Tuple Interpretations and Applications to Higher-Order Runtime Complexity

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

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

Abstract

Tuple interpretations are a class of algebraic interpretation that subsumes both polynomial and matrix interpretations as it does not impose simple termination and allows non-linear interpretations. It was developed in the context of higher-order rewriting to study derivational complexity of algebraic functional systems. In this short paper, we continue our journey to study the complexity of higher-order TRSs by tailoring tuple interpretations to deal with innermost runtime complexity.

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

Keywords and phrases Complexity analysis, higher-order term rewriting, tuple interpretations

Funding The authors are supported by the NWO TOP project "ICHOR", NWO 612.001.803/7571 and the NWO VIDI project "CHORPE", NWO VI.Vidi.193.075.

1 Introduction

The step-by-step computational model induced by term rewriting naturally gives rise to a *complexity* notion. Here, complexity is understood as the number of rewriting steps needed to reach a normal form. In the rewriting setting, a *complexity function* bounds the length of the longest rewrite sequence parametrized by the size of the starting term. Two distinct complexity notions are commonly considered: derivational and runtime. In the former, the starting term is unrestricted which allows initial terms with nested function calls. The latter only considers rewriting sequences beginning with *basic* terms. Intuitively, basic terms are those where a single function call is performed with *data* objects as arguments.

There are many techniques to bound the runtime complexity of term rewriting [2, 4]. However, most of the literature focuses on the first-order case. We take a different approach and regard higher-order term rewriting. We present a technique that takes advantage of tuple interpretations [3] tailored to deal with an innermost rewriting strategy. The defining characteristic of tuple interpretations is to allow for a split of the complexity measure into abstract notions of *cost* and *size*. The former is usually interpreted as natural numbers, which accounts for the number of steps needed to reduce terms to normal forms. Meanwhile, the latter is interpreted as tuples over naturals carrying abstract notions of size.

2 Preliminaries

The Syntax of Terms and Rules We assume familiarity with the basics of term rewriting. We will here recall notation for *applicative simply-typed term rewriting systems*.

Let $\mathcal{B}$ be a set of base types (or sorts). The set $\mathcal{T}_{\mathcal{B}}$ of *simple types* is built using the right-associative $\Rightarrow$ as follows. Every $\iota \in \mathcal{B}$ is a type of order 0. If σ, τ are types of order n and m respectively, then $\sigma \Rightarrow \tau$ is a type of order $\max(n+1, m)$. A signature is a non-empty set $\mathcal{F}$ of function symbols together with a function $\mathtt{typeOf} : \mathcal{F} \longrightarrow \mathcal{T}_{\mathcal{B}}$. Additionally, we assume, for each $\sigma \in \mathcal{T}_{\mathcal{B}}$, a countable infinite set of type-annotated variables $\mathcal{X}_\sigma$ disjoint from $\mathcal{F}$. We will denote $\mathtt{f}, \mathtt{g}, \ldots$ for function symbols and $x, y, \ldots$ for variables.

This typing scheme imposes a restriction on the formation of terms which consists of those expressions s such that $s :: \sigma$ can be derived for some type σ using the following clauses: (i) $x :: \sigma$, if $x \in \mathcal{X}_\sigma$; (ii) $\mathsf{f} :: \sigma$, if $\mathtt{typeOf}(\mathsf{f}) = \sigma$; and (iii) $(s\,t) :: \tau$, if $s :: \sigma \Rightarrow \tau$ and $t :: \tau$. Application is left-associative. We denote $\mathtt{vars}(s)$ for the set of variables occurring in s and say s is *ground* if $\mathtt{vars}(s) = \emptyset$. A rewriting rule $\ell \to r$ is a pair of terms of the same type such that $\ell = \mathsf{f}\,\ell_1 \dots \ell_m$ and $\mathtt{vars}(\ell) \supseteq \mathtt{vars}(r)$. An *applicative simply-typed term rewriting system* (shortly denoted TRS), is a set $\mathcal{R}$ of rules. The rewrite relation induced by $\mathcal{R}$ is the smallest monotonic relation that contains $\mathcal{R}$ and is stable under application of substitution. A term s is in *normal form* if there is no t such that $s \to t$. The *innermost rewrite relation* induced by $\mathcal{R}$ is defined as follows:

- $\ell\gamma \to^i r\gamma$, if $\ell \to r \in \mathcal{R}$ and all proper subterms of $\ell\gamma$ are in $\mathcal{R}$-normal form;
- $s\,t \to^i s'\,t$, if $s \to^i s'$; and $s\,t \to^i s\,t'$, if $t \to^i t'$.

In what follows we only allow for innermost reductions. So, we drop the i from the arrow, and $s \to t$ is to be read as $s \to^i t$. We shall use the explicit notation if confusion may arise.

▶ **Example 1.** We will use a system over the sorts nat (numbers) and list (lists of numbers). Let $\mathsf{0} :: \mathsf{nat}$, $\mathsf{s} :: \mathsf{nat} \Rightarrow \mathsf{nat}$, $\mathsf{nil} :: \mathsf{list}$, $\mathsf{cons} :: \mathsf{nat} \Rightarrow \mathsf{list} \Rightarrow \mathsf{list}$, and $F, G \in \mathcal{X}_{\mathsf{nat}\Rightarrow\mathsf{nat}}$; types of other function symbols and variables can be easily deduced.

$$
\begin{array}{rcl@{\qquad}rcl}
\mathtt{map}\,F\,\mathsf{nil} & \to & \mathsf{nil} & \mathtt{comp}\,F\,G\,x & \to & F\,(G\,x) \\
\mathtt{map}\,F\,(\mathsf{cons}\,x\,xs) & \to & \mathsf{cons}\,(F\,x)\,(\mathtt{map}\,F\,xs) & \mathtt{app}\,F\,x & \to & F\,x \\
\mathtt{d}\,\mathsf{0} & \to & \mathsf{0} & \mathtt{add}\,x\,\mathsf{0} & \to & x \\
\mathtt{d}\,(\mathsf{s}\,x) & \to & \mathsf{s}\,(\mathsf{s}\,(\mathtt{d}\,x)) & \mathtt{add}\,x\,(\mathsf{s}\,y) & \to & \mathsf{s}\,(\mathtt{add}\,x\,y)
\end{array}
$$

Functions and orderings A quasi-ordered set $(A, \sqsupseteq)$ consists of a nonempty set A and a quasi-order $\sqsupseteq$ over A. A well-founded set $(A, >, \geq)$ is a nonempty set A together with a well-founded order $>$ and a compatible quasi-order $\geq$ on A, i.e., $> \circ \geq \subseteq >$. For quasi-ordered sets A and B, we say that a function $f : A \longrightarrow B$ is weakly monotonic if for all $x, y \in A$, $x \sqsupseteq_A y$ implies $f(x) \sqsupseteq_B f(y)$. If $(B, >, \geq)$ is a well-founded set, then $>$ and $\geq$ induce a point-wise comparison on $A \longrightarrow B$ as usual. If A, B are quasi-ordered, the notation $A \Longrightarrow B$ refers to the set of all weakly monotonic functions from A to B. Functional equality is extensional. The unit set is the quasi-ordered set defined by $\mathtt{unit} = (\{\mathtt{u}\}, \sqsupseteq)$, where $\mathtt{u} \sqsupseteq \mathtt{u}$.

3 Higher-Order Tuple Interpretations for Innermost Rewriting

To define interpretations, we will start by providing an interpretation of *types* (Def. 2). Types σ are interpreted by tuples $(\!|\sigma|\!)$ that carry information about cost and size. We will first show how application works in this newly defined cost-size domain (Def. 4). Interpretation of types will then set the domain for the tuple algebras we are interested in (Def. 7).

▶ **Definition 2** (Interpretation of Types). *For each type σ, we define the cost-size tuple interpretation of σ as $(\!|\sigma|\!) = \mathcal{C}_\sigma \times \mathcal{S}_\sigma$ where $\mathcal{C}_\sigma$ (respectively $\mathcal{S}_\sigma$) is defined as follows:*

$$
\begin{aligned}
\mathcal{C}_\sigma &= \mathbb{N} \times \mathcal{F}^{\mathsf{c}}_\sigma & \mathcal{S}_\iota &= (\mathbb{N}^{K[\iota]}, \sqsupseteq),\ K[\iota] \geq 1 \\
\mathcal{F}^{\mathsf{c}}_\iota &= \mathtt{unit} & \mathcal{S}_{\sigma\Rightarrow\tau} &= \mathcal{S}_\sigma \Longrightarrow \mathcal{S}_\tau \\
\mathcal{F}^{\mathsf{c}}_{\sigma\Rightarrow\tau} &= (\mathcal{F}^{\mathsf{c}}_\sigma \times \mathcal{S}_\sigma) \Longrightarrow \mathcal{C}_\tau,
\end{aligned}
$$

where $\mathcal{F}^{\mathsf{c}}_{\sigma\Rightarrow\tau}$ ($\mathcal{S}_{\sigma\Rightarrow\tau}$) is the set of weakly monotonic functions from $\mathcal{F}^{\mathsf{c}}_\sigma \times \mathcal{S}_\sigma$ to $\mathcal{C}_\tau$ ($\mathcal{S}_\sigma$ to $\mathcal{S}_\tau$). The quasi-ordering on those sets is the induced point-wise comparison. The set $(\!|\sigma|\!)$ is ordered as follows: $((n, f), s) \succ ((m, g), t)$ if $n > m$, $f \geq g$ and $s \sqsupseteq t$; and $((n, f), s) \succcurlyeq ((m, g), t)$ if $n \geq m$, $f \geq g$ and $s \sqsupseteq t$.

The cost tuple $\mathcal{C}_\sigma = \mathbb{N} \times \mathcal{F}^{\mathsf{c}}_\sigma$ of $(\!|\sigma|\!)$ holds the cost information of reducing a term of type σ to its normal form. It is composed of a numeric and functional component. Base types, which are naturally not functional, have the unit set for $\mathcal{F}^{\mathsf{c}}_\iota$; the cost tuple of a base type is then $\mathcal{C}_\iota = \mathbb{N} \times \mathtt{unit}$. Functional types do possess an intrinsically functional component (the cost of *applying* the function), which in our setting is expressed by $\mathcal{F}^{\mathsf{c}}_{\sigma\Rightarrow\tau} = \mathcal{F}^{\mathsf{c}}_\sigma \times \mathcal{S}_\sigma \Longrightarrow \mathcal{C}_\tau$. For functional types the numeric component represents the cost of partial application.

To determine the number $K[\iota]$, associated to each sort ι, we use a semantic approach that takes the intuitive meaning of the sort we are interpreting into account. The sort nat for instance represents natural numbers, which we implement in unary format. Hence, it makes sense to reckon the number of successor symbols occurring in terms of the form $(\mathsf{s}^n\ \mathsf{0}) :: \mathsf{nat}$ as their *size*. This gives us $K[\mathsf{nat}] = 1$. Another example is the sort list (of natural numbers): it is natural to regard measures like *length* and *maximum element size*. This results in $K[\mathsf{list}] = 2$. Example 8 below shows how to interpret data constructors using this intuition.

The next lemma expresses the soundness of our approach, that is, cost-size tuples define a well-founded domain for the interpretation of types.

▶ **Lemma 3.** *For each type σ, the set $\mathcal{C}_\sigma$ is well-founded and $\mathcal{S}_\sigma$ quasi-ordered. Their product, that is, $((\!|\sigma|\!), \succ, \succcurlyeq)$, is well-founded.*

Semantic Application To interpret each term $s :: \sigma$ to an element of $(\!|\sigma|\!)$ (Def. 7), we will need a notion of application for cost-size tuples. Specifically, given a functional type $\sigma \Rightarrow \tau$, a cost-size tuple $\boldsymbol{f} \in (\!|\sigma \Rightarrow \tau|\!)$, and $\boldsymbol{x} \in (\!|\sigma|\!)$, our goal is to define the application $\boldsymbol{f} \cdot \boldsymbol{x}$ of $\boldsymbol{f}$ to $\boldsymbol{x}$. Let us illustrate the idea with a concrete example: consider the type $\sigma = (\mathsf{nat} \Rightarrow \mathsf{nat}) \Rightarrow \mathsf{list} \Rightarrow \mathsf{list}$, which is the type of `map` defined in Example 1. The function `map` takes as argument a function F of type $\mathsf{nat} \Rightarrow \mathsf{nat}$ and a list q, and applies F to each element of q. The cost interpretation of `map` is a functional in $\mathcal{C}_\sigma$ parametrized by functional arguments carrying the cost and size information of F and a cost-size tuple for q.

$$\mathbb{N} \times \overbrace{(\underbrace{(\mathtt{unit} \times \mathbb{N} \Longrightarrow \mathbb{N} \times \mathtt{unit})}_{\text{cost of } F} \times \underbrace{(\mathbb{N} \Longrightarrow \mathbb{N})}_{\text{size of } F} \Longrightarrow (\mathbb{N} \times (\underbrace{\mathtt{unit}}_{\text{cost of } q} \times \underbrace{\mathbb{N}^2}_{\text{size of } q} \Longrightarrow \mathbb{N} \times \mathtt{unit})))}^{\text{the functional cost of } \mathtt{map}},$$

Hence, we write an element of such space as the tuple (n, f^{c}). Size sets are somewhat simpler with $\underbrace{(\mathbb{N} \Longrightarrow \mathbb{N})}_{\text{size of } F} \Longrightarrow \underbrace{\mathbb{N}^2}_{\text{size of } q} \Longrightarrow \mathbb{N}^2$. Therefore, a functional cost-size tuple $\boldsymbol{f}$ is represented by $\boldsymbol{f} = \langle (n, f^{\mathsf{c}}), f^{\mathsf{s}} \rangle$. An argument to such a cost-size tuple is then an element in the domain of f^{c} and f^{s}, respectively. Therefore, we apply $\boldsymbol{f}$ to a cost-size tuple $\boldsymbol{x}$ of the form $\langle (m, g^{\mathsf{c}}), g^{\mathsf{s}} \rangle$ where g^{c} is the cost of computing F and g^{s} is the size of F. We proceed by applying the respective functions, so $f^{\mathsf{c}}(g^{\mathsf{c}}, g^{\mathsf{s}}) = (k, h)$ belongs to $\mathcal{C}_{\mathsf{list}}$, and add the numeric components together obtaining: $\boldsymbol{f} \cdot \boldsymbol{x} = \langle (n + m + k, f^{\mathsf{c}}(g^{\mathsf{c}}, g^{\mathsf{s}})), f^{\mathsf{s}}(g^{\mathsf{s}}) \rangle$. Notice that this gives us a new cost-size tuple with cost component in $\mathbb{N} \times (\mathcal{C}_{\mathsf{list}} \Longrightarrow \mathcal{C}_{\mathsf{list}})$ and size component in $\mathcal{S}_{\mathsf{list}} \Longrightarrow \mathcal{S}_{\mathsf{list}}$.

▶ **Definition 4.** *Let $\sigma \Rightarrow \tau$ be an arrow type, $\boldsymbol{f} = \langle (n, f^{\mathsf{c}}), f^{\mathsf{s}} \rangle \in (\!|\sigma \Rightarrow \tau|\!)$, and $\boldsymbol{x} = \langle (m, g^{\mathsf{c}}), g^{\mathsf{s}} \rangle \in (\!|\sigma|\!)$. The application of $\boldsymbol{f}$ to $\boldsymbol{x}$, denoted $\boldsymbol{f} \cdot \boldsymbol{x}$, is defined by:*

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

Semantic application is left-associative and respects a form of application rule.

▶ **Lemma 5.** *If $\boldsymbol{f}$ is in $(\!|\sigma \Rightarrow \tau|\!)$ and $\boldsymbol{x}$ is in $(\!|\sigma|\!)$, then $\boldsymbol{f} \cdot \boldsymbol{x}$ belongs to $(\!|\tau|\!)$.*

▶ Remark 6. In order to ease notation, we project sets $\pi_1 : A \times \mathtt{unit} \longrightarrow A$ and $\pi_2 : \mathtt{unit} \times A \longrightarrow A$ and compose functions with projections, so a function in $\mathtt{unit} \times A \Longrightarrow B \times \mathtt{unit}$ is lifted to a function in $A \Longrightarrow B$. The functional cost of `map` is then read as follows:

$$\mathbb{N} \times \overbrace{(\underbrace{(\mathbb{N} \Longrightarrow \mathbb{N})}_{\text{cost of } F} \times \underbrace{(\mathbb{N} \Longrightarrow \mathbb{N})}_{\text{size of } F} \Longrightarrow (\mathbb{N} \times (\underbrace{\mathbb{N}^2}_{\text{size of } q} \Longrightarrow \mathbb{N})))}^{\text{the functional cost of } \mathtt{map}}$$

The $\mathbb{N}$ component of $C_{\sigma \Rightarrow \tau}$ is specific to innermost rewriting (it does not occur in [3]). We need this to handle rules of non-base type; for example, if $\mathtt{add}\,\mathsf{0} \to \mathtt{id}$, then the cost tuple of $\mathtt{add}\,\mathsf{0}$ is $(1, \boldsymbol{\lambda} x.0)$. However, since in *most* cases the first component is 0, we will typically omit these zeroes and simply write for instance $\boldsymbol{\lambda} F q.f^{\mathsf{c}}(F, q)$ instead of $(0, \boldsymbol{\lambda} F.\langle 0, \boldsymbol{\lambda} q.f^{\mathsf{c}}(F, q)\rangle)$. To compute using Definition 4 we still use the complete form.

Tuple algebras are higher-order weakly monotonic algebras [1] with cost-size tuples as interpretation domain.

▶ **Definition 7** (Higher-order tuple algebra). *A higher-order tuple algebra over a signature* $(\mathcal{B}, \mathcal{F}, \mathtt{typeOf})$ *consists of: (i) a family of cost/size tuples* $\{(\!|\sigma|\!)\}_{\sigma \in \mathcal{T}_{\mathcal{B}}}$ *and (ii) an interpretation function* $\mathcal{J}$ *which maps each* $\mathsf{f} \in \mathcal{F}$ *of type* σ *to a cost-size tuple in* $(\!|\sigma|\!)$.

▶ **Example 8.** Following the semantics discussed previously, we interpret the constructors for both nat and list as follows. We call the first component of $\mathcal{S}_{\mathsf{list}}$ *length* and the second *maximum element size*. Those are abbreviated using the letters l and m, respectively.

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

The cost-size tuples for $\mathsf{0}$ and nil are all 0s, as expected. The size components for s and cons describe the increase in size when new data is created. We interpret functions from Example 1 as follows:

$$\begin{aligned}
\mathcal{J}_{\mathtt{app}} &= \langle \boldsymbol{\lambda} F x.F^{\mathsf{c}}(x) + 1, \boldsymbol{\lambda} F x.F^{\mathsf{s}}(x)\rangle \\
\mathcal{J}_{\mathtt{d}} &= \langle \boldsymbol{\lambda} x.x + 1, \boldsymbol{\lambda} x.2x\rangle \\
\mathcal{J}_{\mathtt{add}} &= \langle \boldsymbol{\lambda} xy.y + 1, \boldsymbol{\lambda} xy.x + y\rangle \\
\mathcal{J}_{\mathtt{comp}} &= \langle \boldsymbol{\lambda} FGx.F^{\mathsf{c}}(G^{\mathsf{s}}(x_{\mathsf{s}})) + 1, \boldsymbol{\lambda} FGx.F^{\mathsf{s}}(G^{\mathsf{s}}(x))\rangle \\
\mathcal{J}_{\mathtt{map}} &= \langle \boldsymbol{\lambda} Fq.q_{\mathsf{l}}F^{\mathsf{c}}(q_{\mathsf{m}}) + 1, \boldsymbol{\lambda} Fq.\langle q_{\mathsf{l}}, F^{\mathsf{s}}(q_{\mathsf{m}})\rangle\rangle
\end{aligned}$$

A valuation α is a function that maps each $x :: \sigma$ to a cost-size tuple in $(\!|\sigma|\!)$. Due to innermost strategy, we can assume the interpretation of every variable $x :: \iota$ has zero cost. This is formalized by assigning $\alpha(x) = \langle (0, \mathtt{u}), x^s\rangle$, for all $x \in \mathcal{X}$ of base type. In this paper, we shall only consider valuations that satisfy this property. Variables of functional type, however, may carry cost information even though any instance of a redex needs to be normalized. Hence, we set $\alpha(F) = \langle (0, f^{\mathsf{c}}), f^{\mathsf{s}}\rangle$ when $F :: \sigma \Rightarrow \tau$.

▶ **Definition 9.** *We extend* $\mathcal{J}$ *to an interpretation* $[\![\cdot]\!]_{\alpha,\mathcal{J}}$ *of terms as follows:*

$$[\![x]\!]_{\alpha,\mathcal{J}} = \alpha(x) \qquad [\![\mathsf{f}]\!]_{\alpha,\mathcal{J}} = \langle (n, \mathcal{J}_{\mathsf{f}}^{\mathsf{c}}), \mathcal{J}_{\mathsf{f}}^{\mathsf{s}}\rangle, n \in \mathbb{N} \qquad [\![s\,t]\!]_{\alpha,\mathcal{J}} = [\![s]\!]_{\alpha,\mathcal{J}} \cdot [\![t]\!]_{\alpha,\mathcal{J}}$$

We are interested in interpretations satisfying a compatibility requirement:

▶ **Theorem 10** (Innermost Compatibility Theorem). *Let* α *be a valuation. If* $[\![\ell]\!]_{\alpha,\mathcal{J}} \succ [\![r]\!]_{\alpha,\mathcal{J}}$ *for all rules* $\ell \to r \in \mathcal{R}$, *then* $[\![s]\!]_{\alpha,\mathcal{J}} \succ [\![t]\!]_{\alpha,\mathcal{J}}$, *whenever* $s \to^i_{\mathcal{R}} t$.

One can check that the TRS from Example 1 interpreted as in Example 8 satisfy the compatibility requirement.

4 Higher-Order Innermost Runtime Complexity

In this section, we briefly limn how the cost-size tuple machinery allow us to reason about innermost runtime complexity. We start by reviewing basic definitions.

▶ **Definition 11.** *A symbol* $\mathsf{f} \in \mathcal{F}$ *is a defined symbol if it occurs at the head of a rule, i.e., there is a rule* $\mathsf{f}\,\ell_1 \ldots \ell_k \to r \in \mathcal{R}$*. A symbol* c *of order at most 1 is a data constructor if it is not a defined symbol. A data term has the form* $\mathsf{c}\,d_1 \ldots d_k$ *with* c *a constructor and each* d_i *a data term. A term* s *is basic if* $s :: \iota$ *and* s *is of the form* $\mathsf{f}\,d_1 \ldots d_m$ *with* f *a defined symbol and all* $d_1, \ldots, d_m$ *data terms. The set* $T_{\mathcal{B}}(\mathcal{F})$ *collects all basic terms.*

▶ Remark 12. Notice that our notion of data is intrinsically first-order. This is motivated by applications of rewriting to full program analysis where even if higher-order functions are used a program has type $\iota_1 \Rightarrow \ldots \Rightarrow \iota_m \Rightarrow \kappa$. The sorts ι_i are the input data types and κ the output type of the program.

▶ **Definition 13.** *The innermost derivation height of* s *is* $\mathtt{dh}_{\mathcal{R}}(s) = \{n \mid \exists t : s \to^n t\}$*. The innermost runtime complexity function with respect to a TRS* $\mathcal{R}$ *is* $\mathtt{irc}_{\mathcal{R}}(n) = \max\{\mathtt{dh}_{\mathcal{R}}(s) \mid s \in T_{\mathcal{B}}(\mathcal{F}) \wedge |s| \leq n\}$*.*

To reasonably bound the innermost runtime complexity of a TRS $\mathcal{R}$, we require that size interpretations of constructors have their components bounded by an additive polynomial, that is, a polynomial of the form $\boldsymbol{\lambda} x_1 \ldots x_k. \sum_{i=1}^{k} x_i + a$, with $a \in \mathbb{N}$.

We can build programs by adding a new `main` function taking data variables as arguments and combine it with rules computing functions, including higher-order ones. For instance, using rules from Example 1, we can compute a program that adds a number x to every element in a list q as follows: $\mathtt{main}\,x\,q \to \mathtt{map}\,(\mathtt{add}\,x)\,q$. Hence, computing this program on inputs n and list q is equivalent to reducing the term $\mathtt{main}\,\mathsf{n}\,q$ to normal form. Its runtime complexity is therefore bounded by the cost-tuple of $[\![\mathtt{main}\,\mathsf{n}\,q]\!]$.

5 Conclusion

In this short paper, we shed light on how to use cost-size tuple interpretations to bound innermost runtime complexity of higher-order systems. We defined a new domain of interpretations that takes the intricacies of innermost rewriting into account and defined how application works in this setting. The compatibility result allows us to make use of interpretations as a way to bound the length of derivation chains, as it is expected from an interpretation method. As current, and future work, we are working on automation techniques to find interpretations and develop a completely rewriting-based automated tool for complexity analysis of functional programs.

References

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

2 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.

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

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