Cost–Size Semantics for Call-by-Value Higher-Order Rewriting

Cynthia Kop
Institute for Computing and Information Sciences, Radboud University, Nijmegen, The Netherlands

Deivid Vale
Institute for Computing and Information Sciences, Radboud University, Nijmegen, The Netherlands

Abstract

Higher-order rewriting is a framework in which higher-order programs can be described by transformation rules on expressions. A computation occurs by transforming an expression into another using such rules. This step-by-step computation model induced by rewriting naturally gives rise to a notion of complexity as the number of steps needed to reduce expressions to a normal form, i.e., an expression that cannot be reduced further. The study of *complexity analysis* focuses on the development of automatable techniques to provide bounds to this number. In this paper, we consider a form of higher-order rewriting with a call-by-value evaluation strategy, so as to model call-by-value programs. We provide a cost–size semantics: a class of algebraic interpretations to map terms to tuples which bound both the reduction cost and the size of normal forms.

2012 ACM Subject Classification Theory of computation → Equational logic and rewriting

Keywords and phrases Call-by-Value Evaluation, Complexity Theory, Higher-Order Rewriting

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

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

1 Introduction

Term rewriting is a logical framework that, among other applications, provides a computational model to specify algorithms. Simple programs (especially functional programs) can typically be modeled as a term rewriting system where a program state is expressed as a term and evaluation is modeled by rewriting expressions using reduction rules. *Higher-order* term rewriting in particular provides a natural model for functional programming languages. Due to the abstract nature of rewriting, it is feasible to forgo specific language details and still derive useful term rewriting results that may carry over to program analysis [3, 10, 15, 24].

In this paper, we study *complexity*, which in the context of term rewriting is typically understood as the number of steps needed to reach a normal from when starting in terms of a certain shape and size. A natural way to determine these bounds is adapting termination proof techniques to deduce the complexity. There is a myriad of works following this idea. To mention a few, see [4, 7, 9, 18, 19, 25] for interpretation methods, [8, 17, 31] for lexicographic and path orders, and [16, 28] for dependency pairs.

However, those ideas are focused on *first-order* term rewriting. There is very little work on complexity of *higher-order* term rewriting. While there is a lot of work on complexity of functional programs [2, 13, 20, 26], this work uses quite different ideas from the methods developed for term rewriting. It would be beneficial to combine these ideas.

In a previous work [21], we introduced an extension of the method of *weakly monotonic algebras* [14, 29] to *tuple interpretations*. The idea of algebras is to choose an interpretation domain A, and interpret terms s as elements $[\![s]\!]$ of A compositionally, in such a way that whenever $s \to t$ we have $[\![s]\!] > [\![t]\!]$. Hence, a rewriting step on terms implies a strict decrease

© Cynthia Kop and Deivid Vale;
licensed under Creative Commons License CC-BY 4.0

on A. The defining characteristic of tuple interpretations is to split the complexity measure into abstract notions of cost and size. This coincides with ideas often used in resource analysis of functional programs [2, 13]. This is a popular idea, as a very similar approach was introduced for first-order rewriting around the same time [33].

This previous work considered full higher-order rewriting, so without an evaluation strategy. However, this is not a very realistic setting, especially with the goal of eventually extending the methodology to various functional programming languages. In practice, program evaluation is deterministic, i.e., it follows a specific strategy such as call-by-value evaluation. Reduction below a λ-binder is also not usually allowed. The difference can be substantial: for instance for a pair of rules $\mathsf{f}\ x\ \mathsf{0} \to x$, $\mathsf{f}\ x\ (\mathsf{s}\ y) \to \mathsf{f}\ (\mathsf{pair}\ x\ x)\ y$, if x is instantiated by a term that is not in normal form, the complexity is linear if we evaluate call-by-value, and exponential with an arbitrary evaluation strategy. Also in complexity analysis of first-order term rewriting, considering innermost evaluation is common [27, 28].

In this paper, our goal is to extend the work of [21] to *weak call-by-value* reduction. To our knowledge, this is the first complexity method for higher-order term rewriting with an evaluation strategy. While the restriction of the strategy leads to tighter complexity bounds, the definitions needed to obtain these bounds are much more intricate, largely due to the potential for rules and β-redexes of higher type. We believe that this will bring the method of weakly monotonic algebras closer to the reality of functional program analysis.

Tuple interpretations do not provide a complete termination proof method: there are terminating systems for which interpretations cannot be found. Consequently, it does not induce a complete complexity analysis framework either. Notwithstanding, it has the potential to be very powerful if we choose the cost–size sets wisely. A second limitation is that the question whether a suitable interpretation exists is undecidable in general, which is expected already in the polynomial case [23]. Undecidability never hindered computer scientists' efforts on mechanizing difficult problems, however. Indeed, several proof search methods have been developed over the years to find interpretations automatically [6, 11, 12, 18, 33].

Contribution This paper will introduce tuple interpretations for higher-order term rewriting systems using a weak call-by-value evaluation strategy, and use them to define both a termination method under this strategy, and a new definition of weak call-by-value runtime complexity along with a methodology to derive bounds for it.

This paper builds on the ideas of [21], which introduces tuple interpretations and a notion of runtime complexity for full higher-order rewriting (without evaluation strategy). The key difference here is our focus on a weak call-by-value evaluation strategy. This allows for tighter bounds, but also requires significant technical changes. since the "cost" for a term of higher type can no longer be captured by just a function (as we will explain in Section 3).

An additional change compared to [21] is that we have separated the cost and size components into distinct functions. In [21], it is in theory allowed for the size component to depend on the cost component, even though in practice this never happened. By fully separating the components, it is easier to prove correctness of a given tuple interpretation.

Paper Overview In Section 2 we review basic notation on higher-order rewriting and define our notion of call-by-value strategy. In Section 3 we give an informal overview of how the technique works. These ideas are formalized in Sections 4 and 5 where we respectively provide a formal cost–size semantics for simple types, and interpret terms as cost–size tuples as well as proving some basic properties of them. We provide additional examples in Section 6. In Section 7 we conclude the paper and discuss future work.

2 Preliminaries

Unlike first-order rewriting, there is no single consensus formalism for higher-order rewriting, but rather a variety of sometimes incompatible formats. The formalism we consider here is a style of simply typed lambda calculus extended with function symbols and rules. The matching mechanism is modulo alpha, and beta reduction is included in the rewriting relation. This is essentially the formalism used in the higher-order category of the international termination competition [32], but slightly simplified for easier representation.[1]

Types, Terms, and Equality Let $\mathbb{B}$ be a nonempty set whose elements are called *base types* and range over ι, κ, ν. The set $\mathbb{T}_{\mathbb{B}}$ of *simple types* over $\mathbb{B}$ is generated by the grammar: $\mathbb{T}_{\mathbb{B}} :- \mathbb{B} \mid \mathbb{T}_{\mathbb{B}} \Rightarrow \mathbb{T}_{\mathbb{B}}$. Types from $\mathbb{T}_{\mathbb{B}}$ range over σ, τ, ρ. The $\Rightarrow$ type constructor is right-associative, so we write $\sigma \Rightarrow \tau \Rightarrow \rho$ for $(\sigma \Rightarrow (\tau \Rightarrow \rho))$. Notice that every simple type σ can be written as $\tau_1 \Rightarrow \cdots \Rightarrow \tau_n \Rightarrow \iota$. We informally say that the τ_i's are the *input types* and the base type ι is the *output type*. We abbreviate such types by $\boldsymbol{\tau} \Rightarrow \iota$. The *type order* of a type is the number: (a) $\texttt{ord}(\iota) = 0$ and (b) $\texttt{ord}(\sigma \Rightarrow \tau) = \max(1 + \texttt{ord}(\sigma), \texttt{ord}(\tau))$. A *signature* $\mathbb{F}$ is a triple $(\mathbb{B}, \Sigma, \texttt{ar})$ where $\mathbb{B}$ is a set of base types, Σ is a nonempty finite set of symbols, and $\texttt{ar}$ is a function $\texttt{ar} : \Sigma \longrightarrow \mathbb{T}_{\mathbb{B}}$. For each type σ, we postulate the existence of a nonempty set $\mathbb{X}_\sigma$ of countably many variables. Furthermore, we impose that $\mathbb{X}_\sigma \cap \mathbb{X}_\tau = \emptyset$ whenever $\sigma \neq \tau$. Let $\mathbb{X}$ denote the family of sets $(\mathbb{X}_\sigma)_{\sigma \in \mathbb{T}_{\mathbb{B}}}$ indexed by $\mathbb{T}_{\mathbb{B}}$ 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 the judgment $s : \sigma$ can be deduced using the following rules:

$$\frac{x \in \mathbb{X}_\sigma}{x : \sigma} \qquad \frac{\mathsf{f} \in \Sigma \qquad \texttt{ar}(\mathsf{f}) = \sigma}{\mathsf{f} : \sigma} \qquad \frac{s : \sigma \Rightarrow \tau \qquad t : \sigma}{(s\,t) : \tau} \qquad \frac{x \in \mathbb{X}_\sigma \qquad s : \tau}{(\lambda x.\, s) : \sigma \Rightarrow \tau}$$

Application of terms is left-associative, so we write $s\,t\,u$ for $((s\,t)\,u)$. Abstraction is right-associative, so we write $\lambda xyz.\, s$ for $\lambda x.\,(\lambda y.\,(\lambda z.\, s))$. Application takes precedence over abstraction, which allows us to write $\lambda x.\, s\,t$ for $\lambda x.\,(s\,t)$. Unnecessary parentheses are removed, and we write terms following these rules. The set $\texttt{fv}(s)$ of *free variables* occurring in s is defined as expected. A term s is *closed* if $\texttt{fv}(s) = \emptyset$. It is *ground* if no variable occurs in it. A symbol $\mathsf{f} \in \Sigma$ is called the *head symbol* of s if $s = \mathsf{f}\, s_1 \ldots s_k$. A *subterm* of s is a term t (we write $s \trianglerighteq t$) such that (i) $s = t$; or (ii) t is a subterm of s' or s'', if $s = s'\, s''$; or t is a subterm of s', if $s = \lambda x.\, s'$. A *proper subterm* of s is a subterm of s which is not equal to s.

A *substitution* γ is a type-preserving map from variables to terms such that the set $\texttt{dom}(\gamma) = \{x \in \mathbb{X} \mid \gamma(x) \neq x\}$ is finite. We may explicitly represent γ as a list of mappings $[x_1 := s_1, \ldots, x_k := s_k]$. The *capture avoiding* application of γ to s is defined as follows:

$$x\gamma = \gamma(x) \qquad\qquad (s\,t)\gamma = (s\gamma)\,(t\gamma)$$
$$\mathsf{f}\gamma = \mathsf{f} \qquad\qquad (\lambda x.\, s)\gamma = \lambda y.\,(s^{\{x \mapsto y\}}\gamma),\ \text{for } y \text{ fresh}$$

Here, $s^{\{x \mapsto y\}}$ denotes the term obtained by replacing every free occurrence of x by y in s. The result of $s\gamma$ is unique modulo α-renaming. We identify terms modulo α-equality, so $s = t$ denotes $s =_\alpha t$.

[1] The format in the competition allows both function application and application as separate notions, admitting the formation of terms such as $\mathsf{f}(s) \cdot t$. We here omit the functional notation, which is not necessary since any term can be represented in a curried form. Beyond this, the formalism is the same, including the permissiveness that left-hand sides do not need to be patterns or even in β-normal form.

Higher-Order Rewriting A *rewrite rule* $\ell \to r$ is a pair of terms of the same type such that $\ell = \mathsf{f}\,\ell_1 \dots \ell_k$ and $\mathtt{fv}(r) \subseteq \mathtt{fv}(\ell)$. A *term rewriting system* (TRS)[2] $\mathbb{R}$ is a set of rules. A relation $\to$ on terms is *monotonic* if $s \to s'$ implies $t\,s \to t\,s'$, $s\,u \to s'\,u$, and $\lambda x.\,s \to \lambda x.\,s'$; for all terms t and u of appropriate types. The *rewrite relation* $\to_{\mathbb{R}}$ induced by $\mathbb{R}$ is the smallest monotonic relation containing $\mathbb{R}$ union the β rule-scheme (i.e., $(\lambda x.\,s)\,t \to_\beta s[x := t]$) and closed under application of substitution. An $\mathbb{R}$-*reducible expression* (redex) is a term of form $\ell\gamma$ for some rule $\ell \to r$ and substitution γ. A β-redex is of the form $(\lambda x.\,s)\,t$.

Every rewrite rule $\ell \to r$ *defines* a symbol f, namely, the head symbol of ℓ. For each $\mathsf{f} \in \Sigma$, let $\mathbb{R}_\mathsf{f}$ denote the set of rewrite rules that define f in $\mathbb{R}$. A symbol $\mathsf{f} \in \Sigma$ is a *defined symbol* if $\mathbb{R}_\mathsf{f} \neq \emptyset$. A *constructor symbol* is a symbol $\mathsf{c} \in \Sigma$ such that $\mathbb{R}_\mathsf{f} = \emptyset$. We let Σ^{def} be the set of defined symbols and Σ^{con} the set of constructor symbols. Hence, $\Sigma = \Sigma^{\mathsf{def}} \uplus \Sigma^{\mathsf{con}}$. A *ground constructor term* is a term $\mathsf{c}\,s_1 \dots s_n$ with $n \geq 0$, where each s_i is a ground constructor term.

▶ **Example 1.** In this example we collect some common higher-order functions encoded as rules: map applies a function to each element of a list; comp composes two functions, app is the application functional, and rec encodes primitive recursion. Their monomorphic signature is defined as expected with functional arguments of type $\mathsf{nat} \Rightarrow \mathsf{nat}$ and lists having type list.

$$\mathsf{map}\,F\,\mathsf{nil} \to \mathsf{nil} \qquad \mathsf{comp}\,F\,G \to \lambda x.\,F\,(G\,x)$$
$$\mathsf{map}\,F\,(\mathsf{cons}\,x\,xs) \to \mathsf{cons}\,(F\,x)\,(\mathsf{map}\,F\,xs) \qquad \mathsf{app}\,F \to \lambda x.\,F\,x$$
$$\mathsf{rec}\,\mathsf{0}\,y\,F \to y \qquad \mathsf{rec}\,(\mathsf{s}\,x)\,y\,F \to F\,x\,(\mathsf{rec}\,x\,yF)$$

▶ **Example 2.** Some first-order functions over natural numbers:

$$\mathsf{dbl}\,\mathsf{0} \to \mathsf{0} \qquad \mathsf{add}\,x\,\mathsf{0} \to \mathsf{0} \qquad \mathsf{mult}\,x\,\mathsf{0} \to \mathsf{0}$$
$$\mathsf{dbl}\,(\mathsf{s}\,x) \to \mathsf{s}(\mathsf{s}\,(\mathsf{dbl}\,x)) \qquad \mathsf{add}\,x\,(\mathsf{s}\,y) \to \mathsf{s}\,(\mathsf{add}\,x\,y) \qquad \mathsf{mult}\,x\,(\mathsf{s}\,y) \to \mathsf{add}\,x\,(\mathsf{mult}\,x\,y)$$

Call-by-Value Higher-order Rewriting In this paper, we are interested in a restricted evaluation strategy, which limits reduction to terms whose immediate subterms are *values*:

▶ **Definition 3.** A term s is a *value* whenever s is:

- of the form $\mathsf{f}\,v_1 \dots v_n$, with each v_i a value and there is no rule $\mathsf{f}\,\ell_1 \dots \ell_k \to r$ with $k \leq n$;
- an abstraction, i.e., $s = \lambda x.\,t$.

Notice that by definition ground constructor terms are values, since there is no rule $\mathsf{c}\,\ell_1\,\dots\,\ell_k \to r$ for any k if $\mathsf{c} \in \Sigma^{\mathsf{con}}$. More complex values include partially applied functions and lambda-terms; for example, $\mathsf{add}\,\mathsf{0}$ or a list of functions $[\mathsf{add}\,\mathsf{0}; \lambda x.x; \mathsf{mult}\,\mathsf{0}; \mathsf{dbl}]$. In the weak call-by-value reduction strategy defined below, we shall not reduce under abstractions.

▶ **Definition 4.** The **higher-order weak call-by-value rewrite relation** $\to_v$ induced by $\mathbb{R}$ is defined as follows:

- $\mathsf{f}\,(\ell_1\gamma) \dots (\ell_k\gamma) \to_v r\gamma$, if $\mathsf{f}\,\ell_1 \dots \ell_k \to r \in \mathbb{R}$ and each $\ell_i\gamma$ is a value;
- $(\lambda x.\,s)\,v \to_v s[x := v]$, if v is a value;
- $s\,t \to_v s'\,t$ if $s \to_v s'$; and $s\,t \to_v s\,t'$ if $t \to_v t'$.

[2] Note that we use the acronym TRS for the style of higher-order term rewriting systems introduced in this section; *not* for a limitation to first-order term rewriting systems as is sometimes done in the literature. In this paper, we will not consider first-order TRSs as a special case at all.

Notice that when instantiating rules we use *value substitutions*, that is, their image for any nontrivial variable is always a value. All reductions in this paper are weak call-by-value. So we drop the v from the arrow, and $s \to t$ should be read as $s \to_v t$. We use explicit notation whenever confusion may arise.

We say that a term s is in *normal form* if there is no term t such that $s \to_v t$. A term s *has* a normal form t if $s \to_v^* t$ and t is in normal form. A TRS $\mathbb{R}$ is *terminating* if no infinite rewrite sequence $s \to_v s_1 \to_v \dots$ exists.

Ordered Sets and Monotonic Functions A *quasi-ordered set* $(A, \sqsupseteq)$ consists of a nonempty set A and a quasi-order (reflexive and transitive) $\sqsupseteq$ on A. An *extended well-founded set* $(A, >, \gtrsim)$ is a nonempty set A together with a well-founded order $>$ and a quasi-order $\gtrsim$ on A such that $\gtrsim$ is compatible with $>$, i.e., $x > y$ implies $x \gtrsim y$ and $x > y \gtrsim z$ implies $x > z$. Below we refer to an extended well-founded set simply as *well-founded set*. The `unit` set is the quasi-ordered set $(\{\mathtt{u}\}, \sqsupseteq)$, with $\mathtt{u} \sqsupseteq \mathtt{u}$.

Given quasi-ordered sets $(A, \sqsupseteq)$ and $(B, \sqsupseteq)$, a function $f : A \longrightarrow B$ is *weakly monotonic* if $x \sqsupseteq y$ implies $f(x) \sqsupseteq f(y)$. Let $A \Longrightarrow B$ denote the set of weakly monotonic functions from A to B. The comparison operator $\sqsupseteq$ on B induces point-wise comparison on $A \Longrightarrow B$ as follows: $f \sqsupseteq g$ if $f(x) \sqsupseteq g(x)$ for all $x \in A$. This way $(A \Longrightarrow B, \sqsupseteq)$ is also quasi-ordered. Given well-founded sets $(A, >, \gtrsim)$ and $(B, >, \gtrsim)$, a function $f : A \longrightarrow B$ is said to be *strongly monotonic* if $x > y$ implies $f(x) > f(y)$ and $x \gtrsim y$ implies $f(x) \gtrsim f(y)$.

3 Cost–Size Overview

In this section we sketch the broad idea of the methodology, focusing on intuition.

To start, every term is associated with a *size*. For a closed term of base type, this size could for instance be the number of symbols in its normal form; or a pair of integers, or a set of terms (e.g., the set of all normal forms of the term). We only require that each base type is associated with a *quasi-ordered set* with a minimum element. For a term of higher type, the size is a *weakly monotonic function*, which provides a bound for applications of the term.

▶ **Example 5.** In the signature of Examples 1 and 2, we may let $\mathcal{S}ize(\mathsf{0}) = 1$ and $\mathcal{S}ize(\mathsf{s}\ t) = 1 + \mathcal{S}ize(t)$; intuitively, the size of a ground constructor term of type `nat` is the number of function symbols in it. For lists, we could let $\mathcal{S}ize(\mathsf{nil}) = (0, 0)$ and $\mathcal{S}ize(\mathsf{cons}\ s\ t) = (\mathcal{S}ize(t)_1 + 1,\ \max(\mathcal{S}ize(s), \mathcal{S}ize(t)_2))$; intuitively, the size of a list of numbers is the pair (list length, size of its greatest element). We could let $\mathcal{S}ize(\mathsf{add}\ s)$ be the function that maps n to $\mathcal{S}ize(s) + n$, and $\mathcal{S}ize(\mathsf{map})$ the function that takes a (weakly monotonic) function F and a pair (l, m), and returns $(l, F(m))$; intuitively, if F bounds the size of the first argument, and we are given a list with maximum element of size m and length l, then applying `map` to these arguments yields a list which has length l, and elements have sizes bounded by $F(m)$.

Aside from a size, we need to calculate a *cost* for each term to associate a bound on the number of steps that can be taken from a given starting term. Aside from associating a natural number bounding this cost to each term, terms of higher type have computational content even in normal form; hence, we should associate a *cost function* to such terms: a weakly monotonic function that indicates the cost of applying this term to a value.

▶ **Example 6** (First idea for costs)**.** Intuitively, the number of steps to evaluate `add` $s\ t$ is bounded by the cost of evaluating the arguments, plus $\mathcal{S}ize(s)$ (as we easily see by inspecting the rules defining `add`). Hence, we would let $\mathcal{C}ost(\mathsf{add}\ s\ t) = \mathcal{C}ost(s) + \mathcal{C}ost(t) + \mathcal{S}ize(s)$, and could define $\mathcal{C}ost(\mathsf{add}) = \boldsymbol{\lambda}(c_1, s_1), (c_2, s_2).\ c_1 + c_2 + s_1$. Note that the cost function takes a *pair* of values for each argument: respectively, the cost and size of the argument.

For map, the number of steps to evaluate map $s\ t$ depends heavily on s, even if both s and t are values: map $(\lambda x.\mathsf{add}\ x\ 0)\ t$ will take substantially fewer steps than evaluating map $(\lambda x.\mathsf{mult}\ x\ x)\ t$. Hence, we should take the cost function for s into account as well as its size. This yields $\mathcal{C}ost(\mathsf{map}) = \boldsymbol{\lambda}(F_{cost}, F_{size}), (q_{cost}, (l, m)).\ q_{cost} + l + 1 + l * F_{cost}(0, m)$: the number of steps to evaluate map $s\ t$ is bounded by the cost of evaluating t first, then applying s $\langle$*length of list*$\rangle$ times to the largest element of t, plus the $1 + \langle$*length of list*$\rangle$ steps for the evaluation of map itself. Note that since we use a call-by-value strategy, the list q is evaluated to a value *before* the map rule fires, which is why F_{cost} is given a zero argument.

The cost for constructor applications $\mathsf{c}\ s_1 \cdots s_m$ is always just $\mathcal{C}ost(s_1) + \cdots + \mathcal{C}ost(s_m)$, since applying a constructor to terms does not lead to a further computation being done.

Examples 5 and 6 sketch an idea where $\mathcal{S}ize(s\ t) = \mathcal{S}ize(s)(\mathcal{S}ize(t))$ and $\mathcal{C}ost(s\ t) = \mathcal{C}ost(s)(\mathcal{C}ost(t), \mathcal{S}ize(t))$. Unfortunately, while this idea works well for *sizes*, it has some issues for *costs*; most importantly, that the computational content of terms of higher types is ignored. Although a term $\lambda x.s$ cannot be reduced, a term such as add (dbl 0) can be, and the cost for the dbl 0 reduction should be included. Moreover, terms of higher type can also reduce directly even when their subterms are values; e.g., comp $s\ t$ or $(\lambda x.s)\ t$ of type $\mathsf{nat} \Rightarrow \mathsf{nat}$.

Hence, we will instead consider a *pair* of costs: each term has a *cost number* (a bound on the number of steps to reduce this term to normal form), and a *cost function* (which bounds the cost of applying this normal form to a value, or is unit for base-type terms).

Unfortunately, this choice necessarily imposes a more complicated definition, since a pair cannot be applied like a function can; e.g., if the cost of s is $(12, \boldsymbol{\lambda}(x_{cost}, x_{size}).\ x_{cost} + x_{size})$, then when computing the cost for $s\ t$, we cannot just apply the function and forget the 12. Hence, we will define (formally in Definition 16) an alternative interpretation of application, so that, for $s : \sigma \Rightarrow \tau$ and $t : \sigma$, $\mathcal{C}ost(s\ t) = (\ \mathcal{C}ost\mathcal{N}um(s) + \mathcal{C}ost\mathcal{N}um(t) + c,\ \text{fun}\ )$, where $\mathcal{C}ost\mathcal{F}un(s)(\mathcal{C}ost\mathcal{F}un(t), \mathcal{S}ize(t))$ is the pair (c, fun).

▶ **Example 7** (Cost pairs). We let $\mathcal{C}ost(\mathsf{add}) = (\ 0,\ \boldsymbol{\lambda}(u_1, n).\ (\ 0,\ \boldsymbol{\lambda}(u_2, m).\ n\ )\ )$: the first 0 is the "cost number" for add, which is 0 because add is in normal form; and the function $\boldsymbol{\lambda}(u_1, n).\ (\ 0,\ \boldsymbol{\lambda}(u_2, m).\ n\ )$ takes a unit element and the size of a value, and returns a new pair. With the rough definition of application above, we have $\mathcal{C}ost(\mathsf{add}\ s) = (\ \mathcal{C}ost\mathcal{N}um(s),\ \boldsymbol{\lambda}(u, n).\ (\ \mathcal{S}ize(s),\ \mathtt{u}\ )\ )$. This matches the intuition that the number of steps needed to reduce add s to normal form is just the number of steps needed to reduce s, and the result is a value of function type which, if applied to a value with size n, can be normalized in $\mathcal{S}ize(s)$ steps. We obtain $\mathcal{C}ost(\mathsf{add}\ s\ t) = \mathcal{C}ost(s) + \mathcal{C}ost(t) + \mathcal{S}ize(s)$ as expected.

The notation is rather cumbersome but is needed for the formal definition. In practice, we can identify $\mathtt{unit} \times A$ and $A \times \mathtt{unit}$ with A for any set, and use $(x_1, \dots, x_n) \mapsto \varphi$ as shorthand for $(\ 0,\ \boldsymbol{\lambda}x_1.\ (\ 0,\ \boldsymbol{\lambda}x_2.\dots\ \varphi\ )\ )$. Then we can use the more palatable notation $\mathcal{C}ost(\mathsf{add}) = (n, m) \mapsto n$, or $\mathcal{C}ost(\mathsf{comp}) = ((F_{cost}, F_{size}), (G_{cost}, G_{size})) \mapsto (\ 2, \boldsymbol{\lambda}x_{size}.G_{cost}(x_{size}) + F_{cost}(G_{size}(x_{size}))\ )$ for the symbol comp which admits a rule of higher type $\mathsf{nat} \Rightarrow \mathsf{nat}$.

With these definitions, if we can show that $(\mathcal{C}ost(\ell), \mathcal{S}ize(\ell)) \succ (\mathcal{C}ost(r), \mathcal{S}ize(r))$ for all *value instances* of rules, then $\mathcal{C}ost\mathcal{N}um(s)$ defines a bound on the number of steps that can be taken to reduce s to normal form. We can use this to define bounds on the runtime complexity of the rewriting system – that is, on the number of steps that can be done when starting in certain kinds of terms of a given size (as we will discuss in Section 6).

▶ **Example 8.** We choose $\mathcal{Size}(\mathsf{nil})$, $\mathcal{Size}(\mathsf{cons})$ and $\mathcal{Size}(\mathsf{map})$ following Example 5, and let $\mathcal{Cost}(\mathsf{nil}) = 0$, $\mathcal{Cost}(\mathsf{cons}) = (n, m) \mapsto 0$ and $\mathcal{Cost}(\mathsf{map}) = ((F_{cost}, F_{size}), (l, m)) \mapsto l * F_{cost}(m) + l + 1$. Then, for a list $\mathsf{cons}\ h\ t$ with $\mathcal{Size}(t) = (l, m)$, we have

$$\begin{aligned}
\mathcal{Size}(\mathsf{map}\ F\ (\mathsf{cons}\ h\ t)) &= (l + 1, \mathcal{Size}(F)(\max(\mathcal{Size}(h)), m)) \\
&= (l + 1, \max(\mathcal{Size}(F)(\mathcal{Size}(h)), \mathcal{Size}(F)(m))) \\
&= \mathcal{Size}(\mathsf{cons}\ (F\ h)\ (\mathsf{map}\ F\ t))
\end{aligned}$$

by weak monotonicity of $\mathcal{Size}(F)$. Taking into account that if F, h and t are values, then they all have a cost number of 0, we also have:

$$\begin{aligned}
\mathcal{Cost}(\mathsf{map}\ F\ (\mathsf{cons}\ h\ t)) &= (l + 1) * \mathcal{CostFun}(F)(\max(\mathcal{Size}(h), m)) + l + 2 \\
&> \mathcal{CostFun}(F)(\mathcal{Size}(h)) + l * \mathcal{CostFun}(F)(m) + l + 1 \\
&= \mathcal{Cost}(\mathsf{cons}\ (F\ h)\ (\mathsf{map}\ F\ t))
\end{aligned}$$

Hence, all value instantiations of the left-hand side of this rule both have greater cost, and greater-than-or-equal size, to the right-hand sides. If the other rules are similarly oriented, we can conclude that $\mathcal{CostNum}(s)$ provides a bound on the reduction cost of s.

In the rest of this paper, the ideas above will be formally defined and their correctness proven. We will not use the elaborate names $\mathcal{CostNum}$, $\mathcal{Size}$, etc., but rather define interpretations as tuples that contain all these components.

4 Cost–Size Semantics for Simple Types

In this section we build a set-theoretical cost–size semantics to the simple types in $\mathbb{T}_{\mathbb{B}}$. The goal is to define a function $(\!|\cdot|\!)$ that maps each type $\sigma \in \mathbb{T}_{\mathbb{B}}$ to a well-founded set $(\!|\sigma|\!)$, the cost–size interpretation of σ. We start by formally defining what we mean by cost–size sets.

▶ **Definition 9.** Given a well-founded set $(\mathcal{C}, >, \gtrsim)$, called the *cost set*, and a quasi-ordered set $(\mathcal{S}, \sqsupseteq)$, called the *size set*, we call $\mathcal{C} \times \mathcal{S}$ the **cost–size product** of $(\mathcal{C}, >, \gtrsim)$ and $(\mathcal{S}, \sqsupseteq)$, and its elements *cost–size tuples.*

Given a cost–size product $\mathcal{C} \times \mathcal{S}$, the well-foundedness of $\mathcal{C}$ and quasi-ordering on $\mathcal{S}$ naturally induce an order structure on the product $\mathcal{C} \times \mathcal{S}$ as follows.

▶ **Definition 10** (Product Order)**.** Let $(\mathcal{C}, >, \gtrsim) \times (\mathcal{S}, \sqsupseteq)$ be a cost–size product. Then we define the relations $\succ, \succcurlyeq$ over $\mathcal{C} \times \mathcal{S}$ as follows: for all $\langle x, y\rangle$ and $\langle x', y'\rangle$ in $\mathcal{C} \times \mathcal{S}$,

- $\langle x, y\rangle \succ \langle x', y'\rangle$ iff $x > x'$ and $y \sqsupseteq y'$, and
- $\langle x, y\rangle \succcurlyeq \langle x', y'\rangle$ iff $x \gtrsim x'$ and $y \sqsupseteq y'$.

Next, we show that the triple $(\mathcal{C} \times \mathcal{S}, \succ, \succcurlyeq)$ is well-founded.

▶ **Lemma 11.** The triple $(\mathcal{C} \times \mathcal{S}, \succ, \succcurlyeq)$ defined in Definition 10 is a well-founded set.

Proof. It follows immediately from Definition 10 that $\succ, \succcurlyeq$ are transitive and $\succcurlyeq$ is reflexive. To show well-foundedness of $\succ$ we note that the existence of an infinite chain $\langle x_1, y_1\rangle \succ \langle x_2, y_2\rangle \succ \cdots$ would imply $x_1 > x_2 > \cdots$, which cannot be the case since $>$ is well-founded. We still need to check that $\succcurlyeq$ is compatible with $\succ$.

- Suppose $\langle x, y\rangle \succ \langle x', y'\rangle$. Since $x > x'$ implies $x \gtrsim x'$, we have $\langle x, y\rangle \succcurlyeq \langle x', y'\rangle$.
- Suppose $\langle x, y\rangle \succ \langle x', y'\rangle \succcurlyeq \langle x'', y''\rangle$. Since $x > x' \gtrsim x''$ implies $x > x''$ and $\sqsupseteq$ is transitive, we have $\langle x, y\rangle \succ \langle x'', y''\rangle$. ◀

We shall use product orders to induce well-founded ordering on cost–size sets. Let us define next the requirements for the sets used for size interpretations.

▶ **Definition 12** (Type Interpretation Key). Let $\mathbb{B}$ be a set of base types. An **interpretation key** for $\mathbb{B}$, denoted $\mathcal{J}_{\mathbb{B}}$, is a function that maps each base type $\iota \in \mathbb{B}$ to a quasi-ordered set $(\mathcal{J}_{\mathbb{B}}(\iota), \sqsupseteq)$ with a minimum element, i.e., it contains an element $\bot$, such that $x \sqsupseteq \bot$ for all x.

▶ **Example 13** (Cost–Size Tuples over natural numbers). A first example of an interpretation key is that of tuples over $\mathbb{N}$. For each $\iota \in \mathbb{B}$, $\mathcal{J}_{\mathbb{B}/\mathbb{N}}(\iota)$ is a set of the form $(\mathbb{N}^{K(\iota)}, \sqsupseteq)$, with $K(\iota) \geq 1$ and $(x_1, \ldots, x_{K(\iota)}) \sqsupseteq (y_1, \ldots, y_{K(\iota)})$ iff $x_i \geq y_i$ for all $1 \leq i \leq K(\iota)$. A minimum element for such sets is $(0, \ldots, 0)$. Notice that $(\mathbb{N}^{K(\iota)}, \sqsupseteq)$ is quasi-ordered for any choice of $K(\iota)$ and $\mathcal{J}_{\mathbb{B}/\mathbb{N}}$ is completely determined by a function mapping each $\iota \in \mathbb{B}$ to $K(\iota) \in \mathbb{N}$.

The definition below formalizes our intuition for cost and size from Section 3. Given an interpretation key $\mathcal{J}_{\mathbb{B}}$ we inductively interpret the elements of $\mathbb{T}_{\mathbb{B}}$ as cost–size products.

▶ **Definition 14** (Interpretation of Types). Let $\mathcal{J}_{\mathbb{B}}$ be an interpretation key. We define for each type σ the **cost–size tuple interpretation** of σ as the set $(\!|\sigma|\!) = \mathcal{C}_\sigma \times \mathcal{S}_\sigma$ where $\mathcal{C}_\sigma$ and $\mathcal{S}_\sigma$ are defined as follows (mutually with the set $\mathcal{F}^{\mathsf{c}}_\sigma$):

$$\mathcal{C}_\sigma = \mathbb{N} \times \mathcal{F}^{\mathsf{c}}_\sigma \qquad \mathcal{S}_\iota = \mathcal{J}_{\mathbb{B}}(\iota)$$
$$\mathcal{F}^{\mathsf{c}}_\iota = \texttt{unit} \qquad \mathcal{S}_{\sigma \Rightarrow \tau} = \mathcal{S}_\sigma \Longrightarrow \mathcal{S}_\tau$$
$$\mathcal{F}^{\mathsf{c}}_{\sigma \Rightarrow \tau} = (\mathcal{F}^{\mathsf{c}}_\sigma \times \mathcal{S}_\sigma) \Longrightarrow \mathcal{C}_\tau$$

The set $(\!|\sigma|\!)$ is ordered as follows:

- $\langle (n, f_1), f_2 \rangle \succ \langle (m, g_1), g_2 \rangle$ if $n > m$, $f_1 \gtrsim g_1$ and $f_2 \sqsupseteq g_2$, and
- $\langle (n, f_1), f_2 \rangle \succcurlyeq \langle (m, g_1), g_2 \rangle$ if $n \geq m$, $f_1 \gtrsim g_1$ and $f_2 \sqsupseteq g_2$.

We say a function f is a *cost (size) function* whenever $f \in \mathcal{F}^{\mathsf{c}}_\sigma$ ($f \in \mathcal{S}_\sigma$), for some type σ.

▶ **Lemma 15.** For any type σ, $(\mathcal{C}_\sigma, >, \gtrsim)$ is well-founded and $(\mathcal{S}_\sigma, \sqsupseteq)$ is quasi-ordered with a minimum. Therefore, $(\!|\sigma|\!)$ is a cost–size product.

Proof. When σ is a base type, $\mathcal{C}_\sigma = \mathbb{N} \times \texttt{unit} \cong \mathbb{N}$ and $\mathcal{S}_\sigma = \mathcal{J}_{\mathbb{B}}(\sigma)$, so the statement is trivially true. Let $\sigma = \tau \Rightarrow \rho$, then by induction hypothesis $\mathcal{S}_\tau$ and $\mathcal{S}_\rho$ are quasi-ordered. Quasi-ordering of $(\mathcal{S}_{\tau \Rightarrow \rho}, \sqsupseteq)$ follows from the induced point-wise comparison. A minimum for this size set is the function $\boldsymbol{\lambda} x.\bot$. Well-foundedness of $(\mathcal{C}_\sigma, \succ, \succcurlyeq)$ follows from Lemma 11 by showing that $\mathcal{F}^{\mathsf{c}}_{\tau \Rightarrow \rho}$ is quasi ordered. ◀

To map each term $s : \sigma$ to an element of $(\!|\sigma|\!)$ (Definition 25), we need a notion of application for cost-size tuples. More precisely, assume given a type $\sigma \Rightarrow \tau$ and cost–size tuples $\boldsymbol{f} \in (\!|\sigma \Rightarrow \tau|\!)$ and $\boldsymbol{x} \in (\!|\sigma|\!)$. We define the application of $\boldsymbol{f}$ to $\boldsymbol{x}$, denoted $\boldsymbol{f} \cdot \boldsymbol{x}$, as follows.

▶ **Definition 16.** Let $\sigma \Rightarrow \tau$ be an arrow type, $\boldsymbol{f} = \langle (n, f^{\mathsf{c}}), f^{\mathsf{s}} \rangle \in (\!|\sigma \Rightarrow \tau|\!)$, and $\boldsymbol{x} = \langle (m, x^{\mathsf{c}}), x^{\mathsf{s}} \rangle \in (\!|\sigma|\!)$. The **semantic application** of $\boldsymbol{f}$ to $\boldsymbol{x}$, denoted $\boldsymbol{f} \cdot \boldsymbol{x}$, is defined by:

$$\text{let } f^{\mathsf{c}}(x^{\mathsf{c}}, x^{\mathsf{s}}) = (k, h); \text{ then } \langle (n, f^{\mathsf{c}}), f^{\mathsf{s}} \rangle \cdot \langle (m, x^{\mathsf{c}}), x^{\mathsf{s}} \rangle = \langle (n + m + k, h), f^{\mathsf{s}}(x^{\mathsf{s}}) \rangle$$

We set the semantic application to be left-associative, so $f \cdot g \cdot h$ denotes $(f \cdot g) \cdot h$.

▶ **Example 17.** Let us illustrate semantic application with a concrete example: consider the type $\sigma = (\mathsf{nat} \Rightarrow \mathsf{nat}) \Rightarrow \mathsf{list} \Rightarrow \mathsf{list}$, which is the type of map defined in Example 1. The function map takes as argument a function $F : \mathsf{nat} \Rightarrow \mathsf{nat}$ and list q and applies F to each element of q. This formalizes the cost and size ideas in Examples 5 and 6. Hence, the cost–size interpretation of map is an element $\langle (n, f^{\mathsf{c}}), f^{\mathsf{s}} \rangle$ of $(\!|\sigma|\!)$. Its cost component (n, f^{c}) is in $\mathcal{C}_\sigma = \mathbb{N} \times \mathcal{F}^{\mathsf{c}}_\sigma$ which is composed of a numeric and functional component. The numeric component n carries the cost of partial application. Meanwhile, the functional component in $\mathcal{F}^{\mathsf{c}}_{\sigma}$ is parametrized by functional arguments carrying the cost and size information of F. Indeed, Definition 14 gives us $f^{\mathsf{c}} : \mathcal{F}^{\mathsf{c}}_{\mathsf{nat} \Rightarrow \mathsf{nat}} \times \mathcal{S}_{\mathsf{nat} \Rightarrow \mathsf{nat}} \Longrightarrow \mathcal{C}_{\mathsf{list} \Rightarrow \mathsf{list}}$, which can be written explicitly as:

$$\overbrace{\left(\underbrace{(\mathtt{unit} \times \mathbb{N} \Longrightarrow \mathbb{N} \times \mathtt{unit})}_{\text{cost of } F} \times \underbrace{(\mathcal{S}_{\mathsf{nat}} \Longrightarrow \mathcal{S}_{\mathsf{nat}})}_{\text{size of } F}\right) \Longrightarrow \left(\mathbb{N} \times \left[\underbrace{\mathtt{unit}}_{q^{\mathsf{c}}} \times \underbrace{\mathcal{S}_{\mathsf{list}}}_{q^{\mathsf{s}}} \Longrightarrow \mathbb{N} \times \mathtt{unit}\right]\right)}^{\text{the functional cost of } \mathsf{map}}$$

The set for the size function is somewhat simpler with $f^{\mathsf{s}} : (\mathcal{S}_{\mathsf{nat}} \Longrightarrow \mathcal{S}_{\mathsf{nat}}) \Longrightarrow \mathcal{S}_{\mathsf{list}} \Longrightarrow \mathcal{S}_{\mathsf{list}}$.

Therefore, we apply $\boldsymbol{f}$ to a cost-size tuple $\boldsymbol{x}$ of the form $\langle (m, x^{\mathsf{c}}), x^{\mathsf{s}} \rangle$ where x^{c} is the cost of computing F (so an element of $\mathcal{F}^{\mathsf{c}}_{\mathsf{nat} \Rightarrow \mathsf{nat}}$) and x^{s} is the size of F, so an element of $\mathcal{S}_{\mathsf{nat} \Rightarrow \mathsf{nat}}$. We proceed by applying the respective functions so $f^{\mathsf{c}}(x^{\mathsf{c}}, x^{\mathsf{s}}) = (k, h)$ belongs to $\mathcal{C}_{\mathsf{list} \Rightarrow \mathsf{list}}$ and $f^{\mathsf{s}}(x^{\mathsf{s}})$ is in $\mathcal{S}_{\mathsf{list} \Rightarrow \mathsf{list}}$. We put everything together and add the numeric components to obtain: $\boldsymbol{f} \cdot \boldsymbol{x} = \langle (n + m + k, h), f^{\mathsf{s}}(x^{\mathsf{s}}) \rangle$. Notice that this gives us a new cost–size tuple with the cost component in $\mathbb{N} \times (\mathcal{C}_{\mathsf{list}} \Longrightarrow \mathcal{C}_{\mathsf{list}})$ and size component in $\mathcal{S}_{\mathsf{list}} \Longrightarrow \mathcal{S}_{\mathsf{list}}$, which is a tuple in $(\!|\mathsf{list} \Rightarrow \mathsf{list}|\!)$.

Observe that our intention with Definition 16 is that the semantic application conforms with a form of "application typing rule". A straightforward analysis on Definition 16 shows that this is indeed the case. This is summarized in the lemma below.

▶ **Lemma 18.** If $\boldsymbol{f} \in (\!|\sigma \Rightarrow \tau|\!)$ and $\boldsymbol{x} \in (\!|\sigma|\!)$, then $\boldsymbol{f} \cdot \boldsymbol{x}$ belongs to $(\!|\tau|\!)$.

Definition 14 gives us a family of cost–size sets $\mathcal{T} = \{(\!|\sigma|\!)\}_{\sigma \in \mathbb{T}_{\mathbb{B}}}$ indexed by $\mathbb{T}_{\mathbb{B}}$, and combined with Definition 16 we get a family of *application operators*

$$(\mathcal{T}, \cdot) = \left(\{(\!|\sigma|\!)\}_{\sigma \in \mathbb{T}_{\mathbb{B}}}, \{\cdot_{\sigma,\tau}\}_{\sigma,\tau \in \mathbb{T}_{\mathbb{B}}}\right), \text{ with } \cdot_{\sigma,\tau} : (\!|\sigma \Rightarrow \tau|\!) \times (\!|\sigma|\!) \longrightarrow (\!|\tau|\!)$$

We call the pair $(\mathcal{T}, \cdot)$ the cost–size type structure generated by the interpretation key $\mathcal{J}_{\mathbb{B}}$. Indeed, in the next Lemma we show that such structure preserves the orderings $\succ$ and $\succcurlyeq$ on cost–size tuples.

▶ **Lemma 19.** The application operator is strongly monotonic in both arguments.

Proof. We need to prove the following: (i) if $\boldsymbol{f} \succ \boldsymbol{g}$ and $\boldsymbol{x} \succcurlyeq \boldsymbol{y}$, then $\boldsymbol{f} \cdot \boldsymbol{x} \succ \boldsymbol{g} \cdot \boldsymbol{y}$; (ii) if $\boldsymbol{f} \succcurlyeq \boldsymbol{g}$ and $\boldsymbol{x} \succ \boldsymbol{y}$, then $\boldsymbol{f} \cdot \boldsymbol{x} \succ \boldsymbol{g} \cdot \boldsymbol{y}$; (iii) if $\boldsymbol{f} \succcurlyeq \boldsymbol{g}$ and $\boldsymbol{x} \succcurlyeq \boldsymbol{y}$, then $\boldsymbol{f} \cdot \boldsymbol{x} \succcurlyeq \boldsymbol{g} \cdot \boldsymbol{y}$. Consider cost–size tuples $\boldsymbol{f}, \boldsymbol{g} \in (\!|\sigma \Rightarrow \tau|\!)$ and $\boldsymbol{x}, \boldsymbol{y} \in (\!|\sigma|\!)$. Let $\boldsymbol{f} = \langle (n, f^{\mathsf{c}}), f^{\mathsf{s}} \rangle$, $\boldsymbol{g} = \langle (m, g^{\mathsf{c}}), g^{\mathsf{s}} \rangle$, $\boldsymbol{x} = \langle (j, x^{\mathsf{c}}), x^{\mathsf{s}} \rangle$, and $\boldsymbol{y} = \langle (j', y^{\mathsf{c}}), y^{\mathsf{s}} \rangle$. We proceed to show (i) and observe that (ii) and (iii) follow similar reasoning. Indeed, if $\boldsymbol{f} \succ \boldsymbol{g}$ and $\boldsymbol{x} \succcurlyeq \boldsymbol{y}$ we have that $n > m$, $f^{\mathsf{c}} \gtrsim g^{\mathsf{c}}$, $f^{\mathsf{s}} \sqsupseteq g^{\mathsf{s}}$, $j \geq j'$, $x^{\mathsf{c}} \gtrsim y^{\mathsf{c}}$, and $x^{\mathsf{s}} \sqsupseteq y^{\mathsf{s}}$. Let $f^{\mathsf{c}}(x^{\mathsf{c}}, x^{\mathsf{s}}) = (k, h)$ and $g^{\mathsf{c}}(y^{\mathsf{c}}, y^{\mathsf{s}}) = (k', h')$, we get:

$$\boldsymbol{f} \cdot \boldsymbol{x} = \langle (n + j + k, h), f^{\mathsf{s}}(x^{\mathsf{s}}) \rangle \succ \langle (m + j' + k', h'), g^{\mathsf{s}}(y^{\mathsf{s}}) \rangle = \boldsymbol{g} \cdot \boldsymbol{y}$$

◀

▶ Remark 20. Notice that the type structure $(\mathcal{T}, \cdot)$ is nonstandard. Indeed, the intended standard semantics given to arrow types is usually a functional space [5, Chapter 3]. So inhabitants of functional types are interpreted as functions. Since our intention with defining *cost–size type structures* as above is to capture the complexity-wise behavior of functions (defined by rewriting rules) and a cost component associated with the computational environment, this non-standardness is expected. In the next sections we show that even though our interpretations do not give rise to a standard semantic of simple types, we can still prove classical lemmata for substitution and compatibility.

▶ **Example 21.** In Examples 1 and 2 we have two examples of base types: nat and list. Values of type nat are built using the constructors $\mathsf{0} : \mathsf{nat}$ and $\mathsf{s} : \mathsf{nat} \Rightarrow \mathsf{nat}$. Similarly, for list we have $\mathsf{nil} : \mathsf{list}$ and $\mathsf{cons} : \mathsf{nat} \Rightarrow \mathsf{list} \Rightarrow \mathsf{list}$.

Let us give a cost–size type structure over $\mathbb{N}$ (Example 13) for $\mathbb{B} = \{\mathsf{nat}, \mathsf{list}\}$. Essentially, we need to choose the numbers $K(\mathsf{nat}), K(\mathsf{list})$ associated with nat and list, respectively. To do so we take the intentional size semantic of $\mathsf{nat}, \mathsf{list}$ into account. Let us set $K(\mathsf{nat}) = 1$ and $K(\mathsf{list}) = 2$. This exactly gives the size sets we used in Section 3, and allows us to use "number of symbols" as a notion of size in a unary representation of numbers, and (*length*, *maximum element size*) as a size notion for lists. Intuitively, since a list is a container-like data structure we want to be able to simultaneously give upper bounds to "the size of the container" (which is *length* for lists) and "the size of its elements". This choice of $\mathcal{J}_{\mathbb{B}/\mathbb{N}}$ affects the shape of interpretations for symbols in Σ, as we will see in Example 23.

Even though we have manually chosen the size tuples for $\mathcal{J}_{\mathbb{B}/\mathbb{N}}$ above, an automated procedure can still be devised to determine the number $K(\iota)$, for $\iota \in \mathbb{B}$. A description of such a procedure can be found in [22].

5 Cost–Size Semantics for Terms

In the previous section, we established a cost–size semantics for the simple types in $\mathbb{T}_{\mathbb{B}}$. Our goal in this section is to interpret terms as elements of those sets.

An interpretation of a signature $\mathbb{F} = (\mathbb{B}, \Sigma, \mathtt{ar})$ interprets the base types in $\mathbb{B}$ and each $\mathsf{f} \in \Sigma$ of arity $\mathtt{ar}(f) = \sigma$ as an element of $(\!|\sigma|\!)$ which is constructed by Definition 14. This is formally stated in the definition below.

▶ **Definition 22.** A **cost–size tuple interpretation** $\mathcal{F}$ for a signature $\mathbb{F} = (\mathbb{B}, \Sigma, \mathtt{ar})$ consists of a pair of functions $(\mathcal{J}_{\mathbb{B}}, \mathcal{J}_{\Sigma})$ where

- $\mathcal{J}_{\mathbb{B}}$ is a type interpretation key (Definition 12),
- $\mathcal{J}_{\Sigma}$ is an *interpretation of symbols* in Σ which maps each $\mathsf{f} \in \Sigma$ with $\mathtt{ar}(\mathsf{f}) = \sigma$ to a cost–size tuple in $(\!|\sigma|\!)$, where $(\!|\sigma|\!)$ is built using $\mathcal{J}_{\mathbb{B}}$ in Definition 14.

In what follows we slightly abuse notation by writing $\mathcal{J}_{\mathsf{f}}$ for $\mathcal{J}_{\Sigma}(\mathsf{f})$ and just $\mathcal{J}$ for $\mathcal{J}_{\Sigma}$.

▶ **Example 23.** As a first example of interpretation, let us interpret the data signature from Example 21. Recall that $\mathsf{0} : \mathsf{nat}$, $\mathsf{s} : \mathsf{nat} \Rightarrow \mathsf{nat}$ are the constructors for nat and $K(\mathsf{nat}) = 1$.

$$\mathcal{J}_{\mathsf{0}} = \Big\langle\ (0, \mathtt{u})\ , 1 \Big\rangle \qquad\qquad \mathcal{J}_{\mathsf{s}} = \Big\langle\ (0, \boldsymbol{\lambda} x.(0, \mathtt{u}))\ , \boldsymbol{\lambda} x.x + 1 \Big\rangle$$

The highlighted cost components for the constructors are filled with zeroes. That is because in the rewriting cost model data values do not fire rewriting sequences. In the language of Section 3: the *cost number* for $\mathsf{0}$ is 0, (because it is a value), the *cost function* is $\mathtt{u}$ (because it has base type), and *size component* is 1 (since we chose a notion of size for terms of type nat to mean "number of symbols"). The cost number for s is 0, the cost function is the constant function mapping to 0, and the size component is the function $\boldsymbol{\lambda} x.x + 1$ in $\mathcal{S}_{\mathsf{nat} \Rightarrow \mathsf{nat}}$. We interpret the constructors for list, i.e., nil and cons, following the same principle, with $K(\mathsf{list}) = 2$. We write a size tuple q in $\mathcal{S}_{\mathsf{list}}$ as $(q_{\mathsf{l}}, q_{\mathsf{m}})$ since the first component is to mean the length of the list and the second a bound on the size of its elements.

$$\mathcal{J}_{\mathsf{nil}} = \Big\langle\ (0, \mathtt{u})\ , (0, 0) \Big\rangle \quad \mathcal{J}_{\mathsf{cons}} = \Big\langle\ (0, \boldsymbol{\lambda} x.(0, \boldsymbol{\lambda} q.(0, \mathtt{u})))\ , \boldsymbol{\lambda} xq.(q_{\mathsf{l}} + 1, \max(x, q_{\mathsf{m}})) \Big\rangle$$

The highlighted cost components are filled with zeroes for lists as well. Size components are interpreted as expected, and exactly following Example 5.

The next step is to extend the interpretation of a signature $\mathbb{F}$ to the set of terms. But first, we define *valuation functions* to interpret the variables in $x : \sigma$ as elements of $(\!|\sigma|\!)$.

▶ **Definition 24.** A **cost–size valuation** α is a function that maps each $x : \sigma$ to a cost-size tuple in $(\!|\sigma|\!)$ such that:

- $\alpha(x) = \langle (0, \mathtt{u}), x^{\mathsf{s}} \rangle$, for all $x \in \mathbb{X}$ of base type, and
- $\alpha(F) = \langle (0, F^{\mathsf{c}}), F^{\mathsf{s}} \rangle$ when $F :: \sigma \Rightarrow \tau$.

Notice that, in this definition, the cost component of $\alpha(x)$ has the form $(0, \mathtt{u})$, if $x : \iota$. This interpretation is motivated by Definition 4, where a matching substitution γ (i.e., a substitution such that $\ell\gamma \to_v r\gamma$) must map each $x : \iota$ to a value of base type. Those can only have the form $\mathsf{c}(v_1, \dots, v_m)$ with $\mathsf{c} \in \Sigma^{\mathsf{con}}$. Variables of arrow type still have a cost number 0; however, they can be instantiated to values that carry *indirect* computational content: a partial application or abstraction. For instance, a variable of type $F : \mathsf{nat} \Rightarrow \mathsf{nat}$ can be instantiated with $\mathsf{add}\,0$, which is a value that produces a cost as soon as it is applied to the next argument. We use the notation $F^{\mathsf{c}}/F^{\mathsf{s}}$ to denote the cost/size component of $\alpha(F)$.

▶ **Definition 25.** Assume given a signature $\mathbb{F} = (\mathbb{B}, \Sigma, \mathtt{ar})$ and its cost–size tuple interpretation $\mathcal{F} = (\mathcal{J}_{\mathbb{B}}, \mathcal{J})$ together with a valuation α. The **term interpretation** $[\![s]\!]^{\mathcal{J}}_{\alpha}$ of s under $\mathcal{J}$ and α is defined by induction on the structure of s as follows:

$$[\![x]\!]^{\mathcal{J}}_{\alpha} = \alpha(x) \qquad [\![\mathsf{f}]\!]^{\mathcal{J}}_{\alpha} = \mathcal{J}_{\mathsf{f}} \qquad [\![s\,t]\!]^{\mathcal{J}}_{\alpha} = [\![s]\!]^{\mathcal{J}}_{\alpha} \cdot [\![t]\!]^{\mathcal{J}}_{\alpha}$$

$$[\![\lambda x.\, s]\!]^{\mathcal{J}}_{\alpha} = \Big\langle \Big(0, \boldsymbol{\lambda} d.(1 + \pi_{11}([\![s]\!]^{\mathcal{J}}_{[x:-d]\alpha}), \pi_{12}([\![s]\!]^{\mathcal{J}}_{[x:-d]\alpha}))\Big), \boldsymbol{\lambda} d^{\mathsf{s}}.\pi_2([\![s]\!]^{\mathcal{J}}_{[x:-(\underline{0},d)]\alpha})\Big\rangle,$$

where π_i is the projection on the ith-component and π_{ij} is the composition $\pi_j \circ \pi_i$, and $\underline{0}$ is a cost function of the form $\boldsymbol{\lambda} x_1.(0, \boldsymbol{\lambda} x_2 \dots (0, \mathtt{u}) \dots)$. If $d = (d^c, d^s)$, the notation $[x :- d]\alpha$ denotes the valuation that maps x to $\langle (0, d^c), d^s \rangle$ and every other variable y to $\alpha(y)$.

We write $[\![s]\!]$ for $[\![s]\!]^{\mathcal{J}}_{\alpha}$ whenever α and $\mathcal{J}$ are universally quantified or clear from the context.

The interpretation for abstractions may seem baroque, but can be understood as follows: an abstraction is a value, so its cost number is 0. The cost of applying that abstraction on a value v is 1 plus the cost number for $s[x := v]$ – which is obtained by evaluating $[\![s]\!]^{\mathcal{J}}_{[x:-d]\alpha}$ if d is the cost function/size pair for v. The cost *function* of this application is exactly the cost function of $s[x := v]$. The *size* of an abstraction $\lambda x.s$ is exactly the function that takes a size and maps it to the size interpretation of s where x is mapped to that size. Technically, to obtain the size component of $[\![s]\!]^{\mathcal{J}}_{[x:-d]\alpha}$ we also need a cost component, but by definition, this component does not play a role, so we can safely choose an arbitrary pair $\underline{0}$ in the right set.

▶ **Example 26.** We continue with Example 23 by interpreting ground constructor terms fully. A ground constructor term d of type nat is of the form $\mathsf{s}\,(\mathsf{s} \dots (\mathsf{s}\,0) \dots)$ where the number $n \in \mathbb{N}$ is represented by n successive applications of s to $\mathsf{0}$. Let us write n as shorthand notation for such terms. Similarly, for ground constructor terms of type list, we write $[\mathsf{n}_1; \dots; \mathsf{n}_k]$ for the term $\mathsf{cons}\,\mathsf{n}_1 \dots (\mathsf{cons}\,\mathsf{n}_k\,\mathsf{nil})$. The empty list constructor nil is written as $[]$ in this notation. Hence, the cost–size interpretation of $3 : \mathsf{nat}$ is given by:

$$[\![3]\!] = [\![\mathsf{s}\,(\mathsf{s}\,(\mathsf{s}\,0))]\!] = [\![\mathsf{s}]\!] \cdot ([\![\mathsf{s}]\!] \cdot ([\![\mathsf{s}]\!] \cdot [\![0]\!])) = \Big\langle\ (0, \mathtt{u})\ , 4 \Big\rangle.$$

Consider, for instance, the list $[1; 7; 9]$. Its cost–size interpretation is given by:

$$[\![[1; 7; 9]]\!] = [\![\mathsf{cons}\,1\,(\mathsf{cons}\,7\,(\mathsf{cons}\,9\,\mathsf{nil}))]\!] = \Big\langle\ (0, \mathtt{u})\ , (3, 10) \Big\rangle.$$

The important information we can extract from such interpretations is their size component. Indeed, $[\![3]\!]^{\mathsf{s}} = 4$ counts the number of constructor symbols in the term representation 3 and

$[\![[1;7;9]]\!]^{\mathsf{s}} = (3, 10)$ gives us the length and an upper bound on the size of each element in $[1; 7; 9]$. The size interpretation for the constructors of nat and list correctly capture our notion of "size" given in Example 21.

The next Lemma expresses the soundness of term interpretation, that is, the interpretation of terms preserves the type structure:

▶ **Lemma 27** (Type Soundness)**.** If $s : \sigma$ then $[\![s]\!] \in (\!|\sigma|\!)$.

Proof. The proof is by induction on the structure of s. The base cases follow directly from Definitions 22 and 24. We use Lemmas 18 and 19 in the application case. The abstraction case follows from the induction hypothesis and weak monotonicity of π_i. ◀

Up to now, we have given cost–size semantics for types and terms. Observe that Definition 22 only requires that we interpret function symbols as cost–size tuples in the correct domain. For instance, we might interpret all function components as constant functions. This is a valid, but not so useful, interpretation of terms. So we move on to the next component of our interpretation framework: we want to interpret terms in such a way that $[\![s]\!] \succ [\![t]\!]$ whenever $s \to t$, for any pair of terms s, t.

▶ **Definition 28.** Consider a signature $\mathbb{F} = (\mathbb{B}, \Sigma, \mathtt{ar})$. A **cost–size call-by-value termination model** for a term rewriting system $(\mathbb{F}, \mathbb{R})$ consists of the following ingredients:

- an interpretation key $\mathcal{J}_{\mathbb{B}}$ (Definition 12), together with
- a cost–size interpretation $(\mathcal{J}_{\mathbb{B}}, \mathcal{J}_{\Sigma})$ (Definition 22),

such that the following **compatibility conditions** hold:

- for all value substitutions γ and all terms s and t, $[\![s\gamma]\!] \succ [\![t\gamma]\!]$ whenever $[\![s]\!] \succ [\![t]\!]$;
- for every term s and value v, $[\![(\lambda x. s)\, v]\!] \succ [\![s[x :- v]]\!]$;
- for all terms s and t,
  - $[\![s\, t]\!] \succ [\![s'\, t]\!]$ whenever $[\![s]\!] \succ [\![s']\!]$, and $[\![s\, t]\!] \succ [\![s\, t']\!]$ whenever $[\![t]\!] \succ [\![t']\!]$;
- for all rules $\ell \to r \in \mathbb{R}$, we have $[\![\ell]\!] \succ [\![r]\!]$.

Roughly speaking, a call-by-value termination model is an interpretation of types and terms that is compatible with each rule in $\mathbb{R}$, the call-by-value beta rule and the formation of terms, and which is closed under value substitutions. By a straightforward induction on the reduction $s \to_v t$, we can establish the following result.

▶ **Theorem 29.** Let $(\mathbb{F}, \mathbb{R})$ be a TRS. If we have a termination model of $(\mathbb{F}, \mathbb{R})$, then the higher-order call-by-value rewriting relation $\to_v$ is strongly normalizing.

Hence, termination models collect sufficient conditions for strong normalization. The lemmata below are to show that cost–size interpretations satisfy some of the compatibility conditions for termination models. Let us first prove closure under substitutions.

▶ **Definition 30.** Given a substitution γ and valuation α, we define the **γ-extension of α** as the valuation defined by $\alpha^{\gamma} = [\![\cdot]\!]_{\alpha}^{\mathcal{J}} \circ \gamma$.

▶ **Lemma 31.** If $x \notin \mathtt{fv}(s)$ then $[\![s]\!]_{[x:-d]\alpha} = [\![s]\!]_{\alpha}$. Consequently, if x is not free in $y\gamma$ for any variable y, then $([x :- d]\alpha)^{\gamma} = [x :- d]\alpha^{\gamma}$.

▶ **Lemma 32** (Substitution Lemma)**.** For any value substitution γ and valuation α, we have that $[\![s\gamma]\!]_{\alpha} = [\![s]\!]_{\alpha^{\gamma}}$.

Proof. Let us work out the abstraction case $s = \lambda x.\, t$. Since we assume that the application of substitution is capture-avoiding, we can assume that x does not occur free in any term in the range of γ. Hence,

$$\begin{aligned}
[\![\lambda x.\, (t\gamma)]\!]_\alpha &= \Big\langle \Big(0, \boldsymbol{\lambda} d.(1 + \pi_{11}([\![t\gamma]\!]^{\mathcal{J}}_{[x:-d]\alpha}), \pi_{12}([\![t\gamma]\!]^{\mathcal{J}}_{[x:-d]\alpha}))\Big)\, , \boldsymbol{\lambda} d^{\mathsf{s}}.\pi_2([\![t\gamma]\!]^{\mathcal{J}}_{[x:-(\underline{0},d)]\alpha})\Big\rangle \\
&\overset{IH}{=} \Big\langle \Big(0, \boldsymbol{\lambda} d.(1 + \pi_{11}([\![t]\!]^{\mathcal{J}}_{([x:-d]\alpha)^\gamma}), \pi_{12}([\![t]\!]^{\mathcal{J}}_{([x:-d]\alpha)^\gamma}))\Big), \boldsymbol{\lambda} d^{\mathsf{s}}.\pi_2([\![t]\!]^{\mathcal{J}}_{([x:-(\underline{0},d)]\alpha)^\gamma})\Big\rangle \\
&= \Big\langle \Big(0, \boldsymbol{\lambda} d.(1 + \pi_{11}([\![t]\!]^{\mathcal{J}}_{[x:-d]\alpha^\gamma}), \pi_{12}([\![t]\!]^{\mathcal{J}}_{[x:-d]\alpha^\gamma}))\Big)\, , \boldsymbol{\lambda} d^{\mathsf{s}}.\pi_2([\![t]\!]^{\mathcal{J}}_{[x:-(\underline{0},d)]\alpha^\gamma})\Big\rangle \\
&= [\![\lambda x.\, t]\!]_{\alpha^\gamma}.
\end{aligned}$$

◀

As a consequence of the substitution lemma, if $[\![s]\!]^{\mathcal{J}}_\alpha \succ [\![t]\!]^{\mathcal{J}}_\alpha$ for all α, then $[\![s\gamma]\!]^{\mathcal{J}}_\alpha \succ [\![t\gamma]\!]^{\mathcal{J}}_\alpha$ for all α. Consequently, the first compatibility condition is valid for any interpretation. The second compatibility requirement is for β reductions.

▶ **Lemma 33.** The call-by-value beta rule scheme $(\lambda x.\, s)\, v \to_v s[x :- v]$ is strictly decreasing for any cost–size interpretation.

Proof. The proof reduces to checking $[\![(\lambda x.\, s)\, v]\!] \succ [\![s[x :- v]]\!]$. Let $[\![v]\!] = \langle (0, v^{\mathsf{c}}), v^{\mathsf{s}}\rangle$, and denote V for the pair $(v^{\mathsf{c}}, v^{\mathsf{s}})$. Then we have the following:

$$\begin{aligned}
[\![(\lambda x.\, s)\, v]\!] &= [\![\lambda x.\, s]\!] \cdot [\![v]\!] \\
&= \Big\langle \Big(0, \boldsymbol{\lambda} d.(1 + \pi_{11}([\![s]\!]^{\mathcal{J}}_{[x:-d]\alpha}), \pi_{12}([\![s]\!]^{\mathcal{J}}_{[x:-d]\alpha}))\Big)\, , \boldsymbol{\lambda} d^{\mathsf{s}}.\pi_2([\![s]\!]^{\mathcal{J}}_{[x:-(\underline{0},d^{\mathsf{s}})]\alpha})\Big\rangle \cdot [\![v]\!] \\
&= \Big\langle \Big(0 + 0 + 1 + \pi_{11}([\![s]\!]^{\mathcal{J}}_{[x:-V]\alpha}), \pi_{12}([\![s]\!]^{\mathcal{J}}_{[x:-V]\alpha})\Big)\, , \pi_2([\![s]\!]^{\mathcal{J}}_{[x:-\langle \underline{0},v^{\mathsf{s}}\rangle]\alpha})\Big\rangle \\
&\succ \Big\langle \Big(\pi_{11}([\![s]\!]^{\mathcal{J}}_{[x:-V]\alpha}), \pi_{12}([\![s]\!]^{\mathcal{J}}_{[x:-V]\alpha})\Big)\, , \pi_2([\![s]\!]^{\mathcal{J}}_{[x:-V]\alpha})\Big\rangle \\
&= [\![s[x :- v]]\!]_\alpha.
\end{aligned}$$

In the second-to-last step, we use that the size component of $[\![s]\!]^{\mathcal{J}}_\alpha$ does not regard any cost component in α, so $\pi_2([\![s]\!]^{\mathcal{J}}_{[x:-\langle \underline{0},v^{\mathsf{s}})\rangle]\alpha}) = \pi_2([\![s]\!]^{\mathcal{J}}_{[x:-V]\alpha})$. In the last step, we use the substitution lemma. ◀

Compatibility over applicative terms is a consequence of Lemma 19. Notice that the results above do not depend on a particular interpretation. Hence, to establish a termination model for a TRS, only the last compatibility condition remains to be checked, i.e., $[\![\ell]\!] \succ [\![r]\!]$ for all rules $\ell \to r$ in $\mathbb{R}$. We collect this fact below, which is a consequence of Theorem 29 and the Lemmas above.

▶ **Corollary 34.** *Let $\mathbb{R}$ be a TRS that admits a cost–size interpretation $(\mathcal{J}_{\mathbb{B}}, \mathcal{J}_\Sigma)$. If $[\![\ell]\!] \succ [\![r]\!]$ for all rules $\ell \to r$ in $\mathbb{R}$, then $\mathbb{R}$ is a termination model, and consequently strongly normalizing.*

Interpretation techniques are usually applied to show full termination [7, 18, 25] or as quasi-orderings for the dependency pair approach [1]. In the next example, we show that cost–size interpretations are weak enough to prove termination of call-by-value systems that do not necessarily terminate under full rewriting.

▶ **Example 35.** Let $\mathsf{a}, \mathsf{b} : \iota$, $\mathsf{g} : \iota \Rightarrow \iota \Rightarrow \iota$, and $\mathsf{f} : \iota \Rightarrow \iota \Rightarrow \iota \Rightarrow \iota$. The rewrite system introduced by Toyama [30] and defined by $\mathbb{R} = \{\mathsf{g}\, x\, y \to x, \mathsf{g}\, x\, y \to y, \mathsf{f}\, \mathsf{a}\, \mathsf{b}\, z \to \mathsf{f}\, z\, z\, z\}$ was given to show that termination is not modular for disjoint unions of TRSs. Indeed, it admits the infinite rewriting sequence $\mathsf{f\,a\,b\,(g\,a\,b)} \to_{\mathbb{R}} \mathsf{f\,(g\,a\,b)\,(g\,a\,b)\,(g\,a\,b)} \to^{+}_{\mathbb{R}} \mathsf{f\,a\,b\,(g\,a\,b)}$, whereas the systems $\mathbb{R}_{\mathsf{g}}$ and $\mathbb{R}_{\mathsf{f}}$ are individually terminating. If we restrict reductions to call-by-value, then the rewrite relation $\to_v$ induced by $\mathbb{R}$ is terminating.

In order to prove termination of $\mathbb{R}$, we introduce a non-numeric notion of size. Let $\mathcal{J}_{\mathbb{B}}(\iota) = \mathcal{P}(\mathsf{T}(\mathbb{F}, \mathbb{X}))$, i.e., the set of all subsets of $\mathsf{T}(\mathbb{F}, \mathbb{X})$. This set is partially ordered by inclusion, so we define $x \sqsupseteq y$ iff $x \supseteq y$ which is a quasi-order. Consider the following interpretation:

$$\mathcal{J}_{\mathsf{a}} = \Big\langle (0, \mathtt{u}) , \{\mathsf{a}\} \Big\rangle \qquad \mathcal{J}_{\mathsf{g}} = \Big\langle (0, \boldsymbol{\lambda} x.(0, \boldsymbol{\lambda} y.(1, \mathtt{u}))) , \boldsymbol{\lambda} xy.x \cup y \Big\rangle$$
$$\mathcal{J}_{\mathsf{b}} = \Big\langle (0, \mathtt{u}) , \{\mathsf{b}\} \Big\rangle \qquad \mathcal{J}_{\mathsf{f}} = \Big\langle (0, \boldsymbol{\lambda} x.(0, \boldsymbol{\lambda} y.(0, \boldsymbol{\lambda} z.(H(x,y), \mathtt{u})))) , \boldsymbol{\lambda} xyz.\emptyset \Big\rangle ,$$

where H is a helper function defined by $H(x, y) = \mathtt{if}\ x^{\mathsf{s}} \sqsupseteq \{\mathsf{a}\} \wedge y^{\mathsf{s}} \sqsupseteq \{\mathsf{b}\}\ \mathtt{then}\ 1\ \mathtt{else}\ 0$. Notice that H is weakly monotonic, and the size tuples for interpretation of values are sets of cardinality ≤ 1. Checking compatibility for this interpretation is straightforward: $[\![\mathsf{g}\,x\,y]\!] = \langle (1, \mathtt{u}), x \cup y\rangle \succ \langle (0, \mathtt{u}), x\rangle = [\![x]\!]$ and $[\![\mathsf{g}\,x\,y]\!] = \langle (1, \mathtt{u}), x \cup y\rangle \succ \langle (0, \mathtt{u}), y\rangle = [\![y]\!]$; and finally $[\![\mathsf{f\,a\,b}\,z]\!] = \langle (1, \mathtt{u}), \emptyset\rangle \succ \langle (0, \mathtt{u}), \emptyset\rangle = [\![\mathsf{f}\,z\,z\,z]\!]$, because any instantiation of z is necessarily a value, so it cannot include both a and b.

6 Complexity Analysis of Call-by-Value Rewriting

In the previous section, we showed that cost–size tuples can be used to establish termination of call-by-value rewriting. In this section, we concentrate on a quantitative analysis of such termination proofs. Hence, the goal is not merely to find tuple interpretations that prove termination but also ones that establish "good" upper bounds on the complexity of reducing terms to normal form. To start, we will extend the notion of *derivation height* to our setting:

▶ **Definition 36.** The weak call-by-value **derivation height** of a term s, notation $\mathtt{dh}_{\mathbb{R}}(s)$, is the largest number n such that $s \to_v s_1 \to_v \ldots \to_v s_n$.

This notion is defined for all terms when the TRS is terminating. We will simply refer to the weak call-by-value derivation height as "derivation height".

The methodology of weakly monotonic algebras offers a systematic way to derive bounds for the derivation height of a given term:

▶ **Lemma 37.** If $[\![s]\!] = \langle (n, F^{\mathsf{c}}), F^{\mathsf{s}}\rangle$, then $\mathtt{dh}_{\mathbb{R}}(s) \leq n$.

Proof. By the lemmas in Section 5 we see that $[\![s]\!] \succ [\![t]\!]$ whenever $s \to t$. Since this implies $\pi_{11}(s) > \pi_{11}(t)$, the lemma follows. ◀

As an illustration of how this is used, we present the formalized examples of Section 3 and complete the interpretation of Examples 1 and 2.

Let us start with the system $\mathbb{R}_{\mathsf{add}}$ which intuitively defines addition over nat. We will use the type and constructor interpretations as given in Example 23. The rules $\mathsf{add}\,x\,\mathsf{0} \to \mathsf{0}$ and $\mathsf{add}\,x\,(\mathsf{s}\,y) \to \mathsf{s}\,(\mathsf{add}\,x\,y)$ suggest the following cost–size interpretation:

$$\mathcal{J}_{\mathsf{add}} = \Big\langle (0, \boldsymbol{\lambda} x.(0, \boldsymbol{\lambda} y.(y^{\mathsf{s}}, \mathtt{u}))) , \boldsymbol{\lambda} xy.x + y \Big\rangle .$$

Notice that the (highlighted) cost component of $\mathcal{J}_{\mathsf{add}}$ suggest a linear cost measure for computing with add. We also set the intermediate numeric components in the cost tuple to zero. The reason for this choice is that in a cost tuple $\mathcal{C}_\sigma = \mathbb{N} \times \mathcal{F}^{\mathsf{c}}_\sigma$, the numeric component

$\mathbb{N}$ captures the cost of partially applying terms, which is 0 in this case. Using the shorthand notation of Example 7), we could alternatively write $\mathcal{J}_{\mathsf{add}} = \langle (x^{\mathsf{s}}, y^{\mathsf{s}}) \mapsto y^{\mathsf{s}},\ \boldsymbol{\lambda} x^{\mathsf{s}} y^{\mathsf{s}}.x^{\mathsf{s}} + y^{\mathsf{s}} \rangle$.

Now, consider the partially applied term $s = \mathsf{add}\,(\mathsf{add}\,2\,3)$ (of type $\mathsf{nat} \Rightarrow \mathsf{nat}$). Intuitively, the cost of reducing this term to normal form, is the cost of reducing the subterm $\mathsf{add}\,2\,3$ to 5, since the partially applied term $\mathsf{add}\,5$ cannot be reduced. Hence, $\mathtt{dh}_{\mathbb{R}}(s) = 4$. This is also the bound we find through interpretation:

$$\begin{aligned}
[\![s]\!] &= [\![\mathsf{add}]\!] \cdot ([\![\mathsf{add}]\!] \cdot [\![2]\!] \cdot [\![3]\!]) \\
&= [\![\mathsf{add}]\!] \cdot \langle (4, \mathtt{u}), 7 \rangle \\
&= \Big\langle \ (4, \boldsymbol{\lambda} y.(y^{\mathsf{s}}, \mathtt{u})) \ , \boldsymbol{\lambda} y.7 + y \Big\rangle .
\end{aligned}$$

While in this case the bound we find is tight, this is not always the case; for instance $[\![\mathsf{add}\ 0\ (\mathsf{add}\ 0\ 0)]\!] = \langle (3, \mathtt{u}), 3 \rangle$, even though $\mathtt{dh}_{\mathbb{R}}(\mathsf{add}\ 0\ (\mathsf{add}\ 0\ 0)) = 2$. We could obtain a tight bound by choosing a different interpretation, but this is also not always possible.

▶ Remark 38. Intuitively, we think of the numeric component of a partially applied term $\mathsf{f}\, s_1 \dots s_n$ that cannot be reduced at the root as the "environment cost" of computing functional arguments to values. This plays an important role in the complexity analysis in our setting. Namely, when interpreting terms this is what allows us to limit interest to value substitutions, since the cost of reducing arguments to values is captured implicitly by the $\cdot$ operator. This assumption consequently allows us to limit the class of cost functions to *weakly* monotonic functions as used in Definition 14, as opposed to the *strongly* monotonic functionals used in the full rewriting setting [21, 29].

In complexity analysis of term rewriting, it is common to consider bounds on the derivation height for terms of a given size. However, it is useful to impose some limitations. Consider for example a TRS consisting *only* of the two add rules. Then, we might construct a term $(\lambda x.\mathsf{add}\,x\,x)\,((\lambda x.\mathsf{add}\,x\,x)\,(\dots(\mathsf{s}\ 0)\dots))$, with n occurrences of $(\lambda x.\mathsf{add}\,x\,x)$. The size of this term is linear in n, but its derivation height is exponential, since each contraction of a λ essentially duplicates the number of s occurrences. Hence, the traditional notion of *derivational complexity* (which maps a natural number n to the largest derivation height a term of size n can have) is arguably not so useful in a setting with λ.

Instead, we will consider the *runtime complexity* of a TRS. Following the definition in [21] for *full* higher-order runtime complexity, we define:

▶ **Definition 39.** A *data constructor* is a constructor with a type $\iota_1 \Rightarrow \dots \Rightarrow \iota_m \Rightarrow \kappa$, with κ and all ι_i base types.

A *data term* is a value of the form $\mathsf{c}\, d_1 \, \dots \, d_m$ with $\mathsf{c} : \iota_1 \Rightarrow \dots \Rightarrow \iota_m \Rightarrow \kappa$ a data constructor, and each d_i a data term; that is, it is a value without any higher-order subterm.

A *basic term* is a base-type term of the form $\mathsf{f}\, d_1 \, \dots \, d_m$ with $\mathsf{f} \in \Sigma^{\mathsf{def}}$ a defined symbol and all d_i data terms.

The *weak call-by-value runtime complexity* of a TRS is the function $n \mapsto rc(n)$ that maps each natural number n to the largest number h with $\mathtt{dh}_{\mathbb{R}}(s) = h$ for some term s of size n.

Note that for instance lists of functions are not data terms, and therefore not considered as viable inputs in the notion of runtime complexity. As discussed in [21] this arguably makes the notion somewhat first-order, but it can still be used to analyse higher-order programs or modules (so long as they, for instance, have a rule $\mathsf{start}\ x \to r$ where x has base type, and r is allowed to use abstractions, partial application or calls to higher-order functions).

▶ **Example 40.** Let us collect the interpretation for dbl and mult from Example 2.

$$\mathcal{J}_{\mathsf{dbl}} = \Big\langle\ (0, \boldsymbol{\lambda} x.(x^{\mathsf{s}}, \mathtt{u}))\ , \boldsymbol{\lambda} x.2x \Big\rangle$$
$$\mathcal{J}_{\mathsf{mult}} = \Big\langle\ (0, \boldsymbol{\lambda} x.(0, \boldsymbol{\lambda} y.2x^{\mathsf{s}}y^{\mathsf{s}}, \mathtt{u}))\ , \boldsymbol{\lambda} xy.xy \Big\rangle$$

In the TRS of Example 2, the only basic terms have the form $\mathsf{add}\, v_1\, v_2$ or $\mathsf{dbl}\, v$ or $\mathsf{mult}\, v_1\, v_2$. Since $[\![\mathsf{s}^n\ 0]\!]^{\mathsf{s}} = n + 1$, Lemma 37 allows us to conclude that $rc(n) < n^2$.

Now, the size bound for data constructors introduced in Example 23 is well-behaved. However, suppose we had defined $\mathcal{J}_0 = \langle (0, \mathtt{u}), 1\rangle$ and $\mathcal{J}_{\mathsf{s}} = \langle (0, \boldsymbol{\lambda} x.(0, \mathtt{u})), \boldsymbol{\lambda} x.2x + 1\rangle$. In this case, for a data term $\mathsf{n} = \mathsf{s}^n\, 0$, we would have $[\![\mathsf{n}]\!]^{\mathsf{s}} = 2^n + n \geq 2^n$. As a result, we would only be able to derive exponential runtime complexity. Notice that this choice *is* compatible with $\mathbb{R}_{\mathsf{add}}$, and hence proves its termination; however, it induces an exponential overhead on the cost tuple of add, whose actual runtime complexity is linearly bounded as we saw in Example 40. Such a huge overestimation is not desirable in a complexity analysis setting. This behavior suggests an upper bound to the interpretation of data constructors; namely, we seek to bound the constructor's size interpretations *additively*.

Let c be a data constructor of type $\sigma = \iota_1 \Rightarrow \ldots \Rightarrow \iota_m \Rightarrow \kappa$. The size component of $(\!|\sigma|\!)$ is $\mathcal{S}_\sigma = \mathbb{N}^{K(\iota_1)} \Longrightarrow \ldots \Longrightarrow \mathbb{N}^{K(\iota_m)} \Longrightarrow \mathbb{N}^{K(\kappa)}$. The size tuple $\mathcal{J}^{\mathsf{s}}_{\mathsf{c}}$ when fully applied can be written in terms of its functional components. Hence, $\mathcal{J}^{\mathsf{s}}_{\mathsf{c}}(x_1, \ldots, x_m) = \Big\langle f^{\mathsf{s}}_1(x_1, \ldots, x_m), \ldots, f^{\mathsf{s}}_{K(\kappa)}(x_1, \ldots, x_m)\Big\rangle$.

▶ **Definition 41.** If $\mathsf{c} : \sigma$ is a data constructor as above, we say $\mathcal{J}^{\mathsf{s}}_{\mathsf{c}}$ is **additive** if there is a constant $a \in \mathbb{N}$ such that $\sum_{l=1}^{K(\kappa)} f^{\mathsf{s}}_l(x_1, \ldots, x_m) \leq a + \sum_{i=1}^{m} \sum_{j=1}^{K(\iota_i)} x_{ij}$.

It is easy to show that size components for nat and list in Example 23 are additive.

If data constructors are additive, and there are only finitely many of them, then there exists a constant a such that, for every data term d of size n: $[\![d]\!]^{\mathsf{s}} \leq an$. Hence, for instance the following result from [21] also extends to our setting:

▶ **Lemma 42** (From [21]; Corollary 33). Let $\mathbb{R}$ be a TRS. If all interpretations for data constructors are additive and the interpretations for all defined symbols are polynomially bounded, then the weak call-by-value runtime complexity of $\mathbb{R}$ is polynomially bounded.

This result provides us with a systematic approach to establishing bounds to the runtime complexity of weak call-by-value systems. The difficulty now lies in developing techniques to find suitable interpretation shapes. For instance, a first example of a higher-order function over lists is that of map. We studied the structure of its cost–size tuples in Example 17 to illustrate semantical application. We give a concrete cost–size interpretation for map below:

$$\mathcal{J}_{\mathsf{map}} = \Big\langle\ (0, \boldsymbol{\lambda} F.(0, \boldsymbol{\lambda} q.(q_{\mathsf{l}} + F^{\mathsf{c}}(\mathtt{u}, q_{\mathsf{m}})q_{\mathsf{l}} + 1, \mathtt{u})))\ , \boldsymbol{\lambda} Fq.(q_{\mathsf{l}}, F(q_{\mathsf{m}}))\Big\rangle,$$

The highlighted cost component accounts for q_{l} possible β steps, the cost of applying the higher-order argument F over the list q is bounded by $F^{\mathsf{c}}(\mathtt{u}, q_{\mathsf{m}})q_{\mathsf{l}}$ since F^{c} is assumed to be weakly monotonic, and the unitary component is for dealing with the empty list case.

Finding such interpretations for higher-order systems can become quite challenging. In the example below we collect basic weakly monotonic combinators in order to generate more complex cost/size interpretations.

▶ **Example 43.** We list the following weakly monotonic combinators. Here, sets X, Y, Z are used generically to denote cost/size sets:

- for any X and $a \in Y$, there is a constant functional $\boldsymbol{\lambda} x.a$ in $X \Longrightarrow Y$;
- for $f : X \Longrightarrow Y$ and $g : Y \Longrightarrow Z$, we write $g \circ f : X \Longrightarrow Z$ as the composition of f and g.
- the projection function on the ith coordinate, $\pi_i : X_1 \times \cdots \times X_k \Longrightarrow X_i$;
- given $f : X \Longrightarrow Y$ and $g : X \Longrightarrow Z$, we have a function $\langle f, g\rangle : X \Longrightarrow Y \times Z$ which is defined by $\langle f, g\rangle (x) = \langle f(x), g(x)\rangle$;
- given $f : Y \times X \Longrightarrow Z$, we get a function $\boldsymbol{\lambda}_f : X \Longrightarrow (Y \Longrightarrow Z)$. For each $x \in X$ and $y \in Y$, we define $(\boldsymbol{\lambda}_f(x))(y) = f(y, x)$;
- given $f : X \Longrightarrow (Y \Longrightarrow Z)$ and $g : X \Longrightarrow Y$, we obtain $f \mathbin{.} g : X \Longrightarrow Z$, which is defined as $(f \mathbin{.} g)(x) = f(x)(g(x))$;
- given $f : X \Longrightarrow Y$ and $x \in X$, we have an element application functional with domain $\boldsymbol{app}_x : (X \Longrightarrow Y) \Longrightarrow Y$ which sends f to $f(x)$, i.e., $\boldsymbol{app}_x(f) = f(x)$.

Notice that we can use the combinators above with the usual monotonic functionals and operators over $\mathbb{N}$ to produce new monotonic functionals and pointwise operators over sets $X \Longrightarrow Y$. For instance, we can utilize $+, *, \lfloor\cdot\rfloor, \max, \log(\lfloor\cdot\rfloor)$, and so forth.

These basic weakly monotonic functions provide the building blocks for constructing cost–size interpretations.

▶ **Example 44.** The higher-order functions in Example 1 admit the following interpretations:

$$\begin{aligned}
\mathcal{J}_{\mathsf{app}} &= \Big\langle (0, \boldsymbol{\lambda} F.(2, \boldsymbol{\lambda} x.(F^{\mathsf{c}}(\mathtt{u}, x^{\mathsf{s}}), \mathtt{u}))) \ , \boldsymbol{\lambda} F x.F(x) \Big\rangle \\
\mathcal{J}_{\mathsf{comp}} &= \Big\langle (0, \boldsymbol{\lambda} F.(0, \boldsymbol{\lambda} G.(2, \boldsymbol{\lambda} x.(F^{\mathsf{c}}(\mathtt{u}, G^{\mathsf{s}}(x^{\mathsf{s}})) + G^{\mathsf{c}}(\mathtt{u}, x^{\mathsf{s}}), \mathtt{u})))) \ , \boldsymbol{\lambda} F G x.F(G(x)) \Big\rangle \\
\mathcal{J}_{\mathsf{rec}} &= \Big\langle (0, \boldsymbol{\lambda} x.(0, \boldsymbol{\lambda} y.(0, \boldsymbol{\lambda} F.(x^{\mathsf{s}} + H^{\mathsf{c}}(x, y, F), \mathtt{u})))) \ , \boldsymbol{\lambda} x y F.H^{\mathsf{s}}(x, y, F) \Big\rangle
\end{aligned}$$

In the cost component for $\mathcal{J}_{\mathsf{rec}}$, the term x^{s} computes the total number of rewriting steps using the rec symbol. Meanwhile, H^{c} is an auxiliary symbol computing the total cost of recursively applying the higher-order argument F. It can be defined as follows

$$H^{\mathsf{c}}(x, y, F) = \sum_{i=1}^{x^{\mathsf{s}}-1} \pi_1(F^{\mathsf{c}}((\mathtt{u}, i), (\mathtt{u}, H^{\mathsf{s}}(i, y^{\mathsf{s}}, F^{\mathsf{s}}))))$$

with the size helper function H^{s} given as a weakly monotonic variant of the recursor over $\mathbb{N}$:

$$H^{\mathsf{s}}(x, y, F) = \begin{cases} y & \text{if } x \leq 1 \\ \max(y, F(x-1, H^{\mathsf{s}}(x-1, y, F))) & \text{if } x > 1 \end{cases}$$

7 Conclusions and Future Work

In this paper we introduced an interpretation method for higher-order rewriting with weak call-by-value reduction. In this approach, we build on existing work defining tuple interpretations [21, 33], but restrict the evaluation strategy, and define a cost–size semantics for types and terms which generates a whole new class of cost–size termination models that can be used to reason about both termination and complexity of weak call-by-value systems. We showed that cost–size tuples correctly capture call-by-value termination and allow us to bound both the cost (number of steps to reach normal forms) and a variety of size notions for different data types. A second advantage of our approach compared to [21] is that the cost functionals are now weakly rather than strongly monotonic functionals, which simplifies the search for cost interpretations.

This is foundational work in the research direction of transposing the methodology and tools from (higher-order) term rewriting to program analysis. A first step for future work is to consider more expressive type theories, so we can capture more programs. For instance, the power of the techniques developed here would be greatly improved if polymorphic types are taken into account. A second step is to expand other complexity methods for innermost/call-by-value rewriting to the higher-order setting, such as dependency tuples [27] or polynomial path orders [3]. Also for termination analysis, it would be interesting to combine tuple interpretations with a higher-order variant of innermost dependency pairs [28], similar to what was done for full rewriting with tuple interpretations in [22].

Finally, we plan to implement this work, to automatically derive bounds to the derivation height of individual terms, as well as provide bounds for both full and call-by-value runtime complexity of higher-order term rewriting systems. The automation approach could build on the strategy for higher-order polynomial interpretations for full rewriting (not using tuples) in [14, Section 5]. While the search for tuple interpretations has more unknowns than the search for interpretations to ℕ, and will therefore likely take longer, we expect that the overall methodology can stay largely unchanged at least when it comes to an unrestricted evaluation strategy. Adapting to weak call-by-value rewriting may require some additional study, however.

References

1 T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *TCS*, 2000. doi:10.1016/S0304-3975(99)00207-8.

2 M. Avanzini and U. Dal Lago. Automating sized-type inference for complexity analysis. In *Proc. ICFP*, 2017. doi:10.1145/3110287.

3 M. Avanzini and G. Moser. Complexity analysis by rewriting. In *Proc. FLOPS*, 2008. doi:10.1007/978-3-540-78969-7_11.

4 P. Baillot and U. Dal Lago. Higher-order interpretations and program complexity. *IC*, 2016. doi:10.1016/j.ic.2015.12.008.

5 H. Barendregt, W. Dekkers, and R. Statman. *Lambda Calculus with Types*. Perspectives in Logic. Cambridge University Press, 2013. doi:10.1017/CBO9781139032636.

6 A. Ben Cherifa and P. Lescanne. Termination of rewriting systems by polynomial interpretations and its implementation. *Science of Computer Programming*, 9(2):137–159, 1987. doi:https://doi.org/10.1016/0167-6423(87)90030-X.

7 G. Bonfante, A. Cichon, J.-Y. Marion, and H. Touzet. Algorithms with polynomial interpretation termination proof. *Journal of Functional Programming*, 11(1):33–53, 2001. doi:10.1017/S0956796800003877.

8 G. Bonfante, J. Marion, and J. Moyen. On lexicographic termination ordering with space bound certifications. In *Proc. PSI*, 2001. doi:10.1007/3-540-45575-2_46.

9 A. Cichon and P. Lescanne. Polynomial interpretations and the complexity of algorithms. In *CADE*, pages 139–147, 1992. doi:10.1007/3-540-55602-8_161.

10 M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and J.F. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243, 2002. Rewriting Logic and its Applications. doi:https://doi.org/10.1016/S0304-3975(01)00359-0.

11 M. Codish, I. Gonopolskiy, A. M. Ben-Amram, C. Fuhs, and J. Giesl. Sat-based termination analysis using monotonicity constraints over the integers. *Theory and Practice of Logic Programming*, 11(4-5):503–520, 2011. doi:https://doi.org10.1017/S1471068411000147.

12 E. Contejan, C. Marché, A. P. Tomás, and X. Urbain. Mechanically proving termination using polynomial interpretations. *JAR*, 34, 2005. doi:10.1007/s10817-005-9022-x.

13 N. Danner, D.R. Licata, and R. Ramyaa. Denotational cost semantics for functional languages with inductive types. In *Proc. ICFP*, 2015. doi:10.1145/2784731.2784749.

14 C. Fuhs and C. Kop. Polynomial interpretations for higher-order rewriting. In *Proc. RTA*, 2012. doi:10.4230/LIPIcs.RTA.2012.176.

15 Jürgen Giesl, Christoph Walther, and Jürgen Brauburger. Termination analysis for functional programs. In Wolfgang Bibel and Peter H. Schmitt, editors, *Automated Deduction — A Basis for Applications: Volume III Applications*, pages 135–164. Springer Netherlands, Dordrecht, 1998. doi:10.1007/978-94-017-0437-3_6.

16 N. Hirokawa and G. Moser. Automated complexity analysis based on the dependency pair method. In *Proc. IJCAR*, 2008. doi:10.1007/978-3-540-71070-7_32.

17 D. Hofbauer. Termination proofs by multiset path orderings imply primitive recursive derivation lengths. *TCS*, 1992. doi:10.1007/3-540-53162-9_50.

18 D. Hofbauer. Termination proofs by context-dependent interpretations. In *Proc. RTA*, 2001. doi:10.1007/3-540-45127-7_10.

19 D. Hofbauer and C. Lautemann. Termination proofs and the length of derivations. In *Proc. RTA*, 1989. doi:10.1007/3-540-51081-8_107.

20 D. M. Kahn and J. Hoffmann. Exponential automatic amortized resource analysis. In *Proc. FoSSaCS*, 2020. doi:10.1007/978-3-030-45231-5_19.

21 C. Kop and D. Vale. Tuple interpretations for higher-order complexity. In *Proc. FSCD*, 2021. doi:10.4230/LIPIcs.FSCD.2021.31.

22 Cynthia Kop. Cutting a Proof into Bite-Sized Chunks: Incrementally proving termination in higher-order term rewriting. In *Proc. FSCD22*, pages 1:1–1:17, 2022. doi:10.4230/LIPIcs.FSCD.2022.1.

23 F. Mitterwallner and A. Middeldorp. Polynomial Termination Over $\mathbb{N}$ Is Undecidable. In *Proc. FSCD*, pages 27:1–27:17, 2022. doi:https://doi.org/10.4230/LIPIcs.FSCD.2022.27.

24 G. Moser. Uniform Resource Analysis by Rewriting: Strengths and Weaknesses (Invited Talk). In *Proc. FSCD*, pages 2:1–2:10, 2017. doi:10.4230/LIPIcs.FSCD.2017.2.

25 G. Moser, A. Schnabl, and J. Waldmann. Complexity Analysis of Term Rewriting Based on Matrix and Context Dependent Interpretations. In *Proc. IARCS*, pages 304–315, 2008. doi:10.4230/LIPIcs.FSTTCS.2008.1762.

26 Y. Niu and J. Hoffmann. Automatic space bound analysis for functional programs with garbage collection. In *Proc. LPAR*, 2018. doi:10.29007/xkwx.

27 L. Noschinski, F. Emmes, and J. Giesel. Analysing innermost runtime complexity of term rewriting by dependency pairs. *JAR*, 2013. doi:10.1007/s10817-013-9277-6.

28 L. Noschinski, F. Emmes, and J. Giesl. A dependency pair framework for innermost complexity analysis of term rewrite systems. In *CADE-23*, pages 422–438, 2011. doi:10.1007/978-3-642-22438-6_32.

29 J.C. 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.

30 Yoshihito T. Counterexamples to termination for the direct sum of term rewriting systems. *Information Processing Letters*, 25(3):141–143, 1987. doi:https://doi.org/10.1016/0020-0190(87)90122-0.

31 A. Weiermann. Termination proofs for term rewriting systems by lexicographic path orderings imply multiply recursive derivation lengths. *TCS*, 1995. doi:10.1016/0304-3975(94)00135-6.

32 Wiki. The International Termination Competition (TermComp). http://termination-portal.org/wiki/Termination_Competition, 2018.

33 A. Yamada. Multi-dimensional interpretations for termination of term rewriting. In *In. Proc. CADE28*, volume 12699 of *Lecture Notes in Computer Science*, pages 273–290, 2021. doi:10.1007/978-3-030-79876-5_16.