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

Cynthia Kop /Envelope/home Institute for Computing and Information Sciences, Radboud University, Nijmegen, The Netherlands
Deivid Vale /Envelope/home 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.FSCD.2023.15

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 $s \rightarrow t$ 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 [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 $\lambda$-binder is also not usually allowed. The difference can be substantial: for instance for a pair of rules $f \ x \ 0 \rightarrow x$, $f \ x \ (s \ y) \rightarrow f \ (\text{pair} \ x \ x) \ y$, 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 $\beta$-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 $\iota, \kappa, \nu$. The set $T_\mathbb{B}$ of simple types over $\mathbb{B}$ is generated by the grammar:

$$T_\mathbb{B} := \mathbb{B} | T_\mathbb{B} \to T_\mathbb{B}.$$  

Types from $T_\mathbb{B}$ range over $\sigma, \tau, \rho$. The $\to$ type constructor is right-associative, so we write $\sigma \Rightarrow \tau \Rightarrow \rho$ for $(\sigma \Rightarrow (\tau \Rightarrow \rho))$. Notice that every simple type $\sigma$ can be written as $\tau_1 \Rightarrow \cdots \Rightarrow \tau_n \Rightarrow \iota$. We informally say that the $\tau_i$'s are the input types and the base type $\iota$ is the output type. We abbreviate such types by $\tau \Rightarrow \iota$. The type order of a type is the number: (a) $\text{ord}(\iota) = 0$ and (b) $\text{ord}(\sigma \Rightarrow \tau) = \text{max}(1 + \text{ord}(\sigma), \text{ord}(\tau))$. A signature $\mathbb{F}$ is a triple $(\mathbb{B}, \Sigma, \alpha)$ where $\mathbb{B}$ is a set of base types, $\Sigma$ is a nonempty finite set of symbols, and $\alpha$ is a function $\alpha : \Sigma \to T_\mathbb{B}$. For each type $\sigma$, we postulate the existence of a nonempty set $X_\sigma$ of countably many variables. Furthermore, we impose that $X_\sigma \cap X_\tau = \emptyset$ whenever $\sigma \neq \tau$. Let $X$ denote the family of sets $(X_\sigma)_{\sigma \in T_\mathbb{B}}$ indexed by $T_\mathbb{B}$ and assume that $\Sigma \cap X = \emptyset$.

The set $T(\mathbb{F}, X)$ of terms built from $\mathbb{F}$ and $X$ collects those expressions $s$ for which the judgment $s : \sigma$ can be deduced using the following rules:

$$
\begin{align*}
\frac{x \in X_\sigma}{x : \sigma} & \quad \frac{f \in \Sigma}{f : \sigma} & \quad \frac{s : \sigma \Rightarrow \tau}{t : \sigma} & \quad \frac{(st) : \tau}{(\lambda x. s) : \sigma \Rightarrow \tau}.
\end{align*}
$$

Application of terms is left-associative, so we write $s t u$ for $((s t) u)$. Abstraction is right-associative, so we write $\lambda x y z. s$ for $\lambda y. (\lambda z. s)$. 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 $\mathbb{F}v(s)$ of free variables occurring in $s$ is defined as expected. A term $s$ is closed if $\mathbb{F}v(s) = \emptyset$. It is ground if no variable occurs in it. A symbol $f \in \Sigma$ is called the head symbol of $s$ if $s = f s_1 \ldots s_k$. A subterm of $s$ is a term $t$ (we write $s \supseteq t$) 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 $\gamma$ is a type-preserving map from variables to terms such that the set $\text{dom}(\gamma) = \{ x \in X \mid \gamma(x) \neq x \}$ is finite. We may explicitly represent $\gamma$ as a list of mappings $[x_1 := s_1, \ldots, x_k := s_k]$. The capture avoiding application of $\gamma$ to $s$ is defined as follows:

$$
\begin{align*}
x\gamma &= \gamma(x) & (st)\gamma &= (s\gamma)(t\gamma) \\
f\gamma &= f & (\lambda x. s)\gamma &= \lambda y. (s[x\mapsto y]\gamma), \text{ for } y \text{ fresh}
\end{align*}
$$

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 $\alpha$-renaming. We identify terms modulo $\alpha$-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 $f(s) \cdot t$. 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 $\beta$-normal form.
Higher-Order Rewriting. A rewrite rule \( \ell \rightarrow r \) is a pair of terms of the same type such that \( \ell = f \ell_1 \ldots \ell_k \) and \( \forall v(\ell) \subseteq \forall v(f) \). A term rewriting system (TRS)\(^2\) \( \mathcal{R} \) is a set of rules. A relation \( \rightarrow \) on terms is monotonic if \( s \rightarrow s' \) implies \( ts \rightarrow ts' \), \( su \rightarrow s'u \), and \( \lambda x. s \rightarrow \lambda x. s' \); for all terms \( t \) and \( u \) of appropriate types. The rewrite relation \( \rightarrow_{\mathcal{R}} \) induced by \( \mathcal{R} \) is the smallest monotonic relation containing \( \mathcal{R} \) union the \( \beta \) rule-scheme (i.e., \( \lambda x. s \rightarrow_{\beta} s[x := t] \)) and closed under application of substitution. An \( \mathcal{R} \)-reducible expression (redex) is a term of form \( \ell \gamma \) for some rule \( \ell \rightarrow r \) and substitution \( \gamma \). A \( \beta \)-redex is of the form \( \lambda x. s \). \( t \).

Every rewrite rule \( \ell \rightarrow r \) defines a symbol \( f \), namely, the head symbol of \( \ell \). For each \( f \in \Sigma \), let \( \mathcal{R}_f \) denote the set of rewrite rules that define \( f \) in \( \mathcal{R} \). A symbol \( f \in \Sigma \) is a defined symbol if \( \mathcal{R}_f \neq \emptyset \). A constructor symbol is a symbol \( c \in \Sigma \) such that \( \mathcal{R}_f = \emptyset \). We let \( \Sigma_{\text{def}} \) be the set of defined symbols and \( \Sigma_{\text{con}} \) the set of constructor symbols. Hence, \( \Sigma = \Sigma_{\text{def}} \cup \Sigma_{\text{con}} \). A ground constructor term is a term \( c \ell_1 \ldots \ell_n \) with \( n \geq 0 \), where each \( \ell_i \) 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 nat \( \Rightarrow \) nat and lists having type list.

\[
\begin{align*}
\text{map } F \text{ nil} & \rightarrow \text{ nil} & \text{comp } F \text{ G} & \rightarrow \lambda x. F(Gx) \\
\text{map } F(\text{cons } x \text{ x s}) & \rightarrow \text{ cons } (Fx)(\text{map } Fx s) & \text{app } F & \rightarrow \lambda x. Fx \\
\text{rec } 0 y F & \rightarrow y & \text{rec } (s x) y F & \rightarrow Fx(\text{rec } xyF)
\end{align*}
\]

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

\[
\begin{align*}
dbl \text{ 0} & \rightarrow 0 & \text{add } x \text{ 0} & \rightarrow 0 & \text{mult } x \text{ 0} & \rightarrow 0 \\
dbl \text{ (s x)} & \rightarrow \text{ s (dbl x)} & \text{add } x \text{ (s y)} & \rightarrow \text{ s (add x y)} & \text{mult } x \text{ (s y)} & \rightarrow \text{ add x (mult x y)}
\end{align*}
\]

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 \( f v_1 \ldots v_n \), with each \( v_i \) a value and there is no rule \( f \ell_1 \ldots \ell_k \rightarrow r \) 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 \( c \ell_1 \ldots \ell_k \rightarrow r \) for any \( k \) if \( c \in \Sigma_{\text{con}} \). More complex values include partially applied functions and lambda-terms; for example, \( \text{add } 0 \) or a list of functions \( [\text{add } 0; \lambda x. x; \text{mult } 0; \text{dbl}] \). 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 \( \rightarrow_v \) induced by \( \mathcal{R} \) is defined as follows:

- \( f(\ell_1 \gamma) \ldots (\ell_k \gamma) \rightarrow_v r \gamma \), if \( f \ell_1 \ldots \ell_k \rightarrow r \in \mathcal{R} \) and each \( \ell_i \gamma \) is a value;
- \( (\lambda x. s) v \rightarrow_v s[x := v] \), if \( v \) is a value;
- \( s t \rightarrow_v s' t \) if \( s \rightarrow_v s' \); and \( s t \rightarrow_v s' t' \) if \( t \rightarrow_v t' \).

\(^2\) 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 \( s \rightarrow t \) should be read as \( s \rightarrow_v t \). 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 \( s \rightarrow_v t \). A term \( s \) has a normal form if \( s \rightarrow^* s' \) and \( s' \) is in normal form. A TRS \( \mathcal{R} \) is terminating if no infinite rewrite sequence \( s \rightarrow_v s_1 \rightarrow_v \ldots \) 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 \( \{(u), \sqsupseteq\} \), with \( u \sqsupseteq u \).

Given quasi-ordered sets \((A, \sqsupseteq)\) and \((B, \sqsupseteq)\), a function \( f : A \rightarrow B \) is weakly monotonic if \( x \sqsupseteq y \) implies \( f(x) \sqsubseteq f(y) \). Let \( A \Rightarrow B \) denote the set of weakly monotonic functions from \( A \) to \( B \). The comparison operator \( \sqsupseteq \) on \( B \) induces point-wise comparison on \( A \Rightarrow B \) as follows: \( f \sqsupseteq g \) if \( f(x) \sqsubseteq g(x) \) for all \( x \in A \). This way \((A \Rightarrow B, \sqsupseteq)\) is also quasi-ordered. Given well-founded sets \((A, >, \gtrsim)\) and \((B, >, \gtrsim)\), a function \( f : A \rightarrow B \) 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 \( \text{Size}(0) = 1 \) and \( \text{Size}(s \ t) = 1 + \text{Size}(t) \); intuitively, the size of a ground constructor term of type \( \text{nat} \) is the number of function symbols in it. For lists, we could let \( \text{Size}(\text{nil}) = (0, 0) \) and \( \text{Size}(\text{cons} \ s \ t) = (\text{Size}(t) + 1, \max(\text{Size}(s), \text{Size}(t))) \); intuitively, the size of a list of numbers is the pair (list length, size of its greatest element). We could let \( \text{Size}(\text{add} \ s) \) be the function that maps \( n \) to \( \text{Size}(s) + n \), and \( \text{Size}(\text{map}) \) 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 \( \text{map} \) 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 \( \text{add} \ s \ t \) is bounded by the cost of evaluating the arguments, plus \( \text{Size}(s) \) (as we easily see by inspecting the rules defining \( \text{add} \)). Hence, we would let \( \text{Cost}(\text{add} \ s \ t) = \text{Cost}(s) + \text{Cost}(t) + \text{Size}(s) \), and could define \( \text{Cost}(\text{add}) = \lambda(c_1, s_1), (c_2, s_2). c_1 + c_2 + s_1 \). Note that the cost function takes a pair of values for each argument: respectively, the cost and size of the argument.
For \(\text{map}\), the number of steps to evaluate \(\text{map}\ s\ t\) depends heavily on \(s\), even if both \(s\) and \(t\) are values: \(\text{map}\ ((\lambda x.\text{add } x\ 0)\ t)\) will take substantially fewer steps than evaluating \(\text{map}\ ((\lambda x.\text{mult } x\ x)\ t)\). Hence, we should take the cost function for \(s\) into account as well as its size. This yields \(\text{Cost}(\text{map}) = \lambda (F_{\text{cost}}, F_{\text{size}}), (q_{\text{cost}}, (l, m))\). \(F_{\text{cost}} + l + 1 + l \ast F_{\text{cost}}(0, m)\):

the number of steps to evaluate \(\text{map}\ s\ t\) is bounded by the cost of evaluating \(t\) first, then applying \(s\) (\text{length of list}) times to the largest element of \(t\), plus the \(1 + \text{length of list}\) steps for the evaluation of \(\text{map}\) itself. Note that since we use a call-by-value strategy, the list \(q\) is evaluated to a value before the \(\text{map}\) rule fires, which is why \(F_{\text{cost}}\) is given a zero argument.

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

Examples 5 and 6 sketch an idea where \(\text{Size}(s) = \text{Size}(s)(\text{Size}(t))\) and \(\text{Cost}(s) = \text{Cost}(s)(\text{Cost}(t), \text{Size}(t))\). 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 \(\text{add} \ (\text{dbl } 0)\), can be, and the cost for the \(\text{dbl } 0\) reduction should be included. Moreover, terms of higher type can also reduce directly even when their subterms are values; e.g., \(\text{comp} \ s\ t\) or \((\lambda x. s)\ t\) of type \(\text{nat} \Rightarrow \text{nat}\).

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 \text{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 \((12, \lambda (x_{\text{cost}}, x_{\text{size}}). x_{\text{cost}} + x_{\text{size}})\), 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 \(t : \sigma\), \(\text{Cost}(s\ t) = (\text{CostNum}(s) + \text{CostNum}(t) + c, \text{fun})\), where \(\text{CostFun}(s)(\text{CostFun}(t), \text{Size}(t))\) is the pair \((c, \text{fun})\).

**Example 7** (Cost pairs). We let \(\text{Cost}(\text{add}) = (0, \lambda (u_1, n). (0, \lambda (u_2, m). n))\): the first 0 is the “cost number” for \(\text{add}\), which is 0 because \(\text{add}\) is in normal form; and the function \(\lambda (u_1, n). (0, \lambda (u_2, m). n)\) takes a unit element and the size of a value, and returns a new pair. With the rough definition of application above, we have \(\text{Cost}(\text{add}\ s) = (\text{CostNum}(s), \lambda (u, n). (\text{Size}(s), u))\). This matches the intuition that the number of steps needed to reduce \(\text{add}\ s\) 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 \(\text{Size}(s)\) steps. We obtain \(\text{Cost}(\text{add}\ s\ t) = \text{Cost}(s) + \text{Cost}(t) + \text{Size}(s)\) as expected.

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

With these definitions, if we can show that \(\text{Cost}(t), \text{Size}(t) \vdash (\text{Cost}(r), \text{Size}(r))\) for all value instances of rules, then \(\text{CostNum}(s)\) 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 \(\text{Size}(\text{nil})\), \(\text{Size}(\text{cons})\) and \(\text{Size}(\text{map})\) following Example 5, and let \(\text{Cost}(\text{nil}) = 0\), \(\text{Cost}(\text{cons}) = (n, m) \Rightarrow 0\) and \(\text{Cost}(\text{map}) = ((F_{\text{cost}}, F_{\text{size}}), (l, m)) \Rightarrow l \ast F_{\text{cost}}(m) + l + 1\). Then, for a list \text{cons} \(h t\) with \(\text{Size}(t) = (l, m)\), we have
To show well-foundedness of we can conclude that it follows immediately from Definition 10 that Lemma 11.

Definition 12 define next the requirements for the sets used for size interpretations. Given a well-founded set \( S \) of \( \text{cost–size} \) the relations naturally induce an order structure on the product \( C \times S \) of \( \text{cost–size} \) cost–size tuples.

Given a cost–size product \( C \times S \), the well-foundedness of \( C \) and quasi-ordering on \( S \) induce an order structure on the product \( C \times S \) as follows.

Definition 10 (Product Order). Let \( (C,\succsim,\succeq) \) be a cost–size product. Then we define the relations \( \succsim,\succeq \) over \( C \times S \) as follows: for all \( (x,y) \) and \( (x',y') \) in \( C \times S \),

- \( (x,y) \succsim (x',y') \) iff \( x \succsim x' \) and \( y \succeq y' \), and
- \( (x,y) \succeq (x',y') \) iff \( x \succeq x' \) and \( y \succeq y' \).

Next, we show that the triple \( (C \times S,\succsim,\succeq) \) is well-founded.

Lemma 11. The triple \( (C \times S,\succsim,\succeq) \) defined in Definition 10 is a well-founded set.

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

Suppose \( (x,y) \succsim (x',y') \). Since \( x \succsim x' \) implies \( x \succeq x' \), we have \( (x,y) \succeq (x',y') \).

Suppose \( (x,y) \succeq (x',y') \). Since \( x \succeq x' \) implies \( x \succeq x'' \) and \( \succeq \) is transitive, we have \( (x,y) \succeq (x'',y'') \).

We shall use product orders to induce well-founded ordering on cost–size sets. Let us define the interpretations 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 \( \mathcal{I}_\mathbb{B} \), is a function that maps each base type \( t \in \mathbb{B} \) to a quasi-ordered set \( (\mathcal{I}_\mathbb{B}(t),\succeq) \) with a minimum element, i.e., it contains an element \( \bot \), such that \( x \succeq \bot \) 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 \( \iota \in \mathbb{B} \), \( J_{\mathbb{B}/\mathbb{N}}(\iota) \) 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 \( J_{\mathbb{B}/\mathbb{N}} \) 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 \( J_\mathbb{B} \) we inductively interpret the elements of \( \mathcal{T}_\mathbb{B} \) as cost–size products.

Definition 14 (Interpretation of Types). Let \( J_\mathbb{B} \) be an interpretation key. We define for each type \( \sigma \) the cost–size tuple interpretation of \( \sigma \) as the set \( (\sigma) = C_\sigma \times S_\sigma \) where \( C_\sigma \) and \( S_\sigma \) are defined as follows (mutually with the set \( \mathcal{F}_\sigma \)):

\[
\begin{align*}
C_\sigma &= \mathbb{N} \times \mathcal{F}_\sigma \\
\mathcal{F}_\iota &= \text{unit} \\
\mathcal{F}_\sigma \Rightarrow \tau &= (\mathcal{F}_\sigma \times S_\sigma) \rightarrow C_\tau
\end{align*}
\]

The set \( (\sigma) \) is ordered as follows:

- \( < (n, f_1), (m, f_2) > < (m, g_1), g_2 > \) if \( n > m \) and \( f_1 \geq g_1 \) and \( f_2 \geq g_2 \), and
- \( < (n, f_1), (m, f_2) > < (m, g_1), g_2 > \) if \( n > m \) and \( f_1 \geq g_1 \) and \( f_2 \geq g_2 \).

We say a function \( f \) is a cost (size) function whenever \( f \in \mathcal{F}_\sigma \) (\( f \in S_\sigma \)), for some type \( \sigma \).

Lemma 15. For any type \( \sigma \), \( (C_\sigma, \geq, \supseteq) \) is well-founded and \( (S_\sigma, \sqsubseteq) \) is quasi-ordered with a minimum. Therefore, \( (\sigma) \) is a cost–size product.

Proof. When \( \sigma \) is a base type, \( C_\sigma = \mathbb{N} \times \text{unit} \cong \mathbb{N} \) and \( S_\sigma = J_\mathbb{B}(\sigma) \), so the statement is trivially true. Let \( \sigma = \tau \Rightarrow \rho \), then by induction hypothesis \( S_\tau \) and \( S_\rho \) are quasi-ordered. Quasi-ordering of \( (S_\tau \Rightarrow \rho, \sqsubseteq) \) follows from the induced point-wise comparison. A minimum for this size set is the function \( \lambda x. \bot \). Well-foundedness of \( (C_\sigma, \geq, \supseteq) \) follows from Lemma 11 by showing that \( \mathcal{F}_\sigma \Rightarrow \tau \) 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 \( f \in (\sigma) \) and \( x \in (\sigma) \). We define the application of \( f \) to \( x \), denoted \( f \cdot x \), as follows.

Definition 16. Let \( \sigma \Rightarrow \tau \) be an arrow type, \( f = < (n, f^\circ), f^\circ > \in (\sigma \Rightarrow \tau) \), and \( x = < (m, x^\circ), x^\circ > \in (\sigma) \). The semantic application of \( f \) to \( x \), denoted \( f \cdot x \), is defined by:

\[
\begin{align*}
\text{let } f^\circ(x^\circ, x^\circ) &= (k, h); \text{ then }\langle (n, f^\circ), f^\circ \rangle \cdot \langle (m, x^\circ), x^\circ \rangle &= \langle (n + m + k, h), f^\circ(x^\circ) \rangle
\end{align*}
\]

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 \( \sigma = (\text{nat} \Rightarrow \text{nat}) \Rightarrow \text{list} \Rightarrow \text{list} \), which is the type of map defined in Example 1. The function \( \text{map} \) takes as argument a function \( F : \text{nat} \Rightarrow \text{nat} \) 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 \( \text{map} \) is an element \( < (n, f^\circ), f^\circ > \) of \( (\sigma) \). Its cost component \( < n, f^\circ \rangle \) is in \( C_\sigma = \mathbb{N} \times \mathcal{F}_\sigma \) which is composed of a numeric and functional component. The numeric component \( n \) carries the cost of partial application. Meanwhile, the functional component in \( \mathcal{F}_\sigma \) is parametrized by functional arguments carrying the cost and size information of \( F \). Indeed, Definition 14 gives us \( f^\circ : \mathcal{F}_{\text{nat} \Rightarrow \text{nat}} \times S_{\text{nat} \Rightarrow \text{nat}} \Rightarrow C_{\text{list} \Rightarrow \text{list}} \), which can be written explicitly as:
We call the pair $\langle f, x \rangle$ a cost–size tuple of the form $\langle (m, x^c), x^s \rangle$ where $x^c$ is the cost of computing $F$ (so an element of $F^c_{nat \Rightarrow nat}$) and $x^s$ is the size of $F$, so an element of $S_{nat \Rightarrow nat}$. We proceed by applying the respective functions so $f^c(x^c, x^s) = (k, h)$ belongs to $C_{list \Rightarrow list}$ and $f^s(x^s)$ is in $S_{list \Rightarrow list}$. We put everything together and add the numeric components to obtain: $f \cdot x = ((n + m + k, h), f^s(x^s))$. Notice that this gives us a new cost–size tuple with the cost component in $\mathbb{N} \times (C_{list \Rightarrow list})$ and size component in $S_{list \Rightarrow list}$, which is a tuple in $\langle \text{list} \Rightarrow \text{list} \rangle$.

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 $f \in \langle \sigma \Rightarrow \tau \rangle$ and $x \in \langle \sigma \rangle$, then $f \cdot x$ belongs to $\langle \tau \rangle$.

Definition 14 gives us a family of cost–size sets $\mathcal{T} = \{\langle \sigma \rangle\}_{\sigma \in \mathcal{T}_S}$ indexed by $\mathcal{T}_S$, and combined with Definition 16 we get a family of application operators

$$\langle \mathcal{T}, \cdot \rangle = \left(\{\langle \sigma \rangle\}_{\sigma \in \mathcal{T}_S}, \{\cdot, \tau : \langle \sigma \Rightarrow \tau \rangle \times \langle \sigma \rangle \rightarrow \langle \tau \rangle\}\right)$$

We call the pair $\langle \mathcal{T}, \cdot \rangle$ the cost–size type structure generated by the interpretation key $\mathcal{T}_S$. Indeed, in the next Lemma we show that such structure preserves the orderings $\succ$ and $\succeq$ on cost–size tuples.

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

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

$$f \cdot x = ((n + j + k, h), f^s(x^s)) \succ ((m + j' + k', h'), g^s(y^s)) = g \cdot y$$

**Remark 20.** Notice that the type structure $\langle \mathcal{T}, \cdot \rangle$ is nonstandard. Indeed, the intended standard semantics given to arrow types is usually a functional space $[5, \text{Chapter 3}]$. 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: $\text{nat}$ and $\text{list}$. Values of type $\text{nat}$ are built using the constructors $0 : \text{nat}$ and $s : \text{nat} \Rightarrow \text{nat}$. Similarly, for list we have $\text{nil} : \text{list}$ and $\text{cons} : \text{nat} \Rightarrow \text{list} \Rightarrow \text{list}$.

$$\left(\text{unit} \times \mathbb{N} \Rightarrow \mathbb{N} \times \text{unit}\right) \times (S_{\text{nat}} \Rightarrow S_{\text{nat}}) \Rightarrow \left(\mathbb{N} \times \text{unit} \times S_{\text{list}} \Rightarrow \mathbb{N} \times \text{unit}\right)$$

The set for the size function is somewhat simpler with $f^s : (S_{\text{nat}} \Rightarrow S_{\text{nat}}) \Rightarrow S_{\text{list}} \Rightarrow S_{\text{list}}$.
Let us give a cost–size type structure over \( \mathbb{N} \) (Example 13) for \( B = \{ \text{nat}, \text{list} \} \). Essentially, we need to choose the numbers \( K(\text{nat}), K(\text{list}) \) associated with nat and list, respectively. To do so we take the intentional size semantic of nat, list into account. Let us set \( K(\text{nat}) = 1 \) and \( K(\text{list}) = 2 \). 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 \( \text{(length, maximum element size)} \) 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 \( J_{B/N} \) affects the shape of interpretations for symbols in \( \Sigma \), as we will see in Example 23.

Even though we have manually chosen the size tuples for \( J_{B/N} \) above, an automated procedure can still be devised to determine the number \( K(i) \), for \( i \in B \). 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 \( T_B \). Our goal in this section is to interpret terms as elements of those sets.

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

**Definition 22.** A cost–size tuple interpretation \( J \) for a signature \( F = (B, \Sigma, \text{ar}) \) consists of a pair of functions \( (J_B, J_{\Sigma}) \) where

1. \( J_B \) is a type interpretation key (Definition 12).
2. \( J_{\Sigma} \) is an interpretation of symbols in \( \Sigma \) which maps each \( f \in \Sigma \) with \( \text{ar}(f) = \sigma \) to a cost–size tuple in \( \langle \sigma \rangle \), where \( \langle \sigma \rangle \) is built using \( J_B \) in Definition 14.

In what follows we slightly abuse notation by writing \( J_f \) for \( J_{\Sigma}(f) \) and just \( J \) for \( J_B \).

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

\[
J_0 = \langle (0, \text{u}), 1 \rangle \quad J_s = \langle (0, \lambda x.(0, \text{u})), \lambda x.x + 1 \rangle
\]

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 \( 0 \) is 0, (because it is a value), the cost function is \( \text{u} \) (because it has base type), and size component is 1 (since we chose a notion of size for terms of type \( \text{nat} \) 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 \( \lambda x.x + 1 \) in \( S_{\text{nat} = \text{nat}} \). We interpret the constructors for list, i.e., \( \text{nil} \) and \( \text{cons} \), following the same principle, with \( K(\text{list}) = 2 \). We write a size tuple \( q \) in \( S_{\text{list}} \) as \( (q_l, q_m) \) since the first component is to mean the length of the list and the second a bound on the size of its elements.

\[
J_{\text{nil}} = \langle (0, \text{u}), (0, 0) \rangle \quad J_{\text{cons}} = \langle (0, \lambda x.(0, \lambda q.(0, \text{u}))), \lambda xq.(q_l + 1, \max(x, q_m)) \rangle
\]

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 \( F \) to the set of terms. But first, we define valuation functions to interpret the variables in \( x : \sigma \) as elements of \( \langle \sigma \rangle \).
Definition 24. A cost–size valuation $\alpha$ is a function that maps each $x : \sigma$ to a cost–size tuple in $[\sigma]$ such that:

- $\alpha(x) = ([0, u], x^2)$, for all $x \in X$ of base type, and
- $\alpha(F) = ([0, F^c], F^s)$ when $F : \sigma \Rightarrow \tau$.

Notice that, in this definition, the cost component of $\alpha(x)$ has the form $(0, u)$, if $x : \iota$. This interpretation is motivated by Definition 4, where a matching substitution $\gamma$ (i.e., a substitution such that $\ell \gamma \Rightarrow \rho \gamma$) must map each $x : \iota$ to a value of base type. Those can only have the form $c(v_1, \ldots, v_m)$ with $c \in \Sigma_{\text{con}}$. 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 $F : \text{nat} \Rightarrow \text{nat}$ can be instantiated with $\text{add}0$, which is a value that produces a cost as soon as it is applied to the next argument. We use the notation $F^c/F^s$ to denote the cost/size component of $\alpha(F)$.

Definition 25. Assume given a signature $F = (B, \Sigma, a, r)$ and its cost–size tuple interpretation $J = (J_B, J)$ together with a valuation $\alpha$. The term interpretation $[s]^J_\alpha$ of $s$ under $J$ and $\alpha$ is defined by induction on the structure of $s$ as follows:

- $[x]^J_\alpha = \alpha(x)$
- $[f]^J_\alpha = J_f$
- $[s\, t]^J_\alpha = [s]^J_\alpha \cdot [t]^J_\alpha$
- $[\lambda x. s]^J_\alpha = \big( (0, J_{\lambda x} (1 + \pi_1([s]^J_\alpha | x := d|\alpha)), \pi_2([s]^J_\alpha | x := d|\alpha))) \cdot J_{\lambda x} \cdot \lambda d. [s]^J_\alpha | x := (0, d)|\alpha \big)$,

where $\pi_i$ is the projection on the $i$th-component and $\pi_{ij}$ is the composition $\pi_j \circ \pi_i$, and $\emptyset$ is a cost function of the form $\lambda x_1.((0, \lambda x_2 \ldots (0, u) \ldots)$. If $d = (d^c, d^s)$, the notation $[x := d]_\alpha$ denotes the valuation that maps $x$ to $((0, d^c), d^s)$ and every other variable $y$ to $\alpha(y)$.

We write $[s]$ for $[s]^J_\alpha$ whenever $\alpha$ and $J$ 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 the cost function/size pair for $s$ when $x$ is mapped to that size. Technically, to obtain the size component of $[s]^J_\alpha | x := d|\alpha$, we also need a cost component, but by definition, this component does not play a role, so we can safely choose an arbitrary pair $\emptyset$ in the right set.

Example 26. We continue with Example 23 by interpreting ground constructor terms fully. A ground constructor term $d$ of type $\text{nat}$ is of the form $s(s \ldots (s(0)) \ldots$ where the number $n \in \mathbb{N}$ is represented by $n$ successive applications of $s$ to 0. Let us write $n$ as shorthand notation for such terms. Similarly, for ground constructor terms of type list, we write $[n_1; \ldots; n_k]$ for the term $\text{cons}n_1 \ldots (\text{cons}n_k \text{nil})$. The empty list constructor nil is written as $\emptyset$ in this notation. Hence, the cost–size interpretation of $3 : \text{nat}$ is given by:

$$[3] = [s(s(s(0)))] = [s] \cdot ([s] \cdot ([s] \cdot [0])) = \langle (0, u), 4 \rangle.$$ 

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

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

The important information we can extract from such interpretations is their size component. Indeed, $[3]^s = 4$ counts the number of constructor symbols in the term representation $3$ and $[[1; 7; 9]]^s = (3, 10)$ 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 $\text{nat}$ and $\text{list}$ 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 \([s] \in \llbracket \sigma \rrbracket\).

**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 \( \pi_i \). ◀

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 \([s] \triangleright [t] \) whenever \( s \rightarrow t \), for any pair of terms \( s, t \).

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

- an interpretation key \( J_B \) (Definition 12), together with
- a cost–size interpretation \((J_B, J_\Sigma)\) (Definition 22),

such that the following compatibility conditions hold:

1. for all value substitutions \( \gamma \) and all terms \( s \) and \( t \), \([s] \triangleright \gamma \triangleright [t] \) whenever \([s] \triangleright [t] \);
2. for every term \( s \) and value \( v \), \([\lambda x. s v] \triangleright [s[x := v]] \);
3. for all terms \( s \) and \( t \), \([st] \triangleright [st'] \) whenever \([s] \triangleright [s'] \), and \([st] \triangleright [st''] \) whenever \([t] \triangleright [t''] \);
4. for all rules \( \ell \rightarrow r \in R \), we have \([\ell] \triangleright [r] \).

Roughly speaking, a call-by-value termination model is an interpretation of types and terms that is compatible with each rule in \( R \), 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 \( s \rightarrow_v t \), we can establish the following result.

**Theorem 29.** Let \((F, R)\) be a TRS. If we have a termination model of \((F, R)\), then the higher-order call-by-value rewriting relation \( \rightarrow_v \) 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 \( \gamma \) and valuation \( \alpha \), we define the \( \gamma \)-extension of \( \alpha \) as the valuation defined by \( \alpha^\gamma = \llbracket x \alpha \rrbracket \circ \gamma \).

**Lemma 31.** If \( x \notin \text{fv}(s) \) 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 \( \gamma \) and valuation \( \alpha \), we have that \([s\gamma]_\alpha = [s]_{\alpha^\gamma} \).
Lemma 33. The call-by-value beta rule scheme \((\lambda x. t) v \rightarrow s, s[x := v]\) is strictly decreasing for any cost-size interpretation.

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 \(\gamma\). Hence,

\[
\lambda x. (t\gamma) |_\alpha = \langle \langle 0, \lambda d. (1 + \pi_{11}(t\gamma)|_{x := d}|_\alpha), \pi_{12}(t\gamma)|_{x := d}|_\alpha \rangle, \lambda d. \pi_2(t\gamma)|_{x := (d, d)}|_\alpha \rangle \\
\overset{IH}{=} \langle \langle 0, \lambda d. (1 + \pi_{11}(t\gamma)|_{x := d}|_\alpha), \pi_{12}(t\gamma)|_{x := d}|_\alpha \rangle, \lambda d. \pi_2(t\gamma)|_{x := (d, d)}|_\alpha \rangle \\
= \langle \langle 0, \lambda d. (1 + \pi_{11}(t\gamma)|_{x := d}|_\alpha), \pi_{12}(t\gamma)|_{x := d}|_\alpha \rangle, \lambda d. \pi_2(t\gamma)|_{x := (d, d)}|_\alpha \rangle \\
= \|\lambda x. t\|_{\alpha, \gamma}.
\]

As a consequence of the substitution lemma, if \([s]^{\gamma}_\alpha \gg [t]^{\gamma}_\alpha\) for all \(\alpha\), then \([s\gamma]^{\gamma}_\alpha \gg [t\gamma]^{\gamma}_\alpha\) for all \(\alpha\). Consequently, the first compatibility condition is valid for any interpretation. The second compatibility requirement is for \(\alpha\) reductions.

Corollary 34. Let \(R\) be a TRS that admits a cost-size interpretation \((J_\mathbb{R}, J_\Sigma)\). If \([\ell] \gg [r]\) for all rules \(\ell \rightarrow r\) in \(R\), we collect this fact below, which is a consequence of Theorem 29 and the Lemmas above.

Corollary 34. Let \(R\) be a TRS that admits a cost-size interpretation \((J_\mathbb{R}, J_\Sigma)\). If \([\ell] \gg [r]\) for all rules \(\ell \rightarrow r\) in \(R\), then \(R\) 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 \(a, b : \iota, g : \iota \Rightarrow \iota \Rightarrow \iota, \) and \(f : \iota \Rightarrow \iota \Rightarrow \iota\). The rewrite system introduced by Toyama [30] and defined by \(R = \{ g x y \rightarrow x, g x y \rightarrow y, f a b z \rightarrow f z z z \}\) was given to show that termination is not modular for disjoint unions of TRSs. Indeed, it admits the
This notion is defined for all terms when the TRS is terminating. We will simply refer to the notation of Example 7, we could alternatively write \( \lambda \lambda xy.x \) captures the cost of partially applying terms, which is zero. The reason for this choice is that in a cost tuple \( \lambda \lambda xy.x \), we set the intermediate numeric components in the cost tuple to \( x = y = 0 \). We also set the intermediate numeric components in the cost tuple to \( x = y = 0 \). By the lemmas in Section 5 we see that \( J_s \) and \( J_t \) is weakly monotonic, and the size tuples for interpretation of values are \( \lambda \lambda xy.x \cup y \). Notice that \( K_{xy} = \lambda xy.x \) and \( J_{xy} = \lambda xy.x \cup y \). This set is partially ordered by inclusion, so we define \( x \geq y \) if \( x \geq y \) which is a quasi-order. Consider the following interpretation:

\[
\begin{align*}
J_a &= \left\{ (0, u), \{a\} \right\} \\
J_b &= \left\{ (0, u), \{b\} \right\} \\
J_\alpha &= \left\{ (0, \lambda x.(0, \lambda y.(1, u))), \lambda xy.x \cup y \right\} \\
J_t &= \left\{ (0, \lambda x.(0, \lambda y.(0, \lambda z.(H(x, y), u))))), \lambda xy.z \right\},
\end{align*}
\]

where \( H \) is a helper function defined by \( H(x, y) = \text{if } x^a \supseteq \{a\} \land y^a \supseteq \{b\} \text{ then } 1 \text{ else } 0 \). Notice that \( H \) is weakly monotonic, and the size tuples for interpretation of values are sets of cardinality \( \leq 1 \). Checking compatibility for this interpretation is straightforward:

\[
\begin{align*}
[g xy] &= \langle (1, u), x \cup y \rangle \succ \langle (0, u), x \rangle = \langle x \rangle \\
g x y &= \langle (1, u), x \cup y \rangle \succ \langle (0, u), y \rangle = \langle y \rangle;
\end{align*}
\]

and finally \( [f a b z] = \langle (1, u), \emptyset \rangle \succ \langle (0, u), \emptyset \rangle = \langle f z z \rangle \), 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 \( \text{dh}_\mathbb{R}(s) \), is the largest number \( n \) such that \( s \rightarrow_v s_1 \rightarrow_v \ldots \rightarrow_v s_n \).

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 \( [s] = \langle (n, F^c), F^s \rangle \), then \( \text{dh}_\mathbb{R}(s) \leq n \).

**Proof.** By the lemmas in Section 5 we see that \( [s] \sim [t] \) whenever \( s \rightarrow t \). Since this implies \( \pi_{11}(s) \geq \pi_{11}(t) \), 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 \( \mathbb{R}_{\text{add}} \) which intuitively defines addition over \( \text{nat} \). We will use the type and constructor interpretations as given in Example 23. The rules \( \text{add} x 0 \rightarrow 0 \) and \( \text{add} x (s y) \rightarrow s(\text{add} x y) \) suggest the following cost–size interpretation:

\[
\begin{align*}
J_{\text{add}} &= \left\{ (0, \lambda x.(0, \lambda y.(y^a, u))), \lambda xy.x + y \right\}.
\end{align*}
\]

Notice that the (highlighted) cost component of \( J_{\text{add}} \) suggest a linear cost measure for computing with \( \text{add} \). We also set the intermediate numeric components in the cost tuple to zero. The reason for this choice is that in a cost tuple \( C_\sigma = \mathbb{N} \times F_\sigma \), 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 \( J_{\text{add}} = \langle (x^a, y^a) \rightarrow y^a, \lambda x^a y^a.x^a + y^a \rangle \).
Now, consider the partially applied term \( s = \text{add}(\text{add} 2 3) \) (of type \( \text{nat} \Rightarrow \text{nat} \)). Intuitively, the cost of reducing this term to normal form, is the cost of reducing the subterm \( \text{add} 2 3 \) to 5, since the partially applied term \( \text{add} 5 \) cannot be reduced. Hence, \( \text{dhr}(s) = 4 \). This is also the bound we find through interpretation:

\[
[s] = [\text{add}] \cdot ([\text{add}] \cdot [2] \cdot [3])
\]

\[
= [\text{add}] \cdot ([4, u], 7)
\]

\[
= \left\{ (4, \lambda y. (y^2, u)), \lambda y. 7 + y \right\}.
\]

While in this case the bound we find is tight, this is not always the case; for instance \([\text{add} 0 (\text{add} 0 0)] = ((3, u), 3)\), even though \( \text{dhr}([\text{add} 0 (\text{add} 0 0)]) = 2 \). 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 \( f s_1 \ldots s_n \) 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 \( \text{add} \) rules. Then, we might construct a term \((\lambda x. \text{add} x x)((\lambda x. \text{add} x x)((\ldots (\lambda x. \text{add} x x))\ldots))\), with \( n \) occurrences of \((\lambda x. \text{add} x x)\). The size of this term is linear in \( n \), but its derivation height is exponential, since each contraction of a \( \lambda \) 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 \( \lambda \).

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 \( \iota_1 \Rightarrow \ldots \Rightarrow \iota_m \Rightarrow \kappa \), with \( \kappa \) and all \( \iota_i \) base types.

A *data term* is a value of the form \( \text{cd}_1 \ldots \text{d}_m \) with \( \text{c} : \iota_1 \Rightarrow \ldots \Rightarrow \iota_m \Rightarrow \kappa \), a data constructor, and each \( \text{d}_i \) 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 \( \text{f} \text{d}_1 \ldots \text{d}_m \) with \( \text{f} \in \Sigma^{\text{def}} \), a defined symbol and all \( \text{d}_i \) data terms.

The *weak call-by-value runtime complexity* of a TRS is the function \( n \mapsto \text{rc}(n) \) that maps each natural number \( n \) to the largest number \( h \) with \( \text{dhr}(s) = h \) 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 \( \text{start} x \rightarrow r \) 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 \( \text{dbl} \) and \( \text{mult} \) from Example 2.

\[
\mathcal{J}_{\text{dbl}} = \left\{ (0, \lambda x. (x^2, u)), \lambda x. 2x \right\}
\]

\[
\mathcal{J}_{\text{mult}} = \left\{ (0, \lambda x. (0, \lambda y. 2x^2y^2, u)), \lambda xy. xy \right\}
\]
In the TRS of Example 2, the only basic terms have the form \texttt{add} \(v_1 v_2\) or \texttt{dbl} \(v\) or \texttt{mult} \(v_1 v_2\). Since \(\sigma^n 0^n = n + 1\), Lemma 37 allows us to conclude that \(r_c(n) < n^2\).

Now, the size bound for data constructors introduced in Example 23 is well-behaved. However, suppose we had defined \(J_0 = ((0, u), 1)\) and \(J_n = ((0, \lambda x.((0, u)), \lambda x.2x + 1))\). In this case, for a data term \(n = s^n 0\), we would have \(n^n = 2^n + n > 2^n\). As a result, we would only be able to derive exponential runtime complexity. Notice that this choice is compatible with \(R_{\text{add}}\), and hence proves its termination; however, it induces an exponential overhead on the cost tuple of \texttt{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 \textit{additively}.

Let \(c\) be a data constructor of type \(\sigma = t_1 \Rightarrow \ldots \Rightarrow t_m \Rightarrow \kappa\). The size component of \(\{\sigma\}\) is \(S_c = \mathbb{N}^{K(t_1)} \Rightarrow \ldots \Rightarrow \mathbb{N}^{K(t_m)} \Rightarrow \mathbb{N}^{K(\kappa)}\). The size tuple \(J_c^2\) when fully applied can be written in terms of its functional components. Hence, \(J_c^2(x_1, \ldots, x_m) = \left\langle f_1^1(x_1, \ldots, x_m), \ldots, f_{K(\kappa)}^1(x_1, \ldots, x_m) \right\rangle\).

\textbf{Definition 41.} If \(c : \sigma\) is a data constructor as above, we say \(J_c^2\) is \textit{additive} if there is a constant \(a \in \mathbb{N}\) such that \(\sum_{i=1}^m f_i(x_1, \ldots, x_m) \leq a + \sum_{j=1}^m \sum_{i=1}^m x_{ij}\).

It is easy to show that size components for \texttt{nat} and \texttt{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 \(n\): \(|d|^n \leq an\). Hence, for instance the following result from [21] also extends to our setting:

\textbf{Lemma 42 (From [21]; Corollary 33).} Let \(R\) 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 \(R\) 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 \texttt{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 \texttt{map} below:

\(J_{\text{map}} = \left\langle (0, \lambda F.(0, \lambda q.(q + F^c(u, q_m)q_1 + 1, u))), \lambda F.q.(q, F(q_m)) \right\rangle\).

The highlighted cost component for \(q\) possible \(\beta\) steps, the cost of applying the higher-order argument \(F\) over the list \(q\) is bounded by \(F^c(u, q_m)q\) 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.

\textbf{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 \(\lambda x.a\) in \(X \Rightarrow Y\);
- for \(f : X \Rightarrow Y\) and \(g : Y \Rightarrow Z\), we write \(g \circ f : X \Rightarrow Z\) as the composition of \(f\) and \(g\).
- the projection function on the \(i\)th coordinate, \(\pi_i : X_1 \times \cdots \times X_k \Rightarrow X_i\);
- given \(f : X \Rightarrow Y\) and \(g : X \Rightarrow Z\), we have a function \((f, g) : X \Rightarrow Y \times Z\) which is defined by \((f, g)(x) = (f(x), g(x))\);
These basic weakly monotonic functions provide the building blocks for constructing cost–size interpretations. A second step is to expand other complexity methods for innermost/call-by-value reduction. In this approach, we build on existing work defining tuple interpretations. 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:

\[
H^c(x, y, F) = \sum_{i=1}^{x^2-1} \pi_1(F^c((u, i), (u, H^s(i, y^2, F^s))))
\]

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

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

### 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 \(\mathbb{N}\), 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


