Complexity Analysis 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

In this short 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 limn a cost–size semantics to call-by-value rewriting: a class of algebraic interpretations to map terms to tuples which that bounds both the reduction's 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

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

This short paper is a brief exposition of the conference paper "Cost–Size Semantics for Call-by-Value Higher-Order Rewriting" recently accepted for publication at FSCD 2023. 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 by adapting techniques for proving termination to deduce the complexity. There is a myriad of works following this idea. To mention a few, see [2, 3, 5, 10, 11, 14] for interpretation methods, [4, 9, 18] for lexicographic and path orders, and [8, 16] for dependency pairs.

However, those ideas are focused on *first-order* term rewriting. The literature on complexity of *higher-order* rewriting is scarce. While there is a lot of work on complexity of functional programs [1, 6, 12, 15], 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 [13], we introduced an extension of the method of *weakly monotonic algebras* [7, 17] 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 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 [1, 6]. This is a popular idea, as a very similar approach was introduced for first-order rewriting around the same time [19].

2 Preliminaries

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.

Let $\mathbb{B}$ be a nonempty set of *base types*. The set $\mathbb{T}_\mathbb{B}$ of *simple types* over $\mathbb{B}$ is generated by the grammar: $\mathbb{T}_\mathbb{B} := \mathbb{B} \mid \mathbb{T}_\mathbb{B} \Rightarrow \mathbb{T}_\mathbb{B}$. As usual, we assume that the $\Rightarrow$ type constructor is right-associative. A *signature* $\mathbb{F}$ is a triple $(\mathbb{B}, \Sigma, \mathtt{ar})$ where $\mathbb{B}$ is a set of base types, Σ is a nonempty finite set of symbols, and $\mathtt{ar}$ is a function $\mathtt{ar} : \Sigma \longrightarrow \mathbb{T}_\mathbb{B}$. We postulate, for each type σ, 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 $\sigma \neq \tau$ and let $\mathbb{X}$ denote the family of sets.

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:

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

We assume the usual λ-Calculus association and precedence scheme for application and abstraction. We shall remove unnecessary parentheses and write terms following those rules. Application of substitutions is defined as expected.

Call-by-Value Higher-order Rewriting A *rewrite rule* $\ell \to r$ is a pair of terms of the same type such that $\ell = \mathsf{f}\, \ell_1 \ldots \ell_k$ and $\mathtt{fv}(r) \subseteq \mathtt{fv}(\ell)$. A *term rewriting system* (TRS) $\mathbb{R}$ is a set of rules. In this paper, we are interested in a restricted evaluation strategy, which limits reduction to terms whose immediate subterms are *values*:

▶ **Definition 1.** A term s is a *value* whenever s is:

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

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 $\mathsf{c}\, s_1 \ldots s_n$ with $n \geq 0$, where each s_i is a ground constructor term.

Notice that by definition ground constructor terms are values since there is no rule $\mathsf{c}\, \ell_1\ \ldots\ \ell_k \to r$ 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 $[\mathsf{add}\, \mathsf{0}; \lambda x.x; \mathsf{mult}\, \mathsf{0}; \mathsf{dbl}]$.

▶ **Definition 2.** The **higher-order weak call-by-value rewrite relation** $\to_v$ induced by $\mathbb{R}$ is defined as follows:

- $\mathsf{f}\, (\ell_1\gamma) \ldots (\ell_k\gamma) \to_v r\gamma$, if $\mathsf{f}\, \ell_1 \ldots \ell_k \to r \in \mathbb{R}$ 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 $t \to_v t'$.

▶ **Example 3.** Let us consider two simple examples of functions encoded as rules. The first is map, which applies a function $F : \mathsf{nat} \Rightarrow \mathsf{nat}$.

$$\begin{aligned} &\mathsf{map}\, F\, \mathsf{nil} \to \mathsf{nil} && \mathsf{add}\, x\, \mathsf{0} \to \mathsf{0} \\ &\mathsf{map}\, F\, (\mathsf{cons}\, x\, xs) \to \mathsf{cons}\, (F\, x)\, (\mathsf{map}\, F\, xs) && \mathsf{add}\, x\, (\mathsf{s}\, y) \to \mathsf{s}\, (\mathsf{add}\, x\, y) \end{aligned}$$

3 Cost–Size Semantics for Types and Terms

The kernel of the interpretation of types 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.

▶ **Definition 4** (Interpretation of Types)**.** 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:

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

The set $(\!|\sigma|\!)$ is ordered component-wise. With that this interpretation of types is well-founded, which was proved in the full version of this paper. Next, we need an *application operator* for applying cost–size tuples. More precisely, 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 $\boldsymbol{x}$ as follows.

▶ **Definition 5.** 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:

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

An interpretation of a signature $\mathbb{F} = (\mathbb{B}, \Sigma, \texttt{ar})$ interprets the base types in $\mathbb{B}$ and each $\mathsf{f} \in \Sigma$ of arity $\texttt{ar}(\mathsf{f}) = \sigma$ as an element of $(\!|\sigma|\!)$ which is constructed by Definition 4.

▶ **Definition 6.** A **cost–size tuple interpretation** $\mathcal{F}$ for a signature $\mathbb{F} = (\mathbb{B}, \Sigma, \texttt{ar})$ consists of a pair of functions $(\mathcal{J}_\mathbb{B}, \mathcal{J}_\Sigma)$ where

- $\mathcal{J}_\mathbb{B}$ is a type interpretation key, which maps each base type ι to a size tuple $\mathbb{N}^{K(\iota)}$
- $\mathcal{J}_\Sigma$ is an *interpretation of symbols* in Σ which maps each $\mathsf{f} \in \Sigma$ with $\texttt{ar}(\mathsf{f}) = \sigma$ to a cost–size tuple in $(\!|\sigma|\!)$, where $(\!|\sigma|\!)$ is built using $\mathcal{J}_\mathbb{B}$ in Definition 4.

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 7.** As a first example of interpretation, let us interpret the data constructors from Example 3. Recall that $\mathsf{0} : \mathsf{nat}$, $\mathsf{s} : \mathsf{nat} \Rightarrow \mathsf{nat}$ are the constructors for nat and $\mathcal{J}_\mathbb{B}(\mathsf{nat}) = \mathbb{N}$.

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

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. Intuitively, 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 $\mathcal{J}_\mathbb{B}(\mathsf{list}) = \mathbb{N}^2$. 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.

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

The highlighted cost components are filled with zeroes for lists as well. Size components are interpreted following the semantics we set for the two size components lenght and maximum element size, respectively.

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 8.** 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 $F :: \sigma \Rightarrow \tau$.

▶ **Definition 9.** 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 $\boldsymbol{\lambda} x_1.(0, \boldsymbol{\lambda} x_2 \ldots (0, \mathtt{u}) \ldots)$. 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 10.** We continue with Example 7 by interpreting ground constructor terms fully. A ground constructor term d of type nat is of the form $\mathsf{s}\,(\mathsf{s} \ldots (\mathsf{s}\,\mathsf{0}) \ldots)$ 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 $[\mathsf{n}_1; \ldots; \mathsf{n}_k]$ for the term $\mathsf{cons}\,\mathsf{n}_1 \ldots (\mathsf{cons}\,\mathsf{n}_k\,\mathsf{nil})$. The empty list constructor nil is written as $[]$ in this notation. Hence, the cost–size interpretation of $\mathsf{3} : \mathsf{nat}$ is given by:

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

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

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

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

We give a concrete cost–size interpretation for map and add below:

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

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

4 Complexity Analysis of Call-by-Value Rewriting

Since our analysis is quantitative, our goal is not merely to find tuple interpretations that prove termination but also ones that provide "good" upper bounds on the complexity of reducing terms. To start, we will extend the notion of *derivation height* to our setting:

▶ **Definition 11.** 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. The methodology of weakly monotonic algebras offers a systematic way to derive bounds for the derivation height of a given term:

▶ **Lemma 12.** If $[\![s]\!] = \langle (n, F^{\mathsf{c}}), F^{\mathsf{s}} \rangle$, then $\mathtt{dh}_{\mathbb{R}}(s) \leq n$.

As an illustration of how this is used, let us complete the interpretation of Example 3. We start with the system $\mathbb{R}_{\mathsf{add}}$. We will use the type and constructor interpretations as given in Example 7. The rules in $\mathbb{R}_{\mathsf{add}}$ suggest the following cost–size interpretation:

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

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.

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:

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

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

With this observation, we get a framework that provides us with a systematic approach to establish bounds to the 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 give a concrete cost–size interpretation for map below:

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

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.

5 Conclusions

In this short paper we briefly discussed an interpretation method for higher-order rewriting with weak call-by-value reduction. In this approach, we build on existing work defining tuple interpretations [13, 19], but restrict the evaluation strategy, and define a cost–size semantics for types and terms which generate a whole new class of cost–size semantic techniques that can be used to reason about the complexity of weak call-by-value systems.

References

1. M. Avanzini and U. Dal Lago. Automating sized-type inference for complexity analysis. In *Proc. ICFP*, 2017. doi:10.1145/3110287.
2. P. Baillot and U. Dal Lago. Higher-order interpretations and program complexity. *IC*, 2016. doi:10.1016/j.ic.2015.12.008.
3. 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.
4. 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.
5. 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.
6. 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.
7. C. Fuhs and C. Kop. Polynomial interpretations for higher-order rewriting. In *Proc. RTA*, 2012. doi:10.4230/LIPIcs.RTA.2012.176.
8. 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.
9. D. Hofbauer. Termination proofs by multiset path orderings imply primitive recursive derivation lengths. *TCS*, 1992. doi:10.1007/3-540-53162-9_50.
10. D. Hofbauer. Termination proofs by context-dependent interpretations. In *Proc. RTA*, 2001. doi:10.1007/3-540-45127-7_10.
11. D. Hofbauer and C. Lautemann. Termination proofs and the length of derivations. In *Proc. RTA*, 1989. doi:10.1007/3-540-51081-8_107.
12. D. M. Kahn and J. Hoffmann. Exponential automatic amortized resource analysis. In *Proc. FoSSaCS*, 2020. doi:10.1007/978-3-030-45231-5_19.
13. C. Kop and D. Vale. Tuple interpretations for higher-order complexity. In *Proc. FSCD*, 2021. doi:10.4230/LIPIcs.FSCD.2021.31.
14. 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.
15. Y. Niu and J. Hoffmann. Automatic space bound analysis for functional programs with garbage collection. In *Proc. LPAR*, 2018. doi:10.29007/xkwx.
16. 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.
17. 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.
18. 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.
19. 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.