsf{S}_f\,\mathsf{a}\,\ulcorner\mathsf{n}\urcorner$ yields $|F|_A(n)$.

$$\begin{array}{rclrclrcl} \mathsf{len}\,[] & \to & \mathsf{0} & \mathsf{len}\,(x{::}y) & \to & \mathsf{s}\,(\mathsf{len}\,y) & & & \\ \mathsf{max}\,\mathsf{0}\,m & \to & m & \mathsf{max}\,(\mathsf{s}\,n)\,\mathsf{0} & \to & \mathsf{s}\,n & \mathsf{max}\,(\mathsf{s}\,n)\,(\mathsf{s}\,m) & \to & \mathsf{s}\,(\mathsf{max}\,n\,m) \\ \mathsf{limit}\,[]\,n & \to & [] & \mathsf{limit}\,(x{::}y)\,\mathsf{0} & \to & [] & \mathsf{limit}\,(x{::}y)\,(\mathsf{s}\,n) & \to & x{::}(\mathsf{limit}\,y\,n) \\ \mathsf{retif}\,[]\,n\,z & \to & z & \mathsf{retif}\,(x{::}y)\,\mathsf{0}\,z & \to & [] & \mathsf{retif}\,(x{::}y)\,(\mathsf{s}\,n)\,z & \to & \mathsf{retif}\,y\,n\,z \end{array}$$

$$\begin{array}{rclrcl} & & & \mathsf{tryapply}\,F\,a\,n & \to & \mathsf{len}\,(\mathsf{retif}\,a\,n\,(F\,(\mathsf{limit}\,a\,n))) \\ \mathsf{tryall}\,F\,\emptyset\,n & \to & \mathsf{0} & \mathsf{tryall}\,F\,(\mathsf{scons}\,a\,tl)\,n & \to & \mathsf{max}\,(\mathsf{tryapply}\,F\,a\,n)\,(\mathsf{tryall}\,F\,tl\,n) \end{array}$$

The cost–size interpretations of these rules are as follows.

$$\begin{array}{rclrcl} \mathcal{J}^{\mathsf{s}}_{\mathsf{len}} & = & \boldsymbol{\lambda} x.x & \mathcal{J}^{\mathsf{c}}_{\mathsf{len}} & = & \boldsymbol{\lambda} x.x+1 \\ \mathcal{J}^{\mathsf{s}}_{\mathsf{max}} & = & \boldsymbol{\lambda} nm.\max(n,m) & \mathcal{J}^{\mathsf{c}}_{\mathsf{max}} & = & \boldsymbol{\lambda} nm.n+1 \\ \mathcal{J}^{\mathsf{s}}_{\mathsf{limit}} & = & \boldsymbol{\lambda} xn.n & \mathcal{J}^{\mathsf{c}}_{\mathsf{limit}} & = & \boldsymbol{\lambda} xn.n+1 \\ \mathcal{J}^{\mathsf{s}}_{\mathsf{retif}} & = & \boldsymbol{\lambda} xnz.z & \mathcal{J}^{\mathsf{c}}_{\mathsf{retif}} & = & \boldsymbol{\lambda} xnz.n+1 \end{array}$$

It is easy to see that the corresponding rules are all oriented.

For $\mathsf{tryapply}$, note that $\mathsf{tryapply}\,F\,a\,\ulcorner\mathsf{n}\urcorner$ reduces to $\ulcorner|\mathsf{F(a)}|\urcorner$ if $|a| \leq n$, and to $\ulcorner\mathsf{0}\urcorner$ otherwise. Thus, it indeed returns exactly $|F|_{\{a\}}(n)$.

$$\mathcal{J}^{\mathsf{s}}_{\mathsf{tryapply}} = \boldsymbol{\lambda} Fan.F(n) \qquad \mathcal{J}^{\mathsf{c}}_{\mathsf{tryapply}} = \boldsymbol{\lambda} F^c F^s an.F^c(n) + F^s(n) + 2 * n + 4$$

We easily see that $[\![\mathsf{tryapply}\,a\,n]\!]^{\mathsf{s}} = [\![\mathsf{len}\,(\mathsf{retif}\,a\,n\,(F\,(\mathsf{limit}\,a\,n)))]\!]^{\mathsf{s}}$. As for the cost, note that

$$\begin{array}{rl} & \mathsf{cost}(\mathsf{len}\,(\mathsf{retif}\,a\,n\,(F\,(\mathsf{limit}\,a\,n)))) \\ = & [\![\mathsf{len}\,(\mathsf{retif}\,a\,n\,(F\,(\mathsf{limit}\,a\,n)))]\!]^{\mathsf{c}} + [\![\mathsf{retif}\,a\,n\,(F\,(\mathsf{limit}\,a\,n))]\!]^{\mathsf{c}} + \\ & [\![F\,(\mathsf{limit}\,a\,n]\!]^{\mathsf{c}} + [\![\mathsf{limit}\,a\,n]\!]^{\mathsf{c}} \\ = & (F^c(n) + 1) + (n+1) + F^s(n) + (n+1) = F^c(n) + F^s(n) + 2n + 3 \end{array}$$

Hence, also the $\mathsf{tryapply}$ rule is oriented.

To interpret sets and the apply rule, we use:

$$\begin{array}{rcl} \mathcal{J}^{\mathsf{s}}_{\emptyset} = 0 \quad \mathcal{J}^{\mathsf{c}}_{\emptyset} = 0 \quad \mathcal{J}^{\mathsf{s}}_{\mathsf{scons}} = \boldsymbol{\lambda} xy.y+1 \quad \mathcal{J}^{\mathsf{c}}_{\mathsf{scons}} = \boldsymbol{\lambda} xy.0 \\ \mathcal{J}^{\mathsf{s}}_{\mathsf{tryall}} & = & \boldsymbol{\lambda} Fan.F(n) \\ \mathcal{J}^{\mathsf{c}}_{\mathsf{tryall}} & = & \boldsymbol{\lambda} F^c F^s an.1 + a * (F^c(n) + 2 * F^s(n) + 2 * n + 6) \end{array}$$

To see that the rule is oriented, note:

$$\begin{array}{rcl} [\![\mathsf{tryall}\,F\,(\mathsf{scons}\,a\,tl)\,n]\!]^{\mathsf{s}} & = & F^s(n) \\ & = & \max(F^s(n), F^s(n)) \\ & = & [\![\mathsf{max}\,(\mathsf{tryapply}\,F\,a\,n)\,(\mathsf{tryall}\,F\,tl\,n)]\!]^{\mathsf{s}} \end{array}$$

and

$$\begin{array}{rl} & [\![\mathsf{tryall}\,F\,(\mathsf{scons}\,a\,tl)\,n]\!]^{\mathsf{c}} \\ = & 1 + (tl+1) * (F^c(n) + 2 * F^s(n) + 2 * n + 6) \\ = & 1 + tl * (F^c(n) + 2 * F^s(n) + 2 * n + 6) \\ & \quad + 1 * (F^c(n) + 2 * F^s(n) + 2 * n + 6) \\ = & [\![\mathsf{tryall}\,F\,tl\,n]\!]^{\mathsf{c}} + (F^c(n) + 2 * F^s(n) + 2 * n + 6) \\ = & [\![\mathsf{tryall}\,F\,tl\,n]\!]^{\mathsf{c}} + [\![\mathsf{tryapply}\,F\,a\,n]\!]^{\mathsf{c}} + F^s(n) + 2 \\ = & [\![\mathsf{tryall}\,F\,tl\,n]\!]^{\mathsf{c}} + [\![\mathsf{tryapply}\,F\,a\,n]\!]^{\mathsf{c}} + [\![\mathsf{max}\,(\mathsf{tryapply}\,F\,a\,n)\,(\mathsf{tryall}\,F\,tl\,n)]\!]^{\mathsf{c}} + 1 \\ > & \mathsf{cost}(\mathsf{max}\,(\mathsf{tryapply}\,F\,a\,n)\,(\mathsf{tryall}\,F\,tl\,n)) \end{array}$$

Importantly, the limit function ensures that, in $\mathsf{tryall}\,F\,n$ we never apply F to a word w with $|w| > n$. Therefore we can let $[\![\mathsf{A}]\!]^{\mathsf{s}} = |A|$, the number of words in A, and have $\mathcal{J}^{\mathsf{s}}_{\mathsf{tryall}} = \boldsymbol{\lambda} Fan.F(n)$ and $\mathcal{J}^{\mathsf{c}}_{\mathsf{tryall}} = \boldsymbol{\lambda} F^c F^s an.1 + a + F^c(n) + 2 * F^s(n) + 2 * n + 6$.

Now, for a given second-order polynomial P, fixed f, n, and a term A encoding a set $A \subseteq W$, we can construct a term $\Theta^P_{\mathsf{S}_f;\ulcorner\mathsf{n}\urcorner;\mathsf{A}}$ that computes $P(|f|_A, n)$ using tryall and the functions $\mathsf{add}, \mathsf{mult}$ from Example 2.1. By induction on P, we have $[\![\Theta^P_{\mathsf{S}_f;\ulcorner\mathsf{n}\urcorner;\mathsf{A}}]\!]^{\mathsf{s}} = P(|f|, n)$, while its cost is bounded by a polynomial over $|f|, n, |A|$.

6.4. **Finalising Execution.** Now, we can define execution in a way that can be bounded by a polynomial interpretation. We let $\mathsf{execute} : (\mathsf{word} \Rightarrow \mathsf{word}) \Rightarrow \mathsf{nat} \Rightarrow \mathsf{nnat} \Rightarrow \mathsf{nat} \Rightarrow \mathsf{set} \Rightarrow \mathsf{config} \Rightarrow \mathsf{word}$ and will define rules to reduce expressions $\mathsf{execute}\,F\,n\,m\,z\,a\,c$ where

- F is the function to be used in oracle calls.
- $n - 1$ is a bound on the number of steps that can be done before the next oracle call (or until the machine completes execution).

- m is essentially a natural number that represents the number of steps that have been done so far. We use a new sort nnat with function symbols $\mathsf{o} : \mathsf{nnat}$ and $\mathsf{n} : \mathsf{nnat} \Rightarrow \mathsf{nnat}$ because we will let $\mathcal{S}_{\mathsf{nnat}} = (\mathbb{N}, \leq)$, so ordered in the other direction. This will be essential to find an interpretation for $\mathsf{execute}$.
- z is a unary representation of $|w|$, where w is the input to the OTM.
- c is the current configuration.

Using helper symbols $\mathsf{F}' : (\mathsf{word} \Rightarrow \mathsf{word}) \Rightarrow \mathsf{nat} \Rightarrow \mathsf{config} \Rightarrow \mathsf{word}$, $\mathsf{execute}' : (\mathsf{word} \Rightarrow \mathsf{word}) \Rightarrow \mathsf{nat} \Rightarrow \mathsf{nnat} \Rightarrow \mathsf{nat} \Rightarrow \mathsf{set} \Rightarrow \mathsf{config} \Rightarrow \mathsf{word}$, $\mathsf{extract} : \mathsf{tape} \Rightarrow \mathsf{word}$ and $\mathsf{minus} : \mathsf{nat} \Rightarrow \mathsf{nnat} \Rightarrow \mathsf{nat}$, we introduce the rules:

$$\mathsf{F}\,F\,w \to \mathsf{F}'\,F\,(\mathsf{len}\,w)\,(\mathsf{q_0}\,(\mathsf{split}(\mathsf{L}\,[])\,(\mathsf{R}\,w))\,(\mathsf{split}(\mathsf{L}\,[])\,(\mathsf{R}\,[]))\,(\mathsf{split}(\mathsf{L}\,[])\,(\mathsf{R}\,[])))$$
$$\mathsf{F}'\,F\,z\,c \to \mathsf{execute}\,F\,\Theta^{P_M+1}_{F;z;\emptyset}\,\mathsf{o}\,z\,\emptyset\,c$$

$$\mathsf{execute}\,F\,(\mathsf{s}\,n)\,m\,z\,a\,(\mathsf{q}\,t_1\,t_2\,t_3) \to \mathsf{execute}\,F\,n\,(\mathsf{n}\,m)\,z\,(\mathsf{step}\,F\,(\mathsf{q}\,t_1\,t_2\,t_3)) \text{ for } \mathsf{q} \notin \{\mathtt{query}, \mathtt{end}\}$$
$$\mathsf{execute}\,F\,(\mathsf{s}\,n)\,m\,z\,a\,(\mathtt{query}\,t_1\,t_2\,t_3) \to \mathsf{execute}'\,F\,n\,(\mathsf{n}\,m)\,z\,(\mathsf{scons}\,(\mathsf{extract}\,t_2)\,a)\,(\mathtt{query}\,t_1\,t_2\,t_3)$$
$$\mathsf{execute}'\,F\,n\,m\,z\,a\,c \to \mathsf{execute}\,F\,(\mathsf{minus}\,\Theta^{P_M+1}_{F;z;a}\,m)\,m\,z\,a\,(\mathsf{step}\,F\,c)$$
$$\mathsf{execute}\,F\,n\,m\,z\,a\,(\mathtt{end}\,t_1\,t_2\,t_3) \to \mathsf{extract}\,t_1$$
$$\mathsf{extract}\,(\mathsf{split}\,(\mathsf{L}\,x)\,(\mathsf{R}\,y)) \to \mathsf{clean}\,y$$
$$\mathsf{minus}\,x\,\mathsf{o} \to x \qquad \mathsf{minus}\,0\,(\mathsf{n}\,y) \to \mathsf{o} \qquad \mathsf{minus}\,(\mathsf{s}\,x)\,(\mathsf{n}\,y) \to \mathsf{minus}\,x\,y$$

That is, an execution on $\mathsf{F}\,\mathsf{S}_f\,\underline{\mathsf{w}}$ first computes the length of w and $P_M(|f|_\emptyset, |w|)$, and uses these as arguments to $\mathsf{execute}$. Each normal transition lowers the number n of steps we are allowed to do and increases the number m of steps we have done. Each oracle transition updates A, and either lowers n by one, or updates it to the new value $P_M(|f|_A, |w|) - m$, since we have already done m steps. Once we read the final state, the answer is read off the first tape.

For the interpretation, note that the unusual size set of nnat allows us to choose $\mathcal{J}^{\mathsf{s}}_{\mathsf{minus}} = \boldsymbol{\lambda} xy. \max(x-y, 0)$ without losing monotonicity. Hence, in every step $\mathsf{execute}\,F\,n\,m\,z\,a\,c$, the value $\max(P_M(\llbracket F \rrbracket^{\mathsf{s}}, \llbracket z \rrbracket^{\mathsf{s}}) + 1 - \llbracket m \rrbracket^{\mathsf{s}}, \llbracket n \rrbracket^{\mathsf{s}})$ decreases by at least one. Since $\llbracket \Theta^{P_M+1} F; z; a \rrbracket^{\mathsf{s}} = P_M(\llbracket F \rrbracket^{\mathsf{s}}, \llbracket z \rrbracket^{\mathsf{s}})$ regardless of a, we can use this component as part of the interpretation. So let us provide the interpretations for the nnat functions symbols and the two simple rules for $\mathsf{extract}$ and minus, given as follows:

$$\begin{array}{rclcrcl}
\mathcal{J}^{\mathsf{s}}_{\mathsf{o}} & = & 0 & & \mathcal{J}^{\mathsf{c}}_{\mathsf{o}} & = & 0 \\
\mathcal{J}^{\mathsf{s}}_{\mathsf{n}} & = & \boldsymbol{\lambda} x.x+1 & & \mathcal{J}^{\mathsf{c}}_{\mathsf{n}} & = & \boldsymbol{\lambda} x.0 \\
\mathcal{J}^{\mathsf{s}}_{\mathsf{extract}} & = & \boldsymbol{\lambda} x.x & & \mathcal{J}^{\mathsf{c}}_{\mathsf{extract}} & = & \boldsymbol{\lambda} x.x+2 \\
\mathcal{J}^{\mathsf{s}}_{\mathsf{minus}} & = & \boldsymbol{\lambda} xy. \max(x-y, 0) & & & & \\
\mathcal{J}^{\mathsf{c}}_{\mathsf{minus}} & = & \boldsymbol{\lambda} xy.x & & & &
\end{array}$$

These functions are all monotonic, and their rules are oriented (as can easily be checked). By induction on the polynomial P, we can find polynomials A_P, B_P such that

$$\mathsf{cost}(\Theta^{P}_{F;z;a}) \leq \llbracket a \rrbracket^{\mathsf{s}} * A_P(F^c, F^s, \llbracket z \rrbracket^{\mathsf{s}}) + B_P(F^c, F^s, \llbracket z \rrbracket^{\mathsf{s}})$$

assuming F, z and a are in normal form.

To define our remaining interpretation functions, first let:

- $\theta_{F,z,n,m} := \max(P_M(F^s, z) + 1 - m, n)$
- $\mathtt{POLY}_{F,z}[x] := x * A_{P_M+1}(F^c, F^s, \llbracket z \rrbracket^{\mathsf{s}}) + B_{P_M+1}(F^c, F^s, \llbracket z \rrbracket^{\mathsf{s}})$, so the polynomial bounding $\mathsf{cost}(\Theta^{P_M+1}_{F;z;a})$ if $\llbracket a \rrbracket^{\mathsf{s}} = x$.

Then, we can orient the size interpretations of the rewrite rules by the following interpretation:

$$\begin{array}{rcl} \mathcal{J}^{\mathsf{s}}_{\mathsf{F}} & = & \boldsymbol{\lambda} F n.n + P_M(F, n) + 1 \\ \mathcal{J}^{\mathsf{s}}_{\mathsf{F}'} & = & \boldsymbol{\lambda} F z c.c + P_M(F, z) + 1 \\ \mathcal{J}^{\mathsf{s}}_{\mathsf{execute}} & = & \boldsymbol{\lambda} F n m z a c.c + \theta_{F,z,n,m} \\ \mathcal{J}^{\mathsf{s}}_{\mathsf{execute}'} & = & \boldsymbol{\lambda} F n m z a c.c + 1 + \theta_{F,z,n,m} \end{array}$$

And the cost interpretations by:

$$\begin{array}{rcl} \mathcal{J}^{\mathsf{c}}_{\mathsf{F}} & = & \boldsymbol{\lambda} F n.(P_M(F^s, n) + 1) * \\ & & \quad (8 + 3 * P_M(F^s, n) + 2 * n + F^c(P_M(F^s, n) + n + 1) + \\ & & \quad \mathtt{POLY}_{F,z}[P_M(F^s, n) + 1]) + 6 + 2 * n + \mathtt{POLY}_{F,z}[0] \\ \\ \mathcal{J}^{\mathsf{c}}_{\mathsf{F}'} & = & \boldsymbol{\lambda} F z c.(P_M(F^s, z) + 1) * \\ & & \quad (8 + 3 * P_M(F^s, z) + 2 * c + F^c(P_M(F^s, z) + 1 + c) + \\ & & \quad \mathtt{POLY}_{F,z}[P_M(F^s, z) + 1]) + 4 + c + \mathtt{POLY}_{F,z}[0] \\ \\ \mathcal{J}^{\mathsf{c}}_{\mathsf{execute}} & = & \boldsymbol{\lambda} F n m z a c.\theta_{F,z,n,m} * \\ & & \quad (5 + 2 * (\theta_{F,z,n,m} + c) + F^c(\theta_{F,z,n,m} + c) + \\ & & \quad \mathtt{POLY}_{F,z}[\theta_{F,z,n,m} + a] + P_M(F^s, z)) + 3 + \theta_{F,z,n,m} + c \\ \\ \mathcal{J}^{\mathsf{c}}_{\mathsf{execute}'} & = & \boldsymbol{\lambda} F n m z a c.(\theta_{F,z,n,m} + 1) * \\ & & \quad (5 + 2 * (\theta_{F,z,n,m} + c + 1) + F^c(\theta_{F,z,n,m} + c + 1) + \\ & & \quad \mathtt{POLY}_{F,z}[\theta_{F,z,n,m} + a] + P_M(F^s, z)) + 1 \end{array}$$

To see that these interpretations are correct, we first observe:

$$\begin{array}{rcl} \theta_{F,z,\mathsf{s}\,n,m} & = & \max(P_M(F^s, z) + 1 - m, n + 1) \\ & = & \max(P_M(F^s, z) + 1 - (m + 1), n) + 1 \\ & = & \theta_{F,z,n,m+1} + 1 \end{array}$$

Observe that this holds since $\max(a + 1, b + 1) = \max(a, b) + 1$. We also have the following:

$$\begin{array}{rcl} \theta_{F,z,n,m} & = & \max(P_M(F^s, z) + 1 - m, n) \\ & \geq & \max(P_M(F^s, z) + 1 - m, 0) \\ & = & \max(P_M(F^s, z) + 1 - m, \max(P_M(F^s, z) + 1 - m), 0) \\ & = & \theta_{F,z,[\![\mathsf{minus}\,\Theta^{P_M+1}_{F;z;m}]\!]^{\mathsf{s}},m} \end{array}$$

The inequalities now follow by writing out definitions.

Theorem 6.4. If $\Psi \in \mathtt{BFF}_2$, then there exists a finite orthogonal STRS $\mathcal{R}$ such that F computes Ψ in $\mathcal{R}$ and $\mathcal{R}$ admits a polynomially bounded interpretation.

7. Conclusion

This paper gives the first characterization of the class of type-2 basic feasible functionals by term rewriting based on the interpretation method and in particular on polynomial cost–size interpretations. Surely, it would make sense to try to go beyond types of order 2 and try to characterize classes of basic feasible functionals of order at least 3. Such classes are less well understood than their order-2 counterparts and perhaps an analysis based on tools from rewriting could help to shed some light on their nature (see, e.g., Hugo Férée's work on this subject [Fér17]).

References

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, USA, 1st edition, 2009.

[AL16] Beniamino Accattoli and Ugo Dal Lago. (leftmost-outermost) beta reduction is invariant, indeed. *Log. Methods Comput. Sci.*, 12(1), 2016. doi:10.2168/LMCS-12(1:4)2016.

[AM10] Martin Avanzini and Georg Moser. Closing the gap between runtime complexity and polytime computability. In Christopher Lynch, editor, *Proceedings of the 21st International Conference on Rewriting Techniques and Applications, RTA 2010, July 11-13, 2010, Edinburgh, Scottland, UK*, volume 6 of *LIPIcs*, pages 33–48. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2010. doi:10.4230/LIPICS.RTA.2010.33.

[AM13] Martin Avanzini and Georg Moser. Polynomial path orders. *Log. Methods Comput. Sci.*, 9(4), 2013. doi:10.2168/LMCS-9(4:9)2013.

[AMS16] Martin Avanzini, Georg Moser, and Michael Schaper. TcT: Tyrolean complexity tool. In Marsha Chechik and Jean-François Raskin, editors, *Proceedings of TACAS 2016 conference*, volume 9636 of *Lecture Notes in Computer Science*, pages 407–423. Springer, 2016. doi:10.1007/978-3-662-49674-9_24.

[Bai11] Patrick Baillot. Elementary linear logic revisited for polynomial time and an exponential time hierarchy. In Hongseok Yang, editor, *Programming Languages and Systems - 9th Asian Symposium, APLAS 2011, Kenting, Taiwan, December 5-7, 2011. Proceedings*, volume 7078 of *Lecture Notes in Computer Science*, pages 337–352. Springer, 2011. doi:10.1007/978-3-642-25318-8_25.

[BC92] Stephen J. Bellantoni and Stephen A. Cook. A new recursion-theoretic characterization of the polytime functions. *Comput. Complex.*, 2:97–110, 1992. doi:10.1007/BF01201998.

[BCE+98] Paul Beame, Stephen A. Cook, Jeff Edmonds, Russell Impagliazzo, and Toniann Pitassi. The relative complexity of NP search problems. *J. Comput. Syst. Sci.*, 57(1):3–19, 1998. doi:10.1006/JCSS.1998.1575.

[BCMT98] Guillaume Bonfante, Adam Cichon, Jean-Yves Marion, and Hélène Touzet. Complexity classes and rewrite systems with polynomial interpretation. In Georg Gottlob, Etienne Grandjean, and Katrin Seyr, editors, *Computer Science Logic, 12th International Workshop, CSL '98, Annual Conference of the EACSL, Brno, Czech Republic, August 24-28, 1998, Proceedings*, volume 1584 of *Lecture Notes in Computer Science*, pages 372–384. Springer, 1998. doi:10.1007/10703163_25.

[BCMT01] Guillaume Bonfante, Adam Cichon, Jean-Yves Marion, and Hélène Touzet. Algorithms with polynomial interpretation termination proof. *J. Funct. Program.*, 11(1):33–53, 2001. doi:10.1017/S0956796800003877.

[BDBRDR18] Patrick Baillot, Erika De Benedetti, and Simona Ronchi Della Rocca. Characterizing polynomial and exponential complexity classes in elementary lambda-calculus. *Inf. Comput.*, 261:55–77, 2018. doi:10.1016/J.IC.2018.05.005.

[BDL12] Patrick Baillot and Ugo Dal Lago. Higher-order interpretations and program complexity. In *Proceedings of CSL 2012*, volume 16 of *LIPIcs*, pages 62–76. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012. doi:10.4230/LIPICS.CSL.2012.62.

[BL16] Patrick Baillot and Ugo Dal Lago. Higher-order interpretations and program complexity. *Inf. Comput.*, 248:56–81, 2016. URL: https://doi.org/10.1016/j.ic.2015.12.008.

[BLKV24] Patrick Baillot, Ugo Dal Lago, Cynthia Kop, and Deivid Vale. On basic feasible functionals and the interpretation method. In Naoki Kobayashi and James Worrell, editors, *Foundations of Software Science and Computation Structures - 27th International Conference, FoSSaCS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part II*, volume 14575 of *Lecture Notes in Computer Science*, pages 70–91. Springer, 2024. doi:10.1007/978-3-031-57231-9_4.

[BLM12] Patrick Baillot, Ugo Dal Lago, and Jean-Yves Moyen. On quasi-interpretations, blind abstractions and implicit complexity. *Math. Struct. Comput. Sci.*, 22(4):549–580, 2012. doi:10.1017/S0960129511000685.

[BMM05] Guillaume Bonfante, Jean-Yves Marion, and Jean-Yves Moyen. Quasi-interpretations and small space bounds. In Jürgen Giesl, editor, *Term Rewriting and Applications, 16th International*

Conference, RTA 2005, Nara, Japan, April 19-21, 2005, Proceedings, volume 3467 of *Lecture Notes in Computer Science*, pages 150–164. Springer, 2005. doi:10.1007/978-3-540-32033-3_12.

[BMM11] Guillaume Bonfante, Jean-Yves Marion, and Jean-Yves Moyen. Quasi-interpretations a way to control resources. *Theor. Comput. Sci.*, 412(25):2776–2796, 2011. doi:10.1016/j.tcs.2011.02.007.

[BW96] Arnold Beckmann and Andreas Weiermann. A term rewriting characterization of the polytime functions and related complexity classes. *Arch. Math. Log.*, 36(1):11–30, 1996. doi:10.1007/s001530050054.

[CK89] Stephen A. Cook and Bruce M. Kapron. Characterizations of the basic feasible functionals of finite type (extended abstract). In *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989*, pages 154–159. IEEE Computer Society, 1989. doi:10.1109/SFCS.1989.63471.

[Cob65] Alan Cobham. The intrinsic computational difficulty of functions. In Yehoshua Bar-Hillel, editor, *Logic, Methodology and Philosophy of Science: Proceedings of the 1964 International Congress (Studies in Logic and the Foundations of Mathematics)*, pages 24–30. North-Holland Publishing, 1965.

[Con73] Robert L. Constable. Type two computational complexity. In Alfred V. Aho, Allan Borodin, Robert L. Constable, Robert W. Floyd, Michael A. Harrison, Richard M. Karp, and H. Raymond Strong, editors, *Proceedings of the 5th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1973, Austin, Texas, USA*, pages 108–121. ACM, 1973. doi:10.1145/800125.804041.

[Der82] Nachum Dershowitz. Orderings for term-rewriting systems. *Theor. Comput. Sci.*, 17:279–301, 1982. doi:10.1016/0304-3975(82)90026-3.

[DLH11] Ugo Dal Lago and Martin Hofmann. Realizability models and implicit complexity. *Theor. Comput. Sci.*, 412(20):2029–2047, 2011. doi:10.1016/J.TCS.2010.12.025.

[DLM12] Ugo Dal Lago and Simone Martini. On constructor rewrite systems and the lambda calculus. *Log. Methods Comput. Sci.*, 8(3), 2012. doi:10.2168/LMCS-8(3:12)2012.

[DR06] Norman Danner and James S. Royer. Adventures in time and space. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, pages 168–179. ACM, 2006. doi:10.1145/1111037.1111053.

[Fér17] Hugo Férée. Game semantics approach to higher-order complexity. *J. Comput. Syst. Sci.*, 87:1–15, 2017. doi:10.1016/J.JCSS.2017.02.003.

[FHHP15] Hugo Férée, Emmanuel Hainry, Mathieu Hoyrup, and Romain Péchoux. Characterizing polynomial time complexity of stream programs using interpretations. *Theor. Comput. Sci.*, 585:41–54, 2015. doi:10.1016/J.TCS.2015.03.008.

[Gir98] Jean-Yves Girard. Light linear logic. *Inf. Comput.*, 143(2):175–204, 1998. doi:10.1006/INCO.1998.2700.

[GTS04] Jürgen Giesl, René Thiemann, and Peter Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In Franz Baader and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 11th International Conference, LPAR 2004, Montevideo, Uruguay, March 14-18, 2005, Proceedings*, volume 3452 of *Lecture Notes in Computer Science*, pages 301–331. Springer, 2004. doi:10.1007/978-3-540-32275-7_21.

[HKMP20] Emmanuel Hainry, Bruce M. Kapron, Jean-Yves Marion, and Romain Péchoux. A tier-based typed programming language characterizing feasible functionals. In Holger Hermanns, Lijun Zhang, Naoki Kobayashi, and Dale Miller, editors, *LICS '20: 35th Annual ACM/IEEE Symposium on Logic in Computer Science, Saarbrücken, Germany, July 8-11, 2020*, pages 535–549. ACM, 2020. doi:10.1145/3373718.3394768.

[HKMP22] Emmanuel Hainry, Bruce M. Kapron, Jean-Yves Marion, and Romain Péchoux. Complete and tractable machine-independent characterizations of second-order polytime. In Patricia Bouyer and Lutz Schröder, editors, *Foundations of Software Science and Computation Structures - 25th International Conference, FOSSACS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022,* *Proceedings*, volume 13242 of *Lecture Notes in Computer Science*, pages 368–388. Springer, 2022. doi:10.1007/978-3-030-99253-8_19.

[Hof03] Martin Hofmann. Linear types and non-size-increasing polynomial time computation. *Inf. Comput.*, 183(1):57–85, 2003. doi:10.1016/S0890-5401(03)00009-9.

[HP20] Emmanuel Hainry and Romain Péchoux. Theory of higher order interpretations and application to basic feasible functions. *Log. Methods Comput. Sci.*, 16(4), 2020. URL: https://lmcs.episciences.org/6973.

[HS69] Juris Hartmanis and Richard Edwin Stearns. Automata-based computational complexity. *Inf. Sci.*, 1(2):173–184, 1969. doi:10.1016/0020-0255(69)90014-0.

[IRK01] Robert J. Irwin, James S. Royer, and Bruce M. Kapron. On characterizations of the basic feasible functionals (part i). *J. Funct. Program.*, 11(1):117–153, 2001. doi:10.1017/s0956796800003841.

[Jon01] Neil D. Jones. The expressive power of higher-order types or, life without CONS. *J. Funct. Program.*, 11(1):5–94, 2001. URL: https://doi.org/10.1017/s0956796800003889.

[KC91] Bruce M. Kapron and Stephen A. Cook. A new characterization of Mehlhorn's polynomial time functionals (extended abstract). In *32nd Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, 1-4 October 1991*, pages 342–347. IEEE Computer Society, 1991. doi:10.1109/SFCS.1991.185389.

[KC96] Bruce M. Kapron and Stephen A. Cook. A new characterization of type-2 feasibility. *SIAM J. Comput.*, 25(1):117–132, 1996. doi:10.1137/S0097539794263452.

[KC12] Akitoshi Kawamura and Stephen A. Cook. Complexity theory for operators in analysis. *ACM Trans. Comput. Theory*, 4(2):5:1–5:24, 2012. doi:10.1145/2189778.2189780.

[KS17] Cynthia Kop and Jakob Grue Simonsen. Complexity hierarchies and higher-order cons-free term rewriting. *Log. Methods Comput. Sci.*, 13(3), 2017. doi:10.23638/LMCS-13(3:8)2017.

[KS18] Bruce M. Kapron and Florian Steinberg. Type-two polynomial-time and restricted lookahead. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 579–588. ACM, 2018. doi:10.1145/3209108.3209124.

[Kus01] Keiichirou Kusakari. On proving termination of term rewriting systems with higher-order variables. *IPSJ Transactions on Programming*, 42(SIG 7 (PRO 11)):35–45, 2001. URL: http://id.nii.ac.jp/1001/00016864/.

[KV21] Cynthia Kop and Deivid Vale. Tuple interpretations for higher-order complexity. In Naoki Kobayashi, editor, *6th International Conference on Formal Structures for Computation and Deduction, FSCD 2021, July 17-24, 2021, Buenos Aires, Argentina (Virtual Conference)*, volume 195 of *LIPIcs*, pages 31:1–31:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.FSCD.2021.31.

[KV23] C. Kop and D. Vale. Cost-size semantics for call-by-value higher-order rewriting. In *Proc. FSCD*, volume 260 of *LIPIcs*, pages 15:1–15:19, 2023. doi:10.4230/LIPIcs.FSCD.2023.15.

[KvOvR93] 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. doi:10.1016/0304-3975(93)90091-7.

[Laf04] Yves Lafont. Soft linear logic and polynomial time. *Theor. Comput. Sci.*, 318(1-2):163–180, 2004. doi:10.1016/J.TCS.2003.10.018.

[Lan79] Dallas S Lankford. On proving term rewriting systems are noetherian. *Memo MTP-3*, 1979. URL: https://www.ens-lyon.fr/LIP/REWRITING/TERMINATION/Lankford_Poly_Term.pdf.

[Lei91] Daniel Leivant. A foundational delineation of computational feasiblity. In *Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July 15-18, 1991*, pages 2–11. IEEE Computer Society, 1991. doi:10.1109/LICS.1991.151625.

[LM94] Daniel Leivant and Jean-Yves Marion. Ramified recurrence and computational complexity II: substitution and poly-space. In Leszek Pacholski and Jerzy Tiuryn, editors, *Computer Science Logic, 8th International Workshop, CSL '94, Kazimierz, Poland, September 25-30, 1994, Selected Papers*, volume 933 of *Lecture Notes in Computer Science*, pages 486–500. Springer, 1994. doi:10.1007/BFB0022277.

[LS16] Ugo Dal Lago and Ulrich Schöpp. Computation by interaction for space-bounded functional programming. *Inf. Comput.*, 248:150–194, 2016. doi:10.1016/J.IC.2015.04.006.

[Meh76] Kurt Mehlhorn. Polynomial and abstract subrecursive classes. *J. Comput. Syst. Sci.*, 12(2):147–178, 1976. doi:10.1016/S0022-0000(76)80035-9.

[MM00] Jean-Yves Marion and Jean-Yves Moyen. Efficient first order functional program interpreter with time bound certifications. In Michel Parigot and Andrei Voronkov, editors, *Logic for Programming and Automated Reasoning, 7th International Conference, LPAR 2000, Reunion Island, France, November 11-12, 2000, Proceedings*, volume 1955 of *Lecture Notes in Computer Science*, pages 25–42. Springer, 2000. doi:10.1007/3-540-44404-1_3.

[MN70] Zohar Manna and Steven Ness. On the termination of Markov algorithms. In *Proceedings of the Third Hawaii International Conference on System Science*, pages 789–792, 1970.

[MN98] Richard Mayr and Tobias Nipkow. Higher-order rewrite systems and their confluence. *Theor. Comput. Sci.*, 192(1):3–29, 1998. doi:10.1016/S0304-3975(97)00143-6.

[Oit01] Isabel Oitavem. Implicit characterizations of pspace. In Reinhard Kahle, Peter Schroeder-Heister, and Robert F. Stärk, editors, *Proof Theory in Computer Science, International Seminar, PTCS 2001, Dagstuhl Castle, Germany, October 7-12, 2001, Proceedings*, volume 2183 of *Lecture Notes in Computer Science*, pages 170–190. Springer, 2001. doi:10.1007/3-540-45504-3_11.

[Ter03] Terese. *Term rewriting systems*, volume 55 of *Cambridge tracts in theoretical computer science*. Cambridge University Press, 2003.

[Val24] Deivid Vale. *On Semantical Methods for Higher-Order Complexity Analysis*. PhD thesis, Radboud University Nijmegen, 2024. URL: https://hdl.handle.net/2066/304473.

[vdP96] Jaco van de Pol. *Termination of Higher-order Rewrite Systems*. PhD thesis, University of Utrecht, 1996. URL: https://www.cs.au.dk/~jaco/papers/thesis.pdf.